

Chapter 6

Thread Programming

Several parallel computing platforms, in particular multicore platforms, offer a shared address space. A natural programming model for these architectures is a thread model in which all threads have access to shared variables. These shared variables are then used for information and data exchange. To coordinate the access to shared variables, synchronization mechanisms have to be used to avoid race conditions in case of concurrent accesses. Basic synchronization mechanisms are lock synchronization and condition synchronization, see Sect. 3.8 for an overview.

In this chapter, we consider thread programming in more detail. In particular, we have a closer look at synchronization problems like deadlocks or priority inversion that might occur and present programming techniques to avoid such problems. Moreover, we show how basic synchronization mechanisms like lock synchronization or condition synchronization can be used to build more complex synchronization mechanisms like read/write locks. We also present a set of parallel patterns like task based or pipelined processing that can be used to structure a parallel application. These issues are considered in the context of popular programming environments for thread-based programming to directly show the usage of the mechanisms in practice. The programming environments Pthreads, Java threads, and OpenMP are introduced in detail. For Java, we also give an overview of the package `java.util.concurrent` which provides many advanced synchronization mechanisms as well as a task-based execution environment. The goal of the chapter is to enable the reader to develop correct and efficient thread programs that can be used, for example, on multicore architectures.

6.1 Programming with Pthreads

POSIX threads (also called Pthreads) defines a standard for the programming with threads, based on the programming language C. The threads of a process share a common address space. Thus, the global variables and dynamically generated data

objects can be accessed by all threads of a process. In addition, each thread has a separate runtime stack which is used to control the functions activated and to store their local variables. These variables declared locally within the functions are *local* data of the executing thread and cannot be accessed directly by other threads. Since the runtime stack of a thread is deleted after a thread is terminated, it is dangerous to pass a reference to a local variable in the runtime stack of a thread A to another thread B.

The data types, interface definitions and macros of Pthreads, are usually available via the header file `<pthread.h>`. This header file must therefore be included into a Pthreads program. The functions and data types of Pthreads are defined according to a naming convention. According to this convention, Pthreads functions are named in the form

```
pthread[_<object>]_<operation> ()
```

where `<operation>` describes the operation to be performed and the optional `<object>` describes the object to which this operation is applied. For example, `pthread_mutex_init()` is a function for the initialization of a mutex variable; thus, the `<object>` is `mutex` and the `<operation>` is `init`; we give a more detailed description later.

For functions which are involved in the manipulation of threads, the specification of `<object>` is omitted. For example, the function for the generation of a thread is `pthread_create()`. All Pthreads functions yield a return value 0, if they are executed without failure. In case of a failure, an error code from `<error.h>` will be returned. Thus, this header file should also be included in the program. Pthreads data types describe, similarly to MPI, opaque objects whose exact implementation is hidden from the programmer. Data types are named according to the syntax form

```
pthread_<object>_t
```

where `<object>` specifies the specific data object. For example, a mutex variable is described by the data type `pthread_mutex_t`. If `<object>` is omitted, the data type `pthread_t` for threads results. The following table contains important Pthread data types which will be described in more detail later.

Pthreads data types	meaning
<code>pthread_t</code>	Thread ID
<code>pthread_mutex_t</code>	mutex variable
<code>pthread_cond_t</code>	condition variable
<code>pthread_key_t</code>	access key
<code>pthread_attr_t</code>	thread attributes object
<code>pthread_mutexattr_t</code>	mutex attributes object
<code>pthread_condattr_t</code>	condition variable attributes object
<code>pthread_once_t</code>	<i>one time initialization</i> control context

For the execution of threads, we assume a two-step scheduling method according to Fig. 3.17 in Sect. 3, as this is the most general case. In this model, the programmer has to partition the program into a suitable number of user threads which can be executed concurrently with each other. The user threads are mapped by the library scheduler to system threads which are then brought to execution on the processors of the computing system by the scheduler of the operating system. The programmer cannot control the scheduler of the operating system and has only little influence on the library scheduler. Thus, the programmer cannot directly perform the mapping of the user-level threads to the processors of the computing system, e.g., by a scheduling at program level. This facilitates program development, but also prevents an efficient mapping directly by the programmer according to his specific needs. It should be noted that there are operating system specific extensions that allow thread execution to be bound to specific processors. But in most cases, the scheduling provided by the library and the operating system leads to good results and relieves the programmer from additional programming effort, thus providing more benefits than drawbacks.

In this section, we give an overview of the programming with Pthreads. Section 6.1.1 describes thread generation and management in Pthreads. Section 6.1.2 describes the lock mechanism for the synchronization of threads accessing shared variables. Sections 6.1.3 and 6.1.4 introduce Pthreads condition variables and an extended lock mechanism using condition variables. Sections 6.1.6 – 6.1.8 describe the use of the basic synchronization techniques in the context of more advanced synchronization patterns, like task pools, pipelining, and client-server coordination. Section 6.1.9 discusses additional mechanisms for the control of threads, including scheduling strategies. We describe in sect. 6.1.10 how the programmer can influence the scheduling controlled by the library. The phenomenon of *priority inversion* is then explained in Sect. 6.1.11 and finally thread-specific data are considered in Sect. 6.1.12. Only the most important mechanisms of the Pthreads standard are described; for a more detailed description, we refer to [25, 119, 132, 145, 161].

6.1.1 Creating and Merging Threads

When a Pthreads program is started, a single *main thread* is active, executing the `main()` function of the program. The main thread can generate more threads by calling the function

```
int pthread_create (pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_routine) (void *),
                   void *arg)
```

The first argument is a pointer to an object of type `pthread_t` which is also referred to as *thread identifier* (TID); this TID is generated by `pthread_create()` and can later be used by other Pthreads functions to identify the generated thread. The second argument is a pointer to a previously allocated and initialized attributes

object of type `pthread_attr_t`, defining the desired attributes of the generated thread. The argument value `NULL` causes the generation of a thread with default attributes. If different attribute values are desired, an attribute data structure has to be created and initialized before calling `pthread_create()`; this mechanism is described in more detail in Sect. 6.1.9. The third argument specifies the function `start_routine()` which will be executed by the generated thread. The specified function should expect a single argument of type `void *` and should have a return value of the same type. The fourth argument is a pointer to the argument value with which the thread function `start_routine()` will be executed.

To execute a thread function with more than one argument, all arguments must be put into a single data structure; the address of this data structure can then be specified as argument of the thread function. If several threads are started by a parent-thread using the same thread function but different argument values, *separate* data structures should be used for each of the threads to specify the arguments. This avoids situations where argument values are overwritten too early by the parent thread before they are read by the child threads or where different child threads manipulate the argument values in a common data structure concurrently.

A thread can determine its own thread identifier by calling the function

```
pthread_t pthread_self()
```

This function returns the thread ID of the calling thread. To compare the thread ID of two threads, the function

```
int pthread_equal (pthread_t t1, pthread_t t2)
```

can be used. This function returns the value 0 if t_1 and t_2 do not refer to the same thread. Otherwise, a nonzero value is returned. Since `pthread_t` is an opaque data structure, only `pthread_equal` should be used to compare thread IDs. The number of threads that can be generated by a process is typically limited by the system. The Pthreads standard determines that at least 64 threads can be generated by any process. But depending on the specific system used, this limit may be larger. For most systems, the maximum number of threads that can be started can be determined by calling

```
maxThreads = sysconf (_SC_THREAD_THREADS_MAX)
```

in the program. Knowing this limit, the program can avoid to start more than `maxThreads` threads. If the limit is reached, a call of the `pthread_create()` function returns the error value `EAGAIN`. A thread is terminated if its thread function terminates, e.g., by calling `return`. A thread can terminate itself explicitly by calling the function

```
void pthread_exit (void *valuep)
```

The argument `valuep` specifies the value that will be returned to another thread which waits for the termination of this thread using `pthread_join()`. When a thread terminates its thread function, the function `pthread_exit()` is called *implicitly*, and the return value of the thread function is used as argument of this implicit call of `pthread_exit()`. After the call to `pthread_exit()`, the calling thread is terminated, and its runtime stack is freed and can be used by other threads. Therefore, the return value of the thread should not be a pointer to a local variable of the thread function or another function called by the thread function. These local variables are stored on the runtime stack and may not exist any longer after the termination of the thread. Moreover, the memory space of local variables can be reused by other threads, and it can usually not be determined when the memory space is overwritten, thereby destroying the original value of the local variable. Instead of a local variable, a global variable or a variable that has been dynamically allocated should be used.

A thread can wait for the termination of another thread by calling the function

```
int pthread_join(pthread_t thread, void **valuep)
```

The argument `thread` specifies the thread ID of the thread for which the calling thread waits to be terminated. The argument `valuep` specifies a memory address where the return value of this thread should be stored. The thread calling `pthread_join()` is blocked until the specified thread has terminated. Thus, `pthread_join()` provides a possibility for the *synchronization* of threads. After the thread with TID `thread` has terminated, its return value is stored at the specified memory address. If several threads wait for the termination of the same thread, using `pthread_join()`, all waiting threads are blocked until the specified thread has terminated. But only one of the waiting threads successfully stores the return value. For all other waiting threads, the return value of `pthread_join()` is the error value `ESRCH`. The runtime system of the Pthreads library allocates for each thread an internal data structure to store information and data needed to control the execution of the thread. This internal data structure is preserved by the runtime system also after the termination of the thread to ensure that another thread can later successfully access the return value of the terminated thread using `pthread_join()`.

After the call to `pthread_join()`, the internal data structure of the terminated thread is released and can no longer be accessed. If there is no `pthread_join()` for a specific thread, its internal data structure is not released after its termination and occupies memory space until the complete process is terminated. This can be a problem for large programs with many thread creations and terminations without corresponding calls to `pthread_join()`. The preservation of the internal data structure of a thread after its termination can be avoided by calling the function

```
int pthread_detach(pthread_t thread)
```

This function notifies the runtime system that the internal data structure of the thread with TID `thread` can be detached as soon as the thread has terminated. A thread

may detach itself, and any thread may detach any other thread. After a thread has been set into a detached state, calling `pthread_join()` for this thread returns the error value `EINVAL`.

Example: We give a first example for a Pthreads program; Figure 6.1 shows a program fragment for the multiplication of two matrices, see also [145]. The matrices MA and MB to be multiplied have a fixed size of eight rows and eight columns. For each of the elements of the result matrix MC, a separate thread is created. The IDs of these threads are stored in the array `thread`. Each thread obtains a separate data structure of type `matrix_type_t` which contains pointers to the input matrices

```
#include <pthread.h>

typedef struct {
    int size, row, column;
    double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void *thread_mult (void *w) {
    matrix_type_t *work = (matrix_type_t *) w;
    int i, row = work->row, column = work->column;
    work->MC[row][column] = 0;
    for (i=0; i < work->size; i++)
        work->MC[row][column] += work->MA[row][i] * work->MB[i][column];
    return NULL;
}

int main() {
    int row, column, size = 8, i;
    double MA[8][8], MB[8][8], MC[8][8];
    matrix_type_t *work;
    pthread_t thread[8*8];
    for (row=0; row<size; row++)
        for (column=0; column<size; column++) {
            work = (matrix_type_t *) malloc (sizeof (matrix_type_t));
            work->size = size;
            work->row = row;
            work->column = column;
            work->MA = MA; work->MB = MB; work->MC = MC;
            pthread_create (&(thread[column + row*8]), NULL,
                          thread_mult, (void *) work);
        }

    for (i=0; i<size*size; i++)
        pthread_join (thread[i], NULL);
}
```

Fig. 6.1 Pthreads program for the multiplication of two matrices MA and MB. A separate thread is created for each element of the output matrix MC. A separate data structure `work` is provided for each of the threads created.

MA and MB, the output matrix MC, and the row and column position of the entry of MC to be computed by the corresponding thread. Each thread executes the same thread function `thread_mult()` which computes the scalar product of one row of MA and one column of MB. After creating a new thread for each of the 64 elements of MC to be computed, the main thread waits for the termination of each of these threads using `pthread_join()`. The program in Fig. 6.1 creates 64 threads which is exactly the limit defined by the Pthreads standard for the number of threads that must be supported by each implementation of the standard. Thus, the given program works correctly. But it is not scalable in the sense that it can be extended to the multiplication of matrices of any size. Since a separate thread is created for each element of the output matrix, it can be expected that the upper limit for the number of threads that can be generated will be reached even for matrices of moderate size. Therefore, the program should be re-written when using larger matrices such that a fixed number of threads is used and each thread computes a block of entries of the output matrix; the size of the blocks increases with the size of the matrices. \square

6.1.2 Thread Coordination with Pthreads

The threads of a process share a common address space. Therefore, they can concurrently access shared variables. To avoid race conditions, these concurrent accesses must be coordinated. To perform such coordinations, Pthreads provides *mutex variables* and *condition variables*.

6.1.2.1 Mutex variables

In Pthreads, a **mutex variable** denotes a data structure of the predefined opaque type `pthread_mutex_t`. Such a mutex variable can be used to ensure *mutual exclusion* when accessing common data, i.e., it can be ensured that only one thread at a time has exclusive access to a common data structure, all other threads have to wait. A mutex variable can be in one of two states: *locked* and *unlocked*. To ensure mutual exclusion when accessing a common data structure, a separate mutex variable is assigned to the data structure. All accessing threads must behave as follows: *Before* an access to the common data structure, the accessing thread locks the corresponding mutex variable using a specific Pthreads function. When this is successful, the thread is the *owner* of the mutex variable. *After* each access to the common data structure, the accessing thread unlocks the corresponding mutex variable. After the unlocking, it is no longer owner of the mutex variable and another thread can become owner and is allowed to access the data structure.

When a thread A tries to lock a mutex variable that is already owned by another thread B, thread A is blocked until thread B unlocks the mutex variable. The Pthreads runtime system ensures that only one thread at a time is the owner of a specific mutex variable. Thus, a conflicting manipulation of a common data structure is avoided if

each thread uses the described behavior. But if a thread accesses the data structure without locking the mutex variable before, mutual exclusion is no longer guaranteed.

The assignment of mutex variables to data structures is done implicitly by the programmer by protecting accesses to the data structure with locking and unlocking operations of a specific mutex variable. There is no explicit assignment of mutex variables to data structures. The programmer can improve the readability of Pthreads programs by grouping a common data structure and the protecting mutex variable into a new structure.

In Pthreads, mutex variables have the predefined type `pthread_mutex_t`. Like normal variables, they can be statically declared or dynamically generated. Before a mutex variable can be used, it must be initialized. For a mutex variable `mutex` that is allocated statically, this can be done by

```
mutex = PTHREAD_MUTEX_INITIALIZER
```

where `PTHREAD_MUTEX_INITIALIZER` is a predefined macro. For arbitrary mutex variables (statically allocated or dynamically generated), an initialization can be performed dynamically by calling the function

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      const pthread_mutexattr_t *attr)
```

For `attr = NULL`, a mutex variable with default properties results. The properties of mutex variables can be influenced by using different attribute values, see Sect. 6.1.9. If a mutex variable that has been initialized dynamically is no longer needed, it can be destroyed by calling the function

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

A mutex variable should only be destroyed if none of the threads are waiting for the mutex variable to become owner and if there is currently no owner of the mutex variable. A mutex variable that has been destroyed can later be re-used after a new initialization. A thread can lock a mutex variable `mutex` by calling the function

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

If another thread B is owner of the mutex variable `mutex` when a thread A issues the call of `pthread_mutex_lock()`, then thread A is blocked until thread B unlocks `mutex`. When several threads T_1, \dots, T_n try to lock a mutex variable which is owned by another thread, all threads T_1, \dots, T_n are blocked and are stored in a waiting queue for this mutex variable. When the owner releases the mutex variable, one of the blocked threads in the waiting queue is unblocked and becomes the new owner of the mutex variable. Which one of the waiting threads is unblocked may depend on their priorities and the scheduling strategies used, see Sect. 6.1.9 for more information. The order in which waiting threads become owner of a mutex variable is not defined in the Pthreads standard and may depend on the specific Pthreads library used.

A thread should not try to lock a mutex variable when it is already the owner. Depending on the specific runtime system, this may lead to an error return value EDEADLK or may even cause a self-deadlock. A thread which is owner of a mutex variable `mutex` can unlock `mutex` by calling the function

```
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```

After this call, `mutex` is in the state *unlocked*. If there is no other thread waiting for `mutex`, there is no owner of `mutex` after this call. If there are threads waiting for `mutex`, one of these threads is woken up and becomes the new owner of `mutex`. In some situations, it is useful that a thread can check without blocking whether a mutex variable is owned by another thread. This can be achieved by calling the function

```
int pthread_mutex_trylock (pthread_mutex_t *mutex)
```

If the specified mutex variable is currently not held by another thread, the calling thread becomes the owner of the mutex variable. This is the same behavior as for `pthread_mutex_lock()`. But different from `pthread_mutex_lock()`, the calling thread is *not blocked* if another thread already holds the mutex variable. Instead, the call returns with error return value EBUSY without blocking. The calling thread can then perform other computations and can later retry to lock the mutex variable. The calling thread can also repeatedly try to lock the mutex variable until it is successful (*spinlock*).

Example: Figure 6.2 shows a simple program fragment to illustrate the use of mutex variables to ensure mutual exclusion when concurrently accessing a common data structure, see also [145]. In the example, the common data structure is a linked list. The nodes of the list have type `node_t`. The complete list is protected by a single mutex variable. To indicate this, the pointer to the first element of the list (`first`) is combined with the mutex variable (`mutex`) into a data structure of type `list_t`. The linked list will be kept sorted according to increasing values of the node entry `index`. The function `list_insert()` inserts a new element into the list while keeping the sorting. Before the first call to `list_insert()`, the list must be initialized by calling `list_insert()`, e.g., in the main thread. This call also initializes the mutex variable. In `list_insert()`, the executing thread first locks the mutex variable of the list before performing the actual insertion. After the insertion, the mutex variable is released again using `pthread_mutex_unlock()`. This procedure ensures that it is not possible for different threads to insert new elements at the same time. Hence, the list operations are *sequentialized*. The function `list_insert()` is a *thread-safe* function, since a program can use this function without performing additional synchronization.

In general, a (library) function is thread-safe if it can be called by different threads concurrently, without performing additional operations to avoid race conditions. □

In Fig. 6.2, a single mutex variable is used to control the complete list. This results in a *coarse-grain* lock granularity. Only a single insert operation can happen at a time, independently from the length of the list. An alternative could be to partition the list into fixed-size areas and protect each area with a mutex variable, or even to

```

typedef struct node {
    int index;
    void *data;
    struct node *next;
} node_t;

typedef struct list {
    node_t *first;
    pthread_mutex_t mutex;
} list_t;

void list_init (list_t *listp)
{
    listp->first = NULL;
    pthread_mutex_init (&(listp->mutex), NULL);
}

void list_insert (int newindex, void *newdata, list_t *listp)
{
    node_t *current, *previous, *new;
    int found = FALSE;

    pthread_mutex_lock (&(listp->mutex));
    for (current = previous = listp->first; current != NULL;
        previous = current, current = current->next)
    {
        if (current->index == newindex) {
            found = TRUE; break;
        }
        else
            if (current->index > newindex) break;
    }
    if (!found) {
        new = (node_t *) malloc (sizeof (node_t));
        new->index = newindex;
        new->data = newdata;
        new->next = current;
        if (current == listp->first) listp->first = new;
        else previous->next = new;
    }
    pthread_mutex_unlock (&(listp->mutex));
}

```

Fig. 6.2 Pthread implementation of a linked list. The function `list_insert()` can be called by different threads concurrently to insert new elements into the list. In the form presented, `list_insert()` cannot be used as the start function of a thread, since the function has more than one argument. To be used as start function, the arguments of `list_insert()` have to be put into a new data structure which is then passed as argument. The original arguments could then be extracted from this data structure at the beginning of `list_insert()`.

protect each single element of the list with a separate mutex variable. In this case, the granularity would be **fine grained**, and several threads could access different parts of the list concurrently. But this also requires a substantial re-organization of the synchronization, possibly leading to a larger overhead.

6.1.2.2 Mutex variables and deadlocks

When multiple threads work with different data structures each of which is protected by a separate mutex variable, caution has to be taken to avoid deadlocks. A deadlock may occur, if the threads use a different order for locking the mutex variables. This can be seen for two threads T_1 and T_2 and two mutex variables ma and mb as follows:

- thread T_1 first locks ma and then mb ;
- thread T_2 first locks mb and then ma .

If T_1 is interrupted by the scheduler of the runtime system after locking ma such that T_2 is able to successfully lock mb , a deadlock occurs:

T_2 will be blocked when it is trying to lock ma , since ma is already locked by T_1 ; similarly, T_1 will be blocked when it is trying to lock mb after it has been woken up again, since mb has already been locked by T_2 . In effect, both threads are blocked forever and are mutually waiting for each other. The occurrence of deadlocks can be avoided by using a *fixed locking order* for all threads or by employing a *backoff strategy*.

When using a **fixed locking order**, each thread locks the critical mutex variables always in the same predefined order. Using this approach for the example above, thread T_2 must lock the two mutex variables ma and mb in the same order as T_1 , e.g., both threads must first lock ma and then mb . The deadlock described above cannot occur now, since T_2 cannot lock mb if ma has previously been locked by T_1 . To lock mb , T_2 must first lock ma . If ma has already been locked by T_1 , T_2 will be blocked when trying to lock ma , and hence cannot lock mb . The specific locking order used can in principle be arbitrarily selected, but to avoid deadlocks it is important that the order selected is used throughout the entire program. If this does not conform to the program structure, a backoff strategy should be used.

When using a **backoff strategy**, each participating thread can lock the mutex variables in its individual order, and it is not necessary to use the same predefined order for each thread. But a thread must back off when its attempt to lock a mutex variable fails. In this case, the thread must release all mutex variables that it has previously locked successfully. After the backoff, the thread starts the entire lock procedure from the beginning by trying to lock the first mutex variable again. To implement a backoff strategy, each thread uses `pthread_mutex_lock()` to lock its first mutex variable, and `pthread_mutex_trylock()` to lock the remaining mutex variables needed. If `pthread_mutex_trylock()` returns `EBUSY`, this means that this mutex variable is already locked by another thread. In this case, the calling thread releases all mutex variables that it has previously locked successfully using `pthread_mutex_unlock()`.

Example: Backoff strategy (see Fig. 6.3 and 6.4):

The use of a backoff strategy is demonstrated in Fig. 6.3 for two threads *f* and *b* which lock three mutex variables *m*[0], *m*[1], and *m*[2] in different orders, see [25]. The thread *f* (forward) locks the mutex variables in the order *m*[0], *m*[1], and *m*[2] by calling the function `lock_forward()`. The thread *b* (backward) locks the mutex variables in the opposite order *m*[2], *m*[1], and *m*[0] by calling the function `lock_backward()`, see Fig. 6.4. Both threads repeat the locking 10 times. The main program in Fig. 6.3 uses two control variables `backoff` and `yield_flag` which are read in as arguments. The control variable `backoff` determines whether a backoff strategy is used (value 1) or not (value 0). For `backoff = 1`, no deadlock occurs when running the program because of the backoff strategy. For `backoff = 0`, a deadlock occurs in most cases, in particular if *f* succeeds in locking *m*[0] and *b* succeeds in locking *m*[2].

But depending on the specific scheduling situation concerning *f* and *b*, no deadlock may occur even if no backoff strategy is used. This happens when both threads succeed in locking all three mutex variables, before the other thread is executed. To illustrate this dependence of deadlock occurrence from the specific scheduling situation, the example in Fig. 6.3 and 6.4 contains a mechanism to influence the scheduling of *f* and *b*. This mechanism is activated by using the control variable `yield_flag`. For `yield_flag = 0`, each thread tries to lock the mutex variables without interruption. This is the behavior described so far. For `yield_flag = 1`, each thread calls `sched_yield()` after having locked a mutex variable, thus transferring control to another thread with the same priority. Therefore, the other thread has a chance to lock a mutex variable. For `yield_flag = -1`, each thread calls `sleep(1)` after having locked a mutex variable, thus waiting for 1 s. In this time, the other thread can run and has a chance to lock another mutex variable. In both cases, a deadlock will likely occur if no backoff strategy is used.

Calling `pthread_exit()` in the main thread causes the termination of the main thread, but not of the entire process. Instead, using a normal `return` would terminate the entire process, including the threads *f* and *b*. □

Compared to a fixed locking order, the use of a backoff strategy typically leads to larger execution times, since threads have to back off when they do not succeed in locking a mutex variable. In this case, the locking of the mutex variables has to be started from the beginning.

But using a backoff strategy leads to an increased flexibility, since no fixed locking order has to be ensured. Both techniques can also be used in combination by using a fixed locking order in code regions where this is not a problem, and using a backoff strategy where the additional flexibility is beneficial.

6.1.3 Condition Variables

Mutex variables are typically used to ensure mutual exclusion when accessing global data structures concurrently. But mutex variables can also be used to wait for the

```

#include <pthread.h>
#include <sched.h>
#include <stdlib.h>
#include <stdio.h>
pthread_mutex_t m[3] = {
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_MUTEX_INITIALIZER };
int backoff = 1; // == 1: with backoff strategy
int yield_flag = 0; // > 0: use sched_yield, < 0: sleep
int main(int argc, char *argv[]) {
    pthread_t f, b;
    if (argc > 1) backoff = atoi(argv[1]);
    if (argc > 2) yield_flag = atoi(argv[2]);
    pthread_create(&f, NULL, lock_forward, NULL);
    pthread_create(&b, NULL, lock_backward, NULL);
    pthread_exit(NULL); // both threads continue execution
}

```

Fig. 6.3 Control program to illustrate the use of a backoff strategy.

occurrence of a specific condition which depends on the state of a global data structure and which has to be fulfilled before a certain operation can be applied. An example might be a shared buffer from which a consumer thread can remove entries only if the buffer is not empty. To apply this mechanism, the shared data structure is protected by one or several mutex variables, depending on the specific situation. To check whether the condition is fulfilled, the executing thread locks the mutex variable(s) and then evaluates the condition. If the condition is fulfilled, the intended operation can be performed. Otherwise, the mutex variable(s) are released again and the thread repeats this procedure again at a later time. This method has the drawback that the thread which is waiting for the condition to be fulfilled may have to repeat the evaluation of the condition quite often before the condition becomes true. This consumes execution time (*active waiting*), in particular because the mutex variable(s) have to be locked before the condition can be evaluated. To enable a more efficient method for waiting for a condition, Pthreads provides condition variables.

A **condition variable** is an opaque data structure which enables a thread to wait for the occurrence of an arbitrary condition without active waiting. Instead, a signaling mechanism is provided which blocks the executing thread during the waiting time, so that it does not consume CPU time. The waiting thread is woken up again as soon as the condition is fulfilled. To use this mechanism, the executing thread must define a condition variable and a mutex variable. The mutex variable is used to protect the evaluation of the specific condition which is waited for to be fulfilled. The use of the mutex variable is necessary, since the evaluation of a condition usually requires to access shared data which may be modified by other threads concurrently.

where `cond` is the address of the condition variable to be initialized and `attr` is the address of an attribute data structure for condition variables. Using `attr=NULL` leads to an initialization with the default attributes. For a condition variable `cond` that has been declared statically, the initialization can also be obtained by using the `PTHREAD_COND_INITIALIZER` initialization macro. This can also be done directly with the declaration

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER.
```

The initialization macro cannot be used for condition variables that have been generated dynamically using, e.g., `malloc()`. A condition variable `cond` that has been initialized with `pthread_cond_init()` can be destroyed by calling the function

```
int pthread_cond_destroy (pthread_cond_t *cond)
```

if it is no longer needed. In this case, the runtime system can free the information stored for this condition variable. Condition variables that have been initialized statically with the initialization macro, do not need to be destroyed.

Each condition variable must be uniquely associated with a specific mutex variable. All threads which wait for a condition variable at the same time must use the same associated mutex variable. It is not allowed that different threads associate different mutex variables with a condition variable at the same time. But a mutex variable can be associated to different condition variables. A condition variable should only be used for a single condition to avoid deadlocks or race conditions [25]. A thread must first lock the associated mutex variable `mutex` with `pthread_mutex_lock()` before it can wait for a specific condition to be fulfilled using the function

```
int pthread_cond_wait (pthread_cond_t *cond,  
                      pthread_mutex_t *mutex)
```

where `cond` is the condition variable used and `mutex` is the associated mutex variable. The condition is typically embedded into a surrounding control statement. A standard usage pattern is:

```
pthread_mutex_lock (&mutex);  
while (!condition())  
    pthread_cond_wait (&cond, &mutex);  
compute_something();  
pthread_mutex_unlock (&mutex);
```

The evaluation of the condition and the call of `pthread_cond_wait()` are protected by a mutex variable `mutex` to ensure that the condition does not change between the evaluation and the call of `pthread_cond_wait()`, e.g., because another thread changes the value of a variable that is used within the condition. Therefore, each thread must use this mutex variable `mutex` to protect the manipulation of each variable that is used within the condition. Two cases can occur for this usage pattern for condition variables:

- If the specified condition is fulfilled when executing the code segment from above, the function `pthread_cond_wait()` is **not** called. The executing thread releases the mutex variable and proceeds with the execution of the succeeding program part.
- If the specified condition is not fulfilled, `pthread_cond_wait()` is called. This call has the effect that the specified mutex variable `mutex` is implicitly released and that the executing thread is blocked, waiting for the condition variable until another thread sends a signal using `pthread_cond_signal()` to notify the blocked thread that the condition may now be fulfilled. When the blocked thread is woken up again in this way, it implicitly tries to lock the mutex variable `mutex` again. If this is owned by another thread, the woken-up thread is blocked again, now waiting for the mutex variable to be released. As soon as the thread becomes owner of the mutex variable `mutex`, it continues the execution of the program. In the context of the usage pattern from above, this results in a new evaluation of the condition because of the `while` loop.

In a Pthreads program, it should be ensured that a thread which is waiting for a condition variable is woken up only if the specified condition is fulfilled. Nevertheless, it is useful to evaluate the condition again after the wake up, because there are other threads working concurrently. One of these threads might become owner of the mutex variable before the woken-up thread. Thus, the woken-up thread is blocked again. During the blocking time, the owner of the mutex variable may modify common data, such that the condition is no longer fulfilled. Thus, from the perspective of the executing thread, the state of the condition may change in the time interval between being woken up and becoming owner of the associated mutex variable. Therefore, the thread must again evaluate the condition to be sure that it is still fulfilled. If the condition is fulfilled, it cannot change before the executing thread calls `pthread_mutex_unlock()` or `pthread_cond_wait()` for the same condition variable, since each thread must be owner of the associated mutex variable to modify a variable used in the evaluation of the condition.

Pthreads provides two functions to wake up (*signal*) a thread waiting on a condition variable:

```
int pthread_cond_signal(pthread_cond_t *cond)
int pthread_cond_broadcast(pthread_cond_t *cond).
```

A call of `pthread_cond_signal()` wakes up a *single* thread waiting on the condition variable `cond`. A call of this function has no effect, if there are no threads waiting for `cond`. If there are several threads waiting for `cond`, one of them is selected to be woken up. For the selection, the priorities of the waiting threads and the scheduling method used are taken into account. A call of `pthread_cond_broadcast()` wakes up *all* threads waiting on the condition variable `cond`. If several threads are woken up, only one of them can become owner of the associated mutex variable. All other threads that have been woken up are blocked on the mutex variable.

The functions `pthread_cond_signal()` and `pthread_cond_broadcast()` should only be called if the condition associated with `cond` is fulfilled.

Thus, before calling one of these functions, a thread should evaluate the condition. To do so safely, it must first lock the mutex variable associated with the condition variable to ensure a consistent evaluation of the condition. The actual call of `pthread_cond_signal()` or `pthread_cond_broadcast()` does not need to be protected by the mutex variable. Issuing a call without protection by the mutex variable has the drawback that another thread may become owner of the mutex variable when it has been released after the evaluation of the condition, but before the signaling call. In this situation, the new owner thread can modify shared variables such that the condition is no longer fulfilled. This does not lead to an error, since the woken-up thread will again evaluate the condition. The advantage of not protecting the call of `pthread_cond_signal()` or `pthread_cond_broadcast()` by the mutex variable is the chance that the mutex variable may not have an owner when the waiting thread is woken up. Thus, there is a chance that this thread becomes owner of the mutex variable without waiting. If mutex protection is used, the signaling thread is owner of the mutex variable when the signal arrives, so the woken-up thread must block on the mutex variable immediately after being woken up.

To wait for a condition, Pthreads also provides the function

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t *mutex,
                           const struct timespec *time)
```

The difference to `pthread_cond_wait()` is that the blocking on the condition variable `cond` is ended with return value `ETIMEDOUT` after the specified time interval `time` has elapsed. This maximum waiting time is specified using type

```
struct timespec {
    time_t tv_sec;
    long tv_nsec;
}
```

where `tv_sec` specifies the number of seconds and `tv_nsec` specifies the number of additional nanoseconds. The `time` parameter of `pthread_cond_timedwait()` specifies an absolute clock time rather than a time interval. A typical use may look as follows:

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
struct timespec time;
pthread_mutex_lock (&m);
time.tv_sec = time (NULL) + 10;
time.tv_nsec = 0;
while (!condition)
    if (pthread_cond_timedwait (&c, &m, &time) == ETIMEDOUT)
        timed_out_work();
pthread_mutex_unlock (&m);
```

In this example, the executing thread waits at most 10 s for the condition to be fulfilled. The function `time()` from `<time.h>` is used to define `time.tv_sec`. The call `time(NULL)` yields the absolute time in seconds elapsed since Jan 1, 1970. If no signal arrives after 10 s, the function `timed_out_work()` is called before the condition is evaluated again.

6.1.4 Extended Lock Mechanism

Condition variables can be used to implement more complex synchronization mechanisms that are not directly supported by Pthreads. In the following, we consider a *read/write lock* mechanism as an example for an extension of the standard lock mechanism provided by normal mutex variables. If we use a normal mutex variable to protect a shared data structure, only one thread at a time can access (read or write) the shared data structure. The following user-defined read/write locks extend this mechanism by allowing an arbitrary number of reading threads at a time. But only one thread at a time is allowed to write to the data structure. In the following, we describe a simple implementation of this extension, see also [145]. For more complex and more efficient implementations, we refer to [25, 119].

For the implementation of read/write locks, we define read/write lock variables (r/w lock variables) by combining a mutex variable and a condition variable as follows:

```
typedef struct rw_lock {
    int num_r, num_w;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} rw_lock_t;
```

Here, `num_r` specifies the current number of read permits, and `num_w` specifies the current number of write permits; `num_w` should have a maximum value of 1. The mutex variable `mutex` is used to protect the access to `num_r` and `num_w`. The condition variable `cond` coordinates the access to the read/write lock variable.

Figure 6.5 shows the functions that can be used to implement the read/write lock mechanism. The function `rw_lock_init()` initializes a read/write lock variable. The function `rw_lock_rlock()` requests a read permit to the common data structure. The read permit is granted only if there is no other thread that currently has a write permit. Otherwise, the calling thread is blocked until the write permit is returned. The function `rw_lock_wlock()` requests a write permit to the common data structure. The write permit is granted only if there is no other thread that currently has a read or write permit.

The function `rw_lock_runlock()` is used to return a read permit. This may cause that the number of threads with a read permit decreases to zero. In this case, a thread which is waiting for a write permit is woken up by `pthread_cond_signal()`. The function `rw_lock_wunlock()` is used to return a write permit.

```

int rw_lock_init (rw_lock_t *rwl) {
    rwl->num_r = rwl->num_w = 0;
    pthread_mutex_init (&(rwl->mutex), NULL);
    pthread_cond_init (&(rwl->cond), NULL);
    return 0;
}

int rw_lock_rlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    while (rwl->num_w > 0)
        pthread_cond_wait (&(rwl->cond), &(rwl->mutex));
    rwl->num_r ++;
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

int rw_lock_wlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    while ((rwl->num_w > 0) || (rwl->num_r > 0))
        pthread_cond_wait (&(rwl->cond), &(rwl->mutex));
    rwl->num_w ++;
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

int rw_lock_runlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    rwl->num_r --;
    if (rwl->num_r == 0) pthread_cond_signal (&(rwl->cond));
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

int rw_lock_wunlock (rw_lock_t *rwl) {
    pthread_mutex_lock (&(rwl->mutex));
    rwl->num_w --;
    pthread_cond_broadcast (&(rwl->cond));
    pthread_mutex_unlock (&(rwl->mutex));
    return 0;
}

```

Fig. 6.5 Function for the control of read/write lock variables.

Since only one thread with a write permit is allowed, there cannot be a thread with a write permit after this operation. Therefore, all threads waiting for a read or write permit can be woken up using `pthread_cond_broadcast()`.

The implementation sketched in Fig. 6.5 favors read requests over write requests: If a thread A has a read permit and a thread B waits for a write permit, then other threads will obtain a read permit without waiting, even if they put their read request

long after B has put its write request. Thread B will get a write permit only if there are no other threads requesting a read permit. Depending on the intended usage, it might also be useful to give write requests priority over read requests to keep a data structure up to date. An implementation for this is given in [25].

The read/write lock mechanism can be used for the implementation of a shared linked list, see Fig. 6.2, by replacing the mutex variable `mutex` by a read/write lock variable. In the `list_insert()` function, the list access will then be protected by `rw_lock_wlock()` and `rw_lock_wunlock()`. A function to search for a specific entry in the list could use `rw_lock_rlock()` and `tw_lock_runlock()`, since no entry of the list will be modified when searching.

6.1.5 One-Time Initialization

In some situations, it is useful to perform an operation only once, no matter how many threads are involved. This is useful for initialization operations or opening a file. If several threads are involved, it sometimes cannot be determined in advance which of the threads is first ready to perform an operation. A one-time initialization can be achieved using a boolean variable initialized to 0 and protected by a mutex variable. The first thread arriving at the critical operation sets the boolean variable to 1, protected by the mutex variable, and then performs the one time operation. If a thread arriving at the critical operation finds that the boolean variable has value 1, it does not perform the operation. Pthreads provides another solution for one time operations by using a control variable of the predefined type `pthread_once_t`. This control variable must be statically initialized using the initialization macro `PTHREAD_ONCE_INIT`:

```
pthread_once_t once_control PTHREAD_ONCE_INIT
```

The code to perform the one time operation must be put into a separate function without parameter. We call this function `once_routine()` in the following. The one time operation is then performed by calling the function

```
pthread_once(&pthread_once_t *once_control,  
            void (*once_routine)(void)).
```

This function can be called by several threads. If the execution of `once_routine()` has already been completed, then control is directly returned to the calling thread. If the execution of `once_routine()` has not yet been started, `once_routine()` is executed by the calling thread. If the execution of the function `once_routine()` has been started by another thread, but is not finished yet, then the thread executing `pthread_once()` waits until the other thread has finished its execution of `once_routine()`.

6.1.6 Implementation of a Task Pool

A thread program usually has to perform several operations or tasks. A simple structure results if each task is put into a separate function which is then called by a separate thread which executes exactly this function and then terminates. Depending on the granularity of the tasks, this may lead to the generation and termination of a large number of threads, causing a significant overhead. For many applications, a more efficient implementation can be obtained by using a **task pool** (also called *work crew*). The idea is to use a specific data structure (task pool) to store the tasks that are ready for execution. For task execution, a fixed number of threads is used which are generated by the main thread at program start and exist until the program terminates. The threads access the task pool to retrieve tasks for execution. During the execution of a task, new tasks may be generated which are then inserted into the task pool. The execution of the parallel program is terminated, if the task pool is empty and each thread has finished the execution of its task.

The advantage of this execution scheme is that a fixed number of threads is used, no matter how many tasks are generated. This keeps the overhead for thread management small, independent of the number of tasks. Moreover, tasks can be generated dynamically, thus enabling the realization of adaptive and irregular applications. In the following, we describe a simple implementation of a task pool, see also [145]. More advanced implementations are described in [25, 119].

Figure 6.6 presents the data structure that can be used for the task pool, and a function for the initialization of the task pool. The data type `work_t` represents a single task. It contains a reference `routine` to the function containing the code of the task and the argument `arg` of this function. The tasks are organized as a linked list, and `next` is a pointer to the next task element. The data type `tpool_t` represents the actual task pool. It contains pointers `head` and `tail` to the first and last element of the task list, respectively. The entry `num_threads` specifies the number of threads used for execution of the tasks. The array `threads` contains the reference to the thread IDs of these threads. The entries `max_size` and `current_size` specify the maximum and current number of tasks contained in the task pool.

The mutex variable `lock` is used to ensure mutual exclusion when accessing the task pool. If a thread attempts to retrieve a task from an empty task pool, it is blocked on the condition variable `not_empty`. If a thread inserts a task into an empty task pool, it wakes up a thread that is blocked on `not_empty`. If a thread attempts to insert a task into a full task pool, it is blocked on the condition variable `not_full`. If a thread retrieves a task from a full task pool, it wakes up a thread that is blocked on `not_full`.

The function `tpool_init()` in Fig. 6.6 initializes the task pool by allocating the data structure and initializing it with the argument values provided. Moreover, the threads used for the execution of the tasks are generated and their IDs are stored in `tpl->threads[i]` for $i=0, \dots, \text{num_threads}-1$. Each of these threads uses the function `tpool_thread()` as start function, see Fig. 6.7. This function has one argument specifying the task pool data structure to be used. Task

```

typedef struct work {
    void (*routine)();
    void *arg;
    struct work *next;
} work_t;

typedef struct tpool {
    int num_threads, max_size, current_size;
    pthread_t *threads;
    work_t *head, *tail;
    pthread_mutex_t lock;
    pthread_cond_t not_empty, not_full;
} tpool_t;

tpool_t *tpool_init (int num_threads, int max_size) {
    int i;
    tpool_t *tpl;

    tpl = (tpool_t *) malloc (sizeof (tpool_t));
    tpl->num_threads = num_threads;
    tpl->max_size = max_size;
    tpl->current_size = 0;
    tpl->head = tpl->tail = NULL;

    pthread_mutex_init (&(tpl->lock), NULL);
    pthread_cond_init (&(tpl->not_empty), NULL);
    pthread_cond_init (&(tpl->not_full), NULL);
    tpl->threads = (pthread_t *)malloc(sizeof(pthread_t)*num_threads);
    for (i=0; i<num_threads; i++)
        pthread_create (&(tpl->threads[i]), NULL,
                        tpool_thread, (void *) tpl);

    return tpl;
}

```

Fig. 6.6 Implementation of a task pool (part 1): The data structure `work_t` represents a task to be executed. The task pool data structure `tpool_t` contains a list of tasks with `head` pointing to the first element and `tail` pointing to the last element, as well as a set of threads `threads` to execute the tasks. The function `tpool_init()` is used to initialize a task pool data structure `tpl`.

execution is performed in an infinite loop. In each iteration of the loop, a task is retrieved from the head of the task list. If the task list is empty, the executing thread is blocked on the condition variable `not_empty` as described above. Otherwise, a task `wl` is retrieved from the list. If the task pool has been full before the retrieval, all threads blocked on `not_full`, waiting to insert a task, are woken up using `pthread_cond_broadcast()`. The access to the task pool structure is protected by the mutex variable `tpl->lock`. The retrieved task `wl` is executed by calling the stored task function `wl->routine()` using the stored argument `wl->arg`. The execution of the retrieved task `wl` may lead to the generation of new

```

void *tpool_thread (void *arg) {
    work_t *wl;
    tpool_t *tpl = (tpool_t *) arg;

    for( ; ; ) {
        pthread_mutex_lock (&(tpl->lock));
        while (tpl->current_size == 0)
            pthread_cond_wait (&(tpl->not_empty), &(tpl->lock));
        wl = tpl->head;
        tpl->current_size --;
        if (tpl->current_size == 0)
            tpl->head = tpl->tail = NULL;
        else tpl->head = wl->next;
        if (tpl->current_size == tpl->max_size - 1)
            pthread_cond_broadcast (&(tpl->not_full));
        pthread_mutex_unlock (&(tpl->lock));
        (*(wl->routine))(wl->arg);
        free(wl);
    }
}

void tpool_insert (tpool_t *tpl, void (*routine)(), void *arg) {
    work_t *wl;

    pthread_mutex_lock (&(tpl->lock));
    while (tpl->current_size == tpl->max_size)
        pthread_cond_wait (&(tpl->not_full), &(tpl->lock));
    wl = (work_t *) malloc (sizeof (work_t));
    wl->routine = routine;
    wl->arg = arg;
    wl->next = NULL;
    if (tpl->current_size == 0) {
        tpl->tail = tpl->head = wl;
        pthread_cond_signal (&(tpl->not_empty));
    }
    else {
        tpl->tail->next = wl;
        tpl->tail = wl;
    }
    tpl->current_size ++;
    pthread_mutex_unlock (&(tpl->lock));
}

```

Fig. 6.7 Implementation of a task pool (part 2): The function `tpool_thread()` is used to extract and execute tasks. The function `tpool_insert()` is used to insert tasks into the task pool.

tasks which are then inserted into the task pool using `tpool_insert()` by the executing thread.

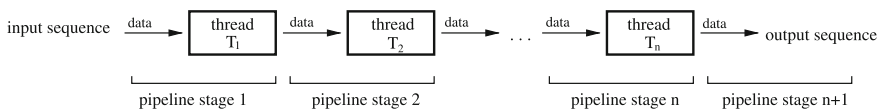
The function `tpool_insert()` is used to insert tasks into the task pool. If the task pool is full when calling this function, the executing thread is blocked on the

condition variable `not_full`. If the task pool is not full, a new task structure is generated and filled, and is inserted at the end of the task list. If the task pool has been empty before the insertion, one of the threads blocked on the condition variable `not_empty` is woken up using `pthread_cond_signal()`. The access to the task pool `tpl` is protected by the mutex variable `tpl->lock`.

The described implementation is especially suited for a master-slave model. A master thread uses `tpool_init()` to generate a number of slave threads each of which executes the function `tpool_thread()`. The tasks to be executed are defined according to the specific requirements of the application problem and are inserted in the task pool by the master thread using `tpool_insert()`. Tasks can also be inserted by the slave threads when the execution of a task leads to the generation of new tasks. After the execution of all tasks is completed, the master thread terminates the slave threads. To do so, the master thread wakes up all threads blocked on the condition variables `not_full` and `not_empty` and terminates them. Threads that are just executing a task are terminated as soon as they have finished the execution of this task.

6.1.7 Parallelism by Pipelining

In the pipelining model, a stream of data items is processed one after another by a sequence of threads T_1, \dots, T_n where each thread T_i performs a specific operation on each element of the data stream and passes the element onto the next thread T_{i+1} :



This results in an input/output relation between the threads: thread T_i receives the output of thread T_{i+1} as input and produces data elements for thread T_{i+1} , $1 < i < n$. Thread T_1 reads the sequence of input elements, thread T_n produces the sequence of output elements. After a start-up phase with n steps, all threads can work in parallel and can be executed by different processors in parallel. The pipeline model requires some coordination between the cooperating threads: thread T_i can start the computation of its corresponding stage only if the predecessor thread T_{i-1} has provided the input data element. Moreover, thread T_i can forward its output element to the successor thread T_{i+1} , if T_{i+1} has finished its computation of the previous data item and is ready to receive a new data element.

The coordination of the threads of the pipeline stages can be organized with the help of condition variables. This will be demonstrated in the following for a simple example in which a sequence of integer values is incremented step by step in each pipeline stage, see also [25]. Thus, in each pipeline stage, the same computation is performed. But the coordination mechanism can also be applied if each pipeline stage performs a different computation.

For each stage of the pipeline, a data structure of type `stage_t` is used, see Fig. 6.8. This data structure contains a mutex variable `m` for synchronizing the access to the stage and two condition variables `avail` and `ready` for synchronizing the threads of neighboring stages. The condition variable `avail` is used to notify a thread that a data element is available to be processed by its pipeline stage. Thus, the thread can start the computation. A thread is blocked on the condition variable `avail` if no data element from the predecessor stage is available. The condition variable `ready` is used to notify the thread of the preceding pipeline stage that it can forward its output to the next pipeline stage. The thread of the preceding pipeline stage is blocked on this condition variable if it cannot directly forward its output data element to the next stage. The entry `data_ready` in the data structure for a stage is used to record whether a data element is currently available (value 1) for this pipeline stage or not (value 0). The entry `data` contains the actual data element to be processed. For the simple example discussed here, this is a single integer value, but this could be any data element for more complex applications. The entry `thread` is the TID of the thread used for this stage, and `next` is a reference to the next pipeline stage.

The entire pipeline is represented by the data structure `pipe_t` containing a mutex variable `m` and two pointers `head` and `tail` to the first and the last stages of the pipeline, respectively. The last stage of the pipeline is used to store the final result of the computation performed for each data element. There is no computation performed in this last stage, and there is no corresponding thread associated to this stage.

The function `pipe_send()`, shown in Fig. 6.9 is used to send a data element to a stage of the pipeline. This function is used to send a data element to the first stage of the pipeline, and it is also used to pass a data element to the next stage of the

```
typedef struct stage { // pipeline stage
    pthread_mutex_t m;
    pthread_cond_t avail; // input data available for this stage?
    pthread_cond_t ready; // stage ready to receive new data?
    int data_ready; // != 0, if other data is currently computed
    long data; // data element
    pthread_t thread; // Thread ID
    struct stage *next;
} stage_t;
typedef struct pipe { // pipeline
    pthread_mutex_t m;
    stage_t *head, *tail; // first/last stage of the pipeline
    int stages; // number of stages of the pipeline
    int active; // number of active data elements in the pipeline
} pipe_t;
const N_STAGES = 10;
```

Fig. 6.8 Implementation of a pipeline (part 1): Data structures for the implementation of a pipeline model in Pthreads.

pipeline after the computation of a stage has been completed. The stage receiving the data element is identified by the parameter `nstage`. Before inserting the data element, the mutex variable `m` of the receiving stage is locked to ensure that only one thread at a time is accessing the stage. A data element can be written into the receiving stage only if the computation of the previous data element in this stage has been finished. This is indicated by the condition `data_ready=0`. If this is not the case, the sending thread is blocked on the condition variable `ready` of the receiving stage. If the receiving stage is ready to receive the data element, the sending thread writes the element into the stage and wakes up the thread of the receiving stage if it is blocked on the condition variable `avail`.

Each of the threads participating in the pipeline computation executes the function `pipe_stage()`, see Fig. 6.9. The same function can be used for each stage for our example, since each stage performs the same computations. The function receives a pointer to its corresponding pipeline stage as an argument. A thread executing

```
int pipe_send(stage_t *nstage, long data) {
    //parameter: target stage and data element to be processed
    pthread_mutex_lock(&nstage->m);
    {
        while (nstage->data_ready)
            pthread_cond_wait(&nstage->ready, &nstage->m);
        nstage->data = data;
        nstage->data_ready = 1;
        pthread_cond_signal(&nstage->avail);
    }
    pthread_mutex_unlock(&nstage->m);
}

void *pipe_stage(void *arg) {
    stage_t *stage = (stage_t*)arg;
    long result_data;
    pthread_mutex_lock(&stage->m);
    {
        for ( ; ; ) {
            while (!stage->data_ready) { // wait for data
                pthread_cond_wait(&stage->avail, &stage->m);
            }
            // process data element and forward to next stage :
            result_data = stage->data + 1; // compute result data element
            pipe_send(stage->next, result_data);
            stage->data_ready = 0; // processing finished
            pthread_cond_signal(&stage->ready);
        }
    }
    pthread_mutex_unlock(&stage->m); //this unlock will never be reached
}
```

Fig. 6.9 Implementation of a pipeline (part 2): Functions to forward data elements to a pipeline stage and thread functions for the pipeline stages.

the function performs an infinite loop waiting for the arrival of data elements to be processed. The thread blocks on the condition variable `avail` if there is currently no data element available. If a data element is available, the thread performs its computation (increment by 1) and sends the result to the next pipeline stage `stage->next` using `pipe_send()`. Then it sends a notification to the thread associated with the next stage, which may be blocked on the condition variable `ready`. The notified thread can then continue its computation.

Thus, the synchronization of two neighboring threads is performed by using the condition variables `avail` and `ready` of the corresponding pipeline stages. The entry `data_ready` is used for the condition and determines which of the two threads is blocked and woken up. The entry of a stage is set to 0 if the stage is ready to receive a new data element to be processed by the associated thread. The entry `data_ready` of the next stage is set to 1 by the associated thread of the preceding stage if a new

```
int pipe_create(pipe_t *pipe, int stages) { // creation of the pipeline
    int pi;
    stage_t **link = &pipe->head;
    stage_t *new_stage, *stage;
    pthread_mutex_init(&pipe->m, NULL);
    pipe->stages = stages;
    pipe->active = 0;
    // create stages+1 stages, last stage is for the result
    for (pi = 0; pi <= stages; pi++) {
        new_stage = (stage_t *)malloc(sizeof(stage_t));
        pthread_mutex_init(&new_stage->m, NULL);
        pthread_cond_init(&new_stage->avail, NULL);
        pthread_cond_init(&new_stage->ready, NULL);
        new_stage->data_ready = 0;
        *link = new_stage; // make list link
        link = &new_stage->next;
    }
    *link = (stage_t *) NULL;
    pipe->tail = new_stage;
    // create a thread for each stage except the last one
    for (stage = pipe_head; stage->next != NULL; stage = stage->next) {
        pthread_create(&stage->thread, NULL, pipe_stage, (void*)stage);
    }
}

int pipe_start(pipe_t *pipe, long v) { // start pipeline computation
    pthread_mutex_lock(&pipe->m);
    {
        pipe->active++;
    }
    pthread_mutex_unlock(&pipe->m);
    pipe_send(pipe->head, v);
}
```

Fig. 6.10 Implementation of a pipeline (part 3): Pthreads functions to generate and start a pipeline computation.

data element has been put into the next stage and is ready to be processed. In the simple example given here, the same computations are performed in each stage, i.e., all corresponding threads execute the same function `pipe_stage()`. For more complex scenarios, it is also possible that the different threads execute different functions, thus performing different computations in each pipeline stage.

The generation of a pipeline with a given number of stages can be achieved by calling the function `pipe_create()`, see Fig. 6.10. This function generates and initializes the data structures for the representation of the different stages. An additional stage is generated to hold the final result of the pipeline computation, i.e., the total number of stages is `stages+1`. For each stage except for the last additional stage, a thread is created. Each of these threads executes the function `pipe_stage()`.

The function `pipe_start()` is used to transfer a data element to the first stage of the pipeline, see Fig. 6.10. The actual transfer of the data element is done by calling the function `pipe_send()`. The thread executing `pipe_start()` does not wait for the result of the pipeline computation. Instead, `pipe_start()` returns control immediately. Thus, the pipeline works asynchronously to the thread which transfers data elements to the pipeline for computation. The synchronization between this thread and the thread of the first pipeline stage is performed within the function `pipe_send()`.

The function `pipe_result()` is used to take a result value out of the last stage of the pipeline, see Fig. 6.11. The entry `active` in the pipeline data structure `pipe_t` is used to count the number of data elements that are currently stored in the different pipeline stages. For `pipe->active = 0`, no data element is stored in the pipeline. In this case, `pipe_result()` immediately returns without providing a data element. For `pipe->active > 0`, `pipe_result()` is blocked on the condition variable `avail` of the last pipeline stage until a data element arrives at this stage. This happens if the thread associated with the next to the last stage uses `pipe_send()` to transfer a processed data element to the last pipeline stage, see Fig. 6.9. By doing so, this thread wakes up a thread that is blocked on the condition variable `avail` of the last stage, if there is a thread waiting. If so, the woken-up thread is the one which tries to take a result value out of the last stage using `pipe_result()`.

The main program of the pipeline example is given in Fig. 6.11. It first uses `pipe_create()` to generate a pipeline with a given number of stages. Then it reads from `stdin` lines with numbers, which are the data elements to be processed. Each such data element is forwarded to the first stage of the pipeline using `pipe_start()`. Doing so, the executing main thread may be blocked on the condition variable `ready` of the first stage until the stage is ready to receive the data element. An input line with a single character '=' causes the main thread to call `pipe_result()` to take a result element out of the last stage, if present.

Figure 6.12 illustrates the synchronization between neighboring pipeline threads as well as between the main thread and the threads of the first or the next to last stage for a pipeline with three stages and two pipeline threads T_1 and T_2 . The figure shows the relevant entries of the data structure `stage_t` for each stage. The order

```

int pipe_result(pipe_t *pipe, long *result) {
    stage_t *tail = pipe->tail;
    int empty = 0;
    pthread_mutex_lock(&pipe->m);
    {
        if (pipe->active <= 0) empty = 1; // empty pipeline
        else pipe->active--; // remove a data element
    }
    pthread_mutex_unlock(&pipe->m);
    if (empty) return 0;
    pthread_mutex_lock(&tail->m);
    {
        while (!tail->data_ready) // wait for data element
            pthread_cond_wait(&tail->avail, &tail->m);
        *result = tail->data;
        tail->data_ready = 0;
        pthread_cond_signal(&tail->ready);
    }
    pthread_mutex_unlock(&tail->m);
    return 1;
}

int main(int argc, char *argv[]) {
    pipe_t pipe;
    long value, result;
    char line[128];
    pipe_create(&pipe, N_STAGES);
    // terminate program for input error or EOF
    while (fgets(line, sizeof(line), stdin)) {
        if (*line == '\0')
            continue; // ignore empty input lines
        if (!strcmp(line, "=")) {
            if (pipe_result(&pipe, &result))
                printf("%ld\n", result);
            else printf("ERROR: Pipe empty\n");
        }
        else {
            if (sscanf(line, "%ld", &value) < 1)
                printf("ERROR: Not an int\n");
            else pipe_start(&pipe, value);
        }
    }
    return 0;
}

```

Fig. 6.11 Implementation of a pipeline (part 4): Main program and Pthreads function to remove a result element from the pipeline.

of the access and synchronization operations performed by the pipeline threads is determined by the statements in `pipe_stage()` and is illustrated by circled numbers. The access and synchronization operations of the main thread result from the statements in `pipe_start()` and `pipe_result()`.


```

#define REQ_READ 1
#define REQ_WRITE 2
#define REQ_QUIT 3
#define PROMPT_SIZE 32
#define TEXT_SIZE 128
typedef struct request {
    struct _request_t *next; // linked list
    int op;
    int synchronous; // 1 iff client waits for server
    int done_flag;
    pthread_cond_t done;
    char prompt[PROMPT_SIZE], text[TEXT_SIZE];
} request_t;
typedef struct tty_server { // data structure for server context
    request_t *first, *last;
    int running; // != 0, if server is running
    pthread_mutex_t m;
    pthread_cond_t request;
} tty_server_t;
#define TTY_SERVER_INITIALIZER { NULL, NULL, 0,
    PTHREAD_MUTEX_INITIALIZER,
    PTHREAD_COND_INITIALIZER }
tty_server_t tty_server = TTY_SERVER_INITIALIZER;
int client_threads;
pthread_mutex_t client_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t client_done = PTHREAD_COND_INITIALIZER;
pthread_t server_thread;

```

Fig. 6.13 Implementation of a client-server system (part 1): Data structure for the implementation of a client-server model with Pthreads.

(value 1) or not (value 0). The condition variable `done` is used for the synchronization between client and server, i.e., the client thread is blocked on `done` to wait until the server has finished the execution of the request. The entries `prompt` and `text` are used to store a prompt to be output or a text read in by the server, respectively. The data structure `tty_server_t` is used to store the requests sent to a server. The requests are stored in a FIFO (*first-in, first-out*) queue which can be accessed by `first` and `last`. The server thread is blocked on the condition variable `request` if the request queue is empty. The entry `running` indicates whether the corresponding server is running (value 1) or not (value 0). The program described in the following works with a single server thread, but can in principle be extended to an arbitrary number of servers.

The server thread executes the function `tty_server_routine()`, see Fig. 6.14. The server is blocked on the condition variable `request` as long as there are no requests to be processed. If there are requests, the server removes the first request from the queue and executes the operation (`REQ_READ`, `REQ_WRITE`, or `REQ_QUIT`) specified in the request. For the `REQ_READ` operation, the `prompt` specified with the request is output and a line is read in and stored into the `text`

```

void *tty_server_routine(void *arg) {
    static pthread_mutex_t prompt_mutex = PTHREAD_MUTEX_INITIALIZER;
    request_t *request;
    int op, len;
    for (;;) {
        pthread_mutex_lock(&tty_server.m);
        {
            while (tty_server.first == NULL)
                pthread_cond_wait(&tty_server.request, &tty_server.m);
            request = tty_server.first;
            tty_server.first = request->next;
            if (tty_server.first == NULL)
                tty_server.last = NULL;
        }
        pthread_mutex_unlock(&tty_server.m);
        switch (request->op) {
            case REQ_READ:
                puts(request->prompt);
                if (fgets(request->text, TEXT_SIZE, stdin) == NULL)
                    request->text[0] = '\0';
                len = strlen(request->text);
                if (len > 0 && request->text[len - 1] == '\n')
                    request->text[len - 1] = '\0';
                break;
            case REQ_WRITE:
                puts(request->text); break;
            default: // auch REQ_QUIT
                break;
        }
        op = request->op;
        if (request->synchronous) {
            pthread_mutex_lock(&tty_server.m);
            request->done_flag = 1;
            pthread_cond_signal(&request->done);
            pthread_mutex_unlock(&tty_server.m);
        }
        else free (request);
        if (op == REQ_QUIT) break;
    }
    return NULL;
}

```

Fig. 6.14 Implementation of a client-server system (part 2): Server thread to process client requests.

entry of the request structure. For a REQ_WRITE operation, the line stored in the text entry is written to stdout. The operation REQ_QUIT causes the server to finish its execution. If an issuing client waits for the termination of a request (entry synchronous), it is blocked on the condition variable done in the corresponding request structure. In this case, the server thread wakes up the blocked client


```

void tty_server_request(int op, int sync, char *prompt, char *string) {
    request_t *request;
    pthread_mutex_lock(&tty_server.m);
    {
        if (!tty_server.running) {
            pthread_create(&server_thread, NULL, tty_server_routine, NULL);
            tty_server.running = 1;
        }
        request = (request_t *) malloc(sizeof(request_t));
        request->op = op;
        request->synchronous = sync;
        request->next = NULL;
        if (sync) {
            request->done_flag = 0;
            pthread_cond_init(&request->done, NULL);
        }
        if (prompt != NULL) {
            strncpy(request->prompt, prompt, PROMPT_SIZE);
            request->prompt[PROMPT_SIZE - 1] = '\0';
        }
        else request->prompt[0] = '\0';
        if (op == REQ_WRITE && string != NULL) {
            strncpy(request->text, string, TEXT_SIZE);
            request->text[TEXT_SIZE - 1] = '\0';
        }
        else request->text[0] = '\0';
        if (tty_server.first == NULL)
            tty_server.first = tty_server.last = request;
        else {
            tty_server.last->next = request;
            tty_server.last = request;
        }
        pthread_cond_signal(&tty_server.request);
        if (sync) {
            while (!request->done_flag)
                pthread_cond_wait(&request->done, &tty_server.m);
            if (op == REQ_READ)
                strcpy(string, request->text);
            pthread_cond_destroy(&request->done);
            free(request);
        }
    }
    pthread_mutex_unlock(&tty_server.m);
}

```

Fig. 6.15 Implementation of a client-server system (part 3): Forwarding of a request to the server thread.

thread using `pthread_cond_signal()` after the request has been processed. For asynchronous requests, the server thread is responsible to free the request data structure.

```

void *client_routine(void *arg) {
    int my_nr = *(int*)arg, loops;
    char prompt[PROMPT_SIZE], string[TEXT_SIZE];
    char format[TEXT_SIZE + 64];
    sprintf(prompt, "Client %d>", my_nr);
    for (;;) {
        tty_server_request(REQ_READ, 1, prompt, string);
        // synchronized input
        if (string[0] == '\0') break; // program exit
        for (loops = 0; loops < 4; loops++) {
            sprintf(format, "(%d # %d)%s", my_nr, loops, string);
            tty_server_request(REQ_WRITE, 0, NULL, format);
            sleep(1);
        }
    }
    pthread_mutex_lock(&client_mutex);
    client_threads--;
    if (client_threads == 0) pthread_cond_signal(&client_done);
    pthread_mutex_unlock(&client_mutex);
    return NULL;
}

#define N_THREADS 4
int main(int argc, char *argv[]) {
    pthread_t thread;
    int i;
    int args[N_THREADS];
    client_threads = N_THREADS;
    pthread_mutex_lock(&client_mutex);
    {
        for (i = 0; i < N_THREADS; i++) {
            args[i] = i;
            pthread_create(&thread, NULL, client_routine, &args[i]);
        }
        while (client_threads > 0)
            pthread_cond_wait(&client_done, &client_mutex);
    }
    pthread_mutex_unlock(&client_mutex);
    printf("All clients done\n");
    tty_server_request(REQ_QUIT, 1, NULL, NULL);
    return 0;
}

```

Fig. 6.16 Implementation of a client-server system (part 4): Client thread and main thread.

The client threads use the function `tty_server_request()` to forward a request to the server, see Fig. 6.15. If the server thread is not running yet, it will be started in `tty_server_request()`. The function allocates a request structure of type `request_t` and initializes it according to the requested operation. The request structure is then inserted into the request queue of the server. If the server is blocked waiting for requests to arrive, it is woken up using `pthread_cond_signal()`. If

the client wants to wait for the termination of the request by the server, it is blocked on the condition variable `done` in the request structure, waiting for the server to wake it up again. The client threads execute the function `client_routine()`, see Fig. 6.16. Each client sends read and write requests to the server using the function `tty_server_request()` until the user terminates the client thread by specifying an empty line as input. When the last client thread has been terminated, the main thread which is blocked on the condition variable `client_done` is woken up again. The main thread generates the client threads and then waits until all client threads have been terminated. The server thread is not started by the main thread, but by the client thread which sends the first request to the server using `tty_server_routine()`. After all client threads are terminated, the server thread is terminated by the main thread by sending a `REQ_QUIT` request.

6.1.9 Thread Attributes and Cancellation

Threads are created using `pthread_create()`. In the previous sections, we have specified `NULL` as the second argument, thus leading to the generation of threads with default characteristics. These characteristics can be changed with the help of attribute objects. To do so, an attribute object has to be allocated and initialized before using the attribute object as parameter of `pthread_create()`. An attribute object for threads has type `pthread_attr_t`. Before an attribute object can be used, it first must be initialized by calling the function

```
int pthread_attr_init(pthread_attr_t *attr).
```

This leads to an initialization with the default attributes, corresponding to the default characteristics. By changing an attribute value, the characteristics can be changed. Pthreads provides attributes to influence the return value of threads, setting the size and address of the runtime stack, or the cancellation behavior of the thread. For each attribute, Pthreads defines functions to get and set the current attribute value. But Pthreads implementations are not required to support the modification of all attributes. In the following, the most important aspects are described.

6.1.9.1 Return value

An important property of a thread is its behavior concerning thread termination. This is captured by the attribute `detachstate`. This attribute can be influenced by all Pthreads libraries. By default, the runtime system assumes that the return value of a thread T_1 may be used by another thread after the termination of T_1 . Therefore, the internal data structure maintained for a thread will be kept by the runtime system after the termination of a thread until another thread retrieves the return value using `pthread_join()`, see Sect 6.1.1. Thus, a thread may bind resources even after its

termination. This can be avoided if the programmer knows in advance that the return value of a thread will not be needed. If so, the thread can be generated such that its resources are immediately returned to the runtime system after its termination. This can be achieved by changing the `detachstate` attribute. The following two functions are provided to get or set this attribute value:

```
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                               int *detachstate)
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                               int detachstate).
```

The attribute value `detachstate=PTHREAD_CREATE_JOINABLE` means that the return value of the thread is kept until it is joined by another thread. The attribute value `detachstate=PTHREAD_CREATE_DETACHED` means that the thread resources are freed immediately after thread termination.

6.1.9.2 Stack characteristics

The different threads of a process have a shared program and data memory and a shared heap, but each thread has its own runtime stack. For most Pthreads libraries, the size and address of the local stack of a thread can be changed, but it is not required that a Pthreads library supports this option. The local stack of a thread is used to store local variables of functions whose execution has not yet been terminated. The size required for the local stack is influenced by the size of the local variables and the nesting depth of function calls to be executed. This size may be large for recursive functions. If the default stack size is too small, it can be increased by changing the corresponding attribute value. The Pthreads library that is used supports this if the macro

`_POSIX_THREAD_ATTR_STACKSIZE`

is defined in `<unistd.h>`. This can be checked by

```
#ifdef _POSIX_THREAD_ATTR_STACKSIZE    or
if (sysconf (_SC_THREAD_ATTR_STACKSIZE) == -1)
```

in the program. If it is supported, the current stack size stored in an attribute object can be retrieved or set by calling the functions

```
int pthread_attr_getstacksize(const pthread_attr_t *attr,
                             size_t *stacksize)
int pthread_attr_setstacksize(pthread_attr_t *attr,
                             size_t stacksize).
```

Here, `size_t` is a datatype defined in `<unistd.h>` which is usually implemented as unsigned `int`. The parameter `stacksize` is the size of the stack in bytes. The value of `stacksize` should be at least `PTHREAD_STACK_MIN` which is

predefined by Pthreads as the minimum stack size required by a thread. Moreover, if the macro

```
_POSIX_THREAD_ATTR_STACKADDR
```

is defined in `<unistd.h>`, the address of the local stack of a thread can also be influenced. The following two functions

```
int pthread_attr_getstackaddr(const pthread_attr_t*attr,
                             size_t **stackaddr)
int pthread_attr_setstackaddr(pthread_attr_t*attr,
                             size_t *stackaddr)
```

are provided to get or set the current stack address stored in an attribute object. The modification of stack-related attributes should be used with caution, since such modification can result in nonportable programs. Moreover, the option is not supported by all Pthreads libraries.

After the modification of specific attribute values in an attribute object, a thread with the chosen characteristics can be generated by specifying the attribute object as second parameter of `pthread_create()`. The characteristics of the new thread are defined by the attribute values stored in the attribute object at the time at which `pthread_create()` is called. These characteristics cannot be changed at a later time by changing attribute values in the attribute object.

6.1.9.3 Thread Cancellation

In some situations, it is useful to stop the execution of a thread from outside, e.g., if the result of the operation performed is no longer needed. An example could be an application where several threads are used to search in a data structure for a specific entry. As soon as the entry is found by one of the threads, all other threads can stop execution to save execution time. This can be reached by sending a cancellation request to these threads.

In Pthreads, a thread can send a cancellation request to another thread by calling the function

```
int pthread_cancel (pthread_t thread)
```

where `thread` is the thread ID of the thread to be terminated. A call of this function does not necessarily lead to an immediate termination of the specified target thread. The exact behavior depends on the cancellation type of this thread. In any case, control immediately returns to the calling thread, i.e., the thread issuing the cancellation request does not wait for the canceled thread to be terminated. By default, the cancellation type of the thread is **deferred**. This means that the thread can only be canceled at specific **cancellation points** in the program. After the arrival of a cancellation request, thread execution continues until the next cancellation point is reached. The Pthreads standard defines obligatory and optional cancellation points. Obligatory cancellation points typically include all functions at which the executing thread may be

blocked for a substantial amount of time. Examples are `pthread_cond_wait()`, `pthread_cond_timedwait()`, `open()`, `read()`, `wait()` or `pthread_join()`, see [25] for a complete list. Optional cancelation points include many file and I/O operations. The programmer can insert additional cancelation points into the program by calling the function

```
void pthread_testcancel();
```

When calling this function, the executing thread checks whether a cancelation request has been sent to it. If so, the thread is terminated. If not, the function has no effect. Similarly, at predefined cancelation points the executing thread also checks for cancelation requests. A thread can set its cancelation type by calling the function

```
int pthread_setcancelstate (int state, int *oldstate).
```

A call with `state = PTHREAD_CANCEL_DISABLE` disables the cancelability of the calling thread. The previous cancelation type is stored in `*oldstate`. If the cancelability of a thread is disabled, it does not check for cancelation requests when reaching a cancelation point or when calling `pthread_testcancel()`, i.e., the thread cannot be canceled from outside. The cancelability of a thread can be enabled again by calling `pthread_setcancelstate()` with the parameter value `state = PTHREAD_CANCEL_ENABLE`.

By default, the cancelation type of a thread is deferred. This can be changed to **asynchronous cancelation** by calling the function

```
int pthread_setcanceltype (int type, int *oldtype)
```

with `type=PTHREAD_CANCEL_ASYNCHRONOUS`. This means that this thread can be canceled not only at cancelation points. Instead, the thread is terminated immediately after the cancelation request arrives, even if the thread is just performing computations within a critical section. This may lead to inconsistent states causing errors for other threads. Therefore, asynchronous cancelation may be harmful and should be avoided. Calling `pthread_setcanceltype()` with `type = PTHREAD_CANCEL_DEFERRED` sets a thread to the usual deferred cancelation type.

6.1.9.4 Cleanup Stack

In some situations, a thread may need to restore some state when it is canceled. For example, a thread may have to release a mutex variable when it is the owner before being canceled. To support such state restorations, a cleanup stack is associated with each thread, containing function calls to be executed just before thread cancelation. These function calls can be used to establish a consistent state at thread cancelation, e.g., by unlocking mutex variables that have previously been locked. This is necessary if there is a cancelation point between acquiring and releasing a mutex variable. If a cancelation happens at such a cancelation point without releasing the mutex variable, another thread might wait forever to become the owner. To avoid such situations, the

cleanup stack can be used: when acquiring the mutex variable, a function call (cleanup handler) to release it is put onto the cleanup stack. This function call is executed when the thread is canceled. A cleanup handler is put onto the cleanup stack by calling the function

```
void pthread_cleanup_push (void (*routine) (void*), void*arg)
```

where *routine* is a pointer to the function used as cleanup handler and *arg* specifies the corresponding argument values. The cleanup handlers on the cleanup stack are organized in LIFO (*last-in, first-out*) order, i.e., the handlers are executed in the opposite order of their placement, beginning with the most recently added handler. The handlers on the cleanup stack are automatically executed when the corresponding thread is canceled or when it exits by calling `pthread_exit()`. A cleanup handler can be removed from the cleanup stack by calling the function

```
void pthread_cleanup_pop (int execute).
```

This call removes the most recently added handler from the cleanup stack. For `execute≠0`, this handler will be executed when it is removed. For `execute=0`, this handler will be removed without execution. To produce portable programs, corresponding calls of `pthread_cleanup_push()` and `pthread_cleanup_pop()` should be organized in pairs within the same function.

Example: To illustrate the use of cleanup handlers, we consider the implementation of a semaphore mechanism in the following. A (*counting*) *semaphore* is a data type with a counter which can have nonnegative integer values and which can be modified by two operations: a *signal* operation increments the counter and wakes up a thread which is blocked on the semaphore, if there is such a thread; a *wait* operation blocks the executing thread until the counter has a value > 0 , and then decrements the counter. Counting semaphores can be used for the management of limited resources. In this case, the counter is initialized with the number of available resources. *Binary semaphores*, on the other hand, can only have value 0 or 1. They can be used to ensure mutual exclusion when executing critical sections.

Figure 6.17 illustrates the use of cleanup handlers to implement a semaphore mechanism based on condition variables, see also [161]. A semaphore is represented by the data type `sema_t`. The function `AcquireSemaphore()` waits until the counter has values > 0 , before decrementing the counter. The function `ReleaseSemaphore()` increments the counter and then wakes up a waiting thread using `pthread_cond_signal()`. The access to the semaphore data structure is protected by a mutex variable in both cases, to avoid inconsistent states by concurrent accesses. At the beginning, both functions call `pthread_mutex_lock()` to lock the mutex variable. At the end, the call `pthread_cleanup_pop(1)` leads to the execution of `pthread_mutex_unlock()`, thus releasing the mutex variable again. If a thread is blocked in `AcquireSemaphore()` when executing the function `pthread_cond_wait(&(ps->cond), &(ps->mutex))` it implicitly releases the mutex variable `ps->mutex`. When the thread is woken up again, it first tries to become owner of this mutex variable again. Since `pthread_`

```

typedef struct Sema {
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    int count;
} sema_t;

void CleanupHandler (void *arg)
{ pthread_mutex_unlock ((pthread_mutex_t *) arg);}

void AquireSemaphore (sema_t *ps)
{
    pthread_mutex_lock (&(ps->mutex));
    pthread_cleanup_push (CleanupHandler, &(ps->mutex));
    while (ps->count == 0)
        pthread_cond_wait (&(ps->cond), &(ps->mutex));
    --ps->count;
    pthread_cleanup_pop (1);
}

void ReleaseSemaphore (sema_t *ps)
{
    pthread_mutex_lock (&(ps->mutex));
    pthread_cleanup_push (CleanupHandler, &(ps->mutex));
    ++ps->count;
    pthread_cond_signal (&(ps->cond));
    pthread_cleanup_pop (1);
}

```

Fig. 6.17 Use of a cleanup handler for the implementation of a semaphore mechanism. The function `AquireSemaphore()` implements the access to the semaphore. The call of `pthread_cond_wait()` ensures that the access is performed not before the value `count` of the semaphore is larger than zero. The function `ReleaseSemaphore()` implements the release of the semaphore.

`cond_wait()` is a cancelation point, a thread might be canceled while waiting for the condition variable `ps->cond`. In this case, the thread first becomes the owner of the mutex variable before termination. Therefore, a cleanup handler is used to release the mutex variable again. This is obtained by the function `Cleanup_Handler()` in Fig. 6.17. □

6.1.9.5 Producer–Consumer threads

The semaphore mechanism from Fig. 6.17 can be used for the synchronization between producer and consumer threads, see Fig. 6.18. A producer thread inserts entries into a buffer of fixed length. A consumer thread removes entries from the buffer for further processing. A producer can insert entries only if the buffer is not full. A consumer can remove entries only if the buffer is not empty. To control this,

two semaphores `full` and `empty` are used. The semaphore `full` counts the number of occupied entries in the buffer. It is initialized with 0 at program start. The semaphore `empty` counts the number of free entries in the buffer. It is initialized with the buffer capacity. In the example, the buffer is implemented as an array of length 100, storing entries of type `ENTRY`. The corresponding data structure `buffer` also contains the two semaphores `full` and `empty`.

As long as the buffer is not full, a producer thread produces entries and inserts them into the shared buffer using `produce_item()`. For each insert operation, `empty` is decremented by using `AcquireSemaphore()` and `full` is incremented by using `ReleaseSemaphore()`. If the buffer is full, a producer thread will be blocked when calling `AcquireSemaphore()` for `empty`. As long as the buffer is not empty, a consumer thread removes entries from the buffer and processes them using `consume_item()`. For each remove operation, `full` is decremented using `AcquireSemaphore()` and `empty` is incremented using `ReleaseSemaphore()`. If the buffer is empty, a consumer thread will be blocked when calling the function `AcquireSemaphore()` for `full`. The internal buffer management is hidden in the functions `produce_item()` and `consume_item()`.

After a producer thread has inserted an entry into the buffer, it wakes up a consumer thread which is waiting for the semaphore `full` by calling the function `ReleaseSemaphore(&buffer.full)`, if there is such a waiting consumer. After a consumer has removed an entry from the buffer, it wakes up a producer which is waiting for `empty` by calling `ReleaseSemaphore(&buffer.empty)`, if there is such a waiting producer. The program in Fig. 6.18 uses one producer and one consumer thread, but it can easily be generalized to an arbitrary number of producer and consumer threads.

6.1.10 Thread Scheduling with Pthreads

The user threads defined by the programmer for each process are mapped to kernel threads by the library scheduler. The kernel threads are then brought to execution on the available processors by the scheduler of the operating system. For many Pthreads libraries, the programmer can influence the mapping of user threads to kernel threads using **scheduling attributes**. The Pthreads standard specifies a scheduling interface for this, but this is not necessarily supported by all Pthreads libraries. A specific Pthreads library supports the scheduling programming interface, if the macro `POSIX_THREAD_PRIORITY_SCHEDULING` is defined in `<unistd.h>`. This can also be checked dynamically in the program using `sysconf()` with parameter `_SC_THREAD_PRIORITY_SCHEDULING`. If the scheduling programming interface is supported and shall be used, the header file `<sched.h>` must be included into the program.

```

struct linebuf {
    ENTRY line[100];
    sema_t full, empty;
} buffer;

void *Producer(void *arg)
{
    while (1) {
        AquireSemaphore (&buffer.empty);
        produce_item();
        ReleaseSemaphore (&buffer.full);
    }
}

void *Consumer(void *arg)
{
    while (1) {
        AquireSemaphore (&buffer.full);
        consume_item();
        ReleaseSemaphore (&buffer.empty);
    }
}

void CreateSemaphore (sema_t *ps, int count)
{
    ps->count = count;
    pthread_mutex_init (&ps->mutex, NULL);
    pthread_cond_init (&ps->cond, NULL);
}

int main()
{
    pthread_t threadID[2];
    int i;
    void *status;

    CreateSemaphore (&buffer.empty, 100);
    CreateSemaphore (&buffer.full, 0);

    pthread_create (&threadID[0], NULL, Consumer, NULL);
    pthread_create (&threadID[1], NULL, Producer, NULL);

    for (i=0; i<2; i++)
        pthread_join (threadID[i], &status);
}

```

Fig. 6.18 Implementation of producer–consumer threads using the semaphore operations from Fig. 6.17.

Scheduling attributes are stored in data structures of type `struct sched_param` which must be provided by the Pthreads library if the scheduling interface is supported. This type must at least have the entry

```
int sched_priority;
```

The scheduling attributes can be used to assign scheduling priorities to threads and to define scheduling policies and scheduling scopes. This can be set when a thread is created, but it can also be changed dynamically during thread execution.

6.1.10.1 Explicit setting of scheduling attributes

In the following, we first describe how scheduling attributes can be set explicitly at thread creation.

The **scheduling priority** of a thread determines how privileged the library scheduler treats the execution of a thread compared to other threads. The priority of a thread is defined by an integer value which is stored in the `sched_priority` entry of the `sched_param` data structure and which must lie between a minimum and maximum value. These minimum and maximum values allowed for a specific scheduling policy can be determined by calling the functions

```
int sched_get_priority_min (int policy)
int sched_get_priority_max (int policy),
```

where `policy` specifies the scheduling policy. The minimum or maximum priority values are given as return value of these functions. The library scheduler maintains for each priority value a separate queue of threads with this priority that are ready for execution. When looking for a new thread to be executed, the library scheduler accesses the thread queue with the highest priority that is not empty. If this queue contains several threads, one of them is selected for execution according to the scheduling policy. If there are always enough executable threads available at each point in program execution, it can happen that threads of low priority are not executed for quite a long time. The two functions

```
int pthread_attr_getschedparam(const pthread_attr_t *attr,
                               struct sched_param *param)
int pthread_attr_setschedparam(pthread_attr_t *attr,
                               const struct sched_param *param)
```

can be used to extract or set the priority value of an attribute data structure `attr`. To set the priority value, the entry `param->sched_priority` must be set to the chosen priority value before calling `pthread_attr_setschedparam()`.

The scheduling policy of a thread determines how threads of the same priority are executed and share the available resources. In particular, the scheduling policy determines how long a thread is executed if it is selected by the library scheduler for execution. Pthreads supports three different scheduling policies:

- **SCHED_FIFO** (*first-in, first-out*): The executable threads of the same priority are stored in a FIFO queue. A new thread to be executed is selected from the beginning of the thread queue with the highest priority. The selected thread is executed until it either exits or blocks, or until a thread with a higher priority becomes ready for execution. In the latter case, the currently executed thread with lower priority is interrupted and is stored at the *beginning* of the corresponding thread queue. Then, the thread of higher priority starts execution. If a thread that has been blocked, e.g., waiting on a condition variable, becomes ready for execution again, it is stored at the *end* of the thread queue of its priority. If the priority of a thread is dynamically changed, it is stored at the *end* of the thread queue with the new priority.
- **SCHED_RR** (*round-robin*): The thread management is similar as for the policy SCHED_FIFO. The difference is that each thread is allowed to run for only a fixed amount of time, given by a predefined timeslice interval. After the interval has elapsed, and another thread of the same priority is ready for execution, the running thread will be interrupted and put at the *end* of the corresponding thread queue. The timeslice intervals are defined by the library scheduler. All threads of the same process use the same timeslice interval. The length of a timeslice interval of a process can be queried with the function

```
int sched_rr_get_interval(pid_t pid, struct timespec *quantum)
```

where `pid` is the process ID of the process. For `pid=0`, the information for that process is returned to which the calling thread belongs. The data structure of type `timespec` is defined as

```
struct timespec { time_t tv_sec; long tv_nsec; }.
```

- **SCHED_OTHER**: Pthreads allows an additional scheduling policy the behavior of which is not specified by the standard, but completely depends on the specific Pthreads library used. This allows the adaptation of the scheduling to a specific operating system. Often, a scheduling strategy is used which adapts the priorities of the threads to their I/O behavior, such that interactive threads get a higher priority as compute-intensive threads. This scheduling policy is often used as default for newly created threads.

The scheduling policy used for a thread is set when the thread is created. If the programmer wants to use a scheduling policy other than the default he can achieve this by creating an attribute data structure with the appropriate values and providing this data structure as argument for `pthread_create()`. The two functions

```
int pthread_attr_getschedpolicy(const pthread_attr_t *attr,
                               int *schedpolicy)
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                               int schedpolicy)
```

can be used to extract or set the scheduling policy of an attribute data structure `attr`. On some Unix systems, setting the scheduling policy may require superuser rights.

The **contention scope** of a thread determines which other threads are taken into consideration for the scheduling of a thread. Two options are provided: The thread may compete for processor resources with the threads of the corresponding process (*process contention scope*) or with the threads of all processes on the system (*system contention scope*). Two functions can be used to extract or set the contention scope of an attribute data structure `attr`:

```
int pthread_attr_getscope(const pthread_attr_t *attr,
                        int *contentionscope)
int pthread_attr_setscope(pthread_attr_t *attr,
                        int contentionscope)
```

The parameter value `contentionscope=PTHREAD_SCOPE_PROCESS` corresponds to a process contention scope, whereas a system contention scope can be obtained by the parameter value `contentionscope=PTHREAD_SCOPE_SYSTEM`. Typically, using a process contention scope leads to better performance than a system contention scope, since the library scheduler can switch between the threads of a process without calling the operating system, whereas switching between threads of different processes usually requires a call of the operating system, and this is usually relatively expensive [25]. A Pthreads library only needs to support one of the two contention scopes. If a call of `pthread_attr_setscope()` tries to set a contention scope that is not supported by the specific Pthreads library, the error value `ENOTSUP` is returned.

6.1.10.2 Implicit setting of scheduling attributes

Some application codes create a lot of threads for specific tasks. To avoid setting the scheduling attributes before each thread creation, Pthreads supports the inheritance of scheduling information from the creating thread. The two functions

```
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                int *inheritsched)
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                int inheritsched)
```

can be used to extract or set the inheritance status of an attribute data structure `attr`. Here, `inheritsched=PTHREAD_INHERIT_SCHED` means that a thread creation with this attribute structure generates a thread with the scheduling attributes of the creating thread, ignoring the scheduling attributes in the attribute structure. The parameter value `inheritsched=PTHREAD_EXPLICIT_SCHED` disables the inheritance, i.e., the scheduling attributes of the created thread must be set explicitly if they should be different from the default setting. The Pthreads standard does not specify a default value for the inheritance status. Therefore, if a specific behavior is required, the inheritance status must be set explicitly.

6.1.10.3 Dynamic setting of scheduling attributes

The priority of a thread and the scheduling policy used can also be changed dynamically during the execution of a thread. The two functions

```
int pthread_getschedparam(pthread_t thread, int *policy,
                          struct sched_param *param)
int pthread_setschedparam(pthread_t thread, int policy,
                          const struct sched_param *param)
```

can be used to dynamically extract or set the scheduling attributes of a thread with TID `thread`. The parameter `policy` defines the scheduling policy, `param` contains the priority value.

Figure 6.19 illustrates how the scheduling attributes can be set explicitly before the creation of a thread. In the example, `SCHED_RR` is used as scheduling policy. Moreover, a medium priority value is used for the thread with ID `thread_id`. The

```
#include <unistd.h>
#include <pthread.h>
#include <sched.h>

void *thread_routine (void *arg) {return NULL;}

int main()
{
    pthread_t thread_id;
    pthread_attr_t attr;
    struct sched_param param;
    int policy, min_prio, max_prio;

    pthread_attr_init (&attr);
    if (sysconf (_SC_THREAD_PRIORITY_SCHEDULING) != -1) {
        pthread_attr_getschedpolicy (&attr, &policy);
        pthread_attr_getschedparam (&attr, &param);
        printf ("Default: Policy %d, Priority %d \n", policy,
                param.sched_priority);
        pthread_attr_setschedpolicy (&attr, SCHED_RR);
        min_prio = sched_get_priority_min (SCHED_RR);
        max_prio = sched_get_priority_max (SCHED_RR);
        param.sched_priority = (min_prio + max_prio)/2;
        pthread_attr_setschedparam (&attr, &param);
        pthread_attr_setinheritsched (&attr, PTHREAD_EXPLICIT_SCHED);
    }
    pthread_create (&thread_id, &attr, thread_routine, NULL);
    pthread_join (thread_id, NULL);
    return 0;
}
```

Fig. 6.19 Use of scheduling attributes to define the scheduling behavior of a generated thread.

inheritance status is set to `PTHREAD_EXPLICIT_SCHED` to transfer the scheduling attributes from `attr` to the newly created thread `thread_id`.

6.1.11 Priority Inversion

When scheduling several threads with different priorities, it can happen with an unsuitable order of synchronization operations that a thread of lower priority prevents a thread of higher priority from being executed. This phenomenon is called **priority inversion**, indicating that a thread of lower priority is running, although a thread of higher priority is ready for execution. This phenomenon is illustrated in the following example, see also [145].

Example: We consider the execution of three threads *A*, *B*, *C* with high, medium and low priority, respectively, on a single processor competing for a mutex variable *m*. The threads perform at program points t_1, \dots, t_6 the following actions, see Fig. 6.20 for an illustration. After the start of the program at time t_1 , thread *C* of low priority is started at time t_2 . At time t_3 , thread *C* calls `pthread_mutex_lock(m)` to lock *m*. Since *m* has not been locked before, *C* becomes owner of *m* and continues execution. At time t_4 , thread *A* of high priority is started. Since *A* has a higher priority than *C*, *C* is blocked and *A* is executed. The mutex variable *m* is still locked by *C*. At time t_5 , thread *A* tries to lock *m* using `pthread_mutex_lock(m)`. Since *m* has already been locked by *C*, *A* blocks on *m*. The execution of *C* resumes. At time t_6 , thread *B* of medium priority is started. Since *B* has a higher priority than *C*, *C* is blocked and *B* is executed. *C* is still owner of *m*. If *B* does not try to lock *m*, it may be executed for quite some time, even if there is a thread *A* of higher priority. But *A* cannot be executed, since it waits for the release of *m* by *C*. But *C* cannot release *m*, since *C* is not executed. Thus, the processor is continuously executing *B* and not *A*, although *A* has a higher priority than *B*. \square

Pthreads provides two mechanisms to avoid priority inversion: priority ceiling and priority inheritance. Both mechanisms are optional, i.e., they are not necessarily supported by each Pthreads library. We describe both mechanisms in the following.

point in time	event	thread A high priority	thread B medium priority	thread C low priority	mutex variable <i>m</i>
t_1	start	/	/	/	free
t_2	start C	/	/	running	free
t_3	C locks <i>m</i>	/	/	running	locked by C
t_4	start A	running	/	ready for execution	locked by C
t_5	A locks <i>m</i>	blocked	/	running	locked by C
t_6	start B	blocked	running	ready for execution	locked by C

Fig. 6.20 Illustration of a priority inversion.


```
int pthread_mutexattr_setprioceiling (pthread_mutexattr_t
                                     *attr, int prio)
```

can be used to extract or set the priority ceiling value stored in the attribute structure `attr`. The ceiling value specified in `prio` must be a valid priority value. After a mutex attributed data structure `attr` has been initialized and possibly modified, it can be used for the initialization of a mutex variable with the specified properties, using the function

```
pthread_mutex_init (pthread_mutex_t *m, pthread_mutexattr_t
                  *attr)
```

see also Section [6.1.2](#).

6.1.11.2 Priority inheritance

When using the priority inheritance protocol, the priority of a thread which is the owner of a mutex variable is automatically raised, if a thread with a higher priority tries to lock the mutex variable and is therefore blocked on the mutex variable. In this situation, the priority of the owner thread is raised to the priority of the blocked thread. Thus, the owner of a mutex variable always has the maximum priority of all threads waiting for the mutex variable. Therefore, the owner thread cannot be interrupted by one of the waiting threads, and priority inversion cannot occur. When the owner thread releases the mutex variable again, its priority is decreased again to the original priority value.

The priority inheritance protocol can be used if the macro

```
_POSIX_THREAD_PRIO_INHERIT
```

is defined in `<unistd.h>`. If supported, priority inheritance can be activated by calling the function `pthread_mutexattr_setprotocol()` with parameter value `prio = PTHREAD_PRIO_INHERIT` as described above. Compared to priority ceiling, priority inheritance has the advantage that no fixed priority ceiling value has to be specified in the program. Priority inversion is avoided also for threads with unknown priority values. But the implementation of priority inheritance in the Pthreads library is more complicated and expensive, and therefore usually leads to a larger overhead than priority ceiling.

6.1.12 Thread-specific Data

The threads of a process share a common address space. Thus, global and dynamically allocated variables can be accessed by each thread of a process. For each thread, a private stack is maintained for the organization of function calls performed by the thread. The local variables of a function are stored in the private stack of the calling

thread. Thus, they can only be accessed by this thread, if this thread does not expose the address of a local variable to another thread. But the lifetime of local variables is only the lifetime of the corresponding function activation. Thus, local variables do not provide a persistent thread-local storage (TLS). To use the value of a local variable throughout the lifetime of a thread, it has to be declared in the start function of the thread and passed as parameter to all functions called by this thread. But depending on the application, this would be quite tedious and would artificially increase the number of parameters. Pthreads supports the use of thread-specific data with an additional mechanism.

To generate thread-specific data, Pthreads provides the concept of *keys* that are maintained in a process-global way. After the creation of a key it can be accessed by each thread of the corresponding process. Each thread can associate thread-specific data to a key. If two threads associate different data to the same key, each of the two threads gets only its own data when accessing the key. The Pthreads library handles the management and storage of the keys and their associated data.

In Pthreads, keys are represented by the predefined data type `pthread_key_t`. A key is generated by calling the function

```
int pthread_key_create(pthread_key_t *key,
                      void (*destructor)(void*))
```

The generated key is returned in the parameter `key`. If the key is used by several threads, the address of a global variable or a dynamically allocated variable must be passed as `key`. The function `pthread_key_create()` should only be called once for each `pthread_key_t` variable. This can be ensured with the `pthread_once()` mechanism, see Sect. 6.1.4. The optional parameter `destructor` can be used to assign a deallocation function to the key to clean up the data stored when the thread terminates. If no deallocation is required, `NULL` should be specified. A key can be deleted by calling the function

```
int pthread_key_delete(pthread_key_t key)
```

After the creation of a key, its associated data are initialized with `NULL`. Each thread can associate new data value to the key by calling the function

```
int pthread_setspecific(pthread_key_t key, void *value)
```

Typically, the address of a *dynamically* generated data object will be passed as `value`. Passing the address of a local variable should be avoided, since this address is no longer valid after the corresponding function has been terminated. The data associated with a key can be retrieved by calling the function

```
void *pthread_getspecific(pthread_key_t key)
```

The calling thread always obtains the data value that it has previously associated with the key using `pthread_setspecific()`. When no data have been associated yet, `NULL` is returned. `NULL` is also returned, if another thread has associated data with the key, but not the calling thread. When a thread uses the function

`pthread_setspecific()` to associate new data to a key, data that have previously been associated to this key by this thread will be overwritten and is lost.

An alternative to thread-specific data is the use of TLS which is provided since the C99 standard. This mechanism allows the declaration of variables with the storage class keyword `__thread` with the effect that each thread gets a separate instance of the variable. The instance is deleted as soon as the thread terminates. The `__thread` storage class keyword can be applied to global variables and static variables. It cannot be applied to block-scoped automatic or nonstatic variables.

6.2 Java Threads

Java supports the development of multi-threaded programs at the language level. Java provides language constructs for the synchronized execution of program parts and supports the creation and management of threads by predefined classes. In this chapter, we demonstrate the use of Java threads for the development of parallel programs for a shared address space. We assume that the reader knows the principles of object-oriented programming as well as the standard language elements of Java. We concentrate on the mechanisms for the development of multi-threaded programs and describe the most important elements. We refer to [147, 128] for a more detailed description. For a detailed description of Java, we refer to [57].

6.2.1 Thread Generation in Java

Each Java program in execution consists of at least one thread of execution, the *main thread*. This is the thread which executes the `main()` method of the class which has been given to the Java Virtual Machine (JVM) as start argument.

More user threads can be created explicitly by the main thread or other user threads that have been started earlier. The creation of threads is supported by the predefined class `Thread` from the standard package `java.lang`. This class is used for the representation of threads and provides methods for the creation and management of threads.

The interface `Runnable` from `java.lang` is used to represent the program code executed by a thread; this code is provided by a `run()` method and is executed asynchronously by a separate thread. There are two possibilities to arrange this: inheriting from the `Thread` class or using the interface `Runnable`.

6.2.1.1 Inheriting from the `Thread` class

One possibility to obtain a new thread is to define a new class `NewClass` which inherits from the predefined class `Thread` and which defines a method `run()`

```

import java.lang.Thread;
public class NewClass extends Thread { // inheritance
    public void run() {
        // overwriting method run() of class Thread
        System.out.println("hello from new thread");
    }
    public static void main (String args[]) {
        NewClass nc = new NewClass();
        nc.start();
    }
}

```

Fig. 6.21 Thread creation by overwriting the `run()` method of the `Thread` class.

containing the statements to be executed by the new thread. The `run()` method defined in `NewClass` overwrites the predefined `run()` method from `Thread`.

The `Thread` class also contains a method `start()` which creates a new thread executing the given `run()` method.

The newly created thread is executed asynchronously to the generating thread. After the execution of `start()` and the creation of the new thread, the control will be immediately returned to the generating thread. Thus, the generating thread resumes execution usually before the new thread has terminated, i.e., the generating thread and the new thread are executed concurrently to each other.

The new thread is terminated when the execution of the `run()` method has been finished. This mechanism for thread creation is illustrated in Fig. 6.21 with a class `NewClass` whose `main()` method generates an object of `NewClass` and whose `run()` method is activated by calling the `start()` method of the newly created object. Thus, thread creation can be performed in two steps:

- (1) definition of a class `NewClass` which inherits from `Thread` and which defines a `run()` method for the new thread;
- (2) instantiation of an object `nc` of class `NewClass` and activation of `nc.start()`.

The creation method just described requires that the class `NewClass` inherits from `Thread`. Since Java does not support multiple inheritance, this method has the drawback that `NewClass` cannot be embedded into another inheritance hierarchy. Java provides interfaces to obtain a similar mechanism as multiple inheritance. For thread creation, the interface `Runnable` is used.

6.2.1.2 Using the interface `Runnable`

The interface `Runnable` defines an abstract `run()` method as follows:

```

public interface Runnable {
    public abstract void run();
}

```

```

import java.lang.Thread;
public class NewClass implements Runnable {
    public void run() {
        System.out.println("hello from new thread");
    }
    public static void main (String args[]) {
        NewClass nc = new NewClass();
        Thread th = new Thread(nc);
        th.start(); // start() activates nc.run() in a new thread
    }
}

```

Fig. 6.22 Thread creation by using the interface `Runnable` based on the definition of a new class `NewClass`.

The predefined class `Thread` implements the interface `Runnable`. Therefore, each class which inherits from `Thread`, also implements the interface `Runnable`. Hence, instead of inheriting from `Thread` the newly defined class `NewClass` can directly implement the interface `Runnable`.

This way, objects of class `NewClass` are not thread objects. The creation of a new thread requires the generation of a new `Thread` object to which the object `NewClass` is passed as parameter. This is obtained by using the constructor

```
public Thread (Runnable target).
```

Using this constructor, the `start()` method of `Thread` activates the `run()` method of the `Runnable` object which has been passed as argument to the constructor.

This is obtained by the `run()` method of `Thread` which is specified as follows:

```

public void run() {
    if (target != null) target.run();
}

```

After activating `start()`, the `run()` method is executed by a separate thread which runs asynchronously to the calling thread. Thus, thread creation can be performed by the following steps:

- (1) definition of a class `NewClass` which implements `Runnable` and which defines a `run()` method with the code to be executed by the new thread;
- (2) instantiation of a `Thread` object using the constructor `Thread (Runnable target)` and of an object of `NewClass` which is passed to the `Thread` constructor;
- (3) activation of the `start()` method of the `Thread` object.

This is illustrated in Fig. 6.22 for a class `NewClass`. An object of this class is passed to the `Thread` constructor as parameter.

6.2.1.3 Further methods of the Thread class

A Java thread can wait for the termination of another Java thread `t` by calling `t.join()`. This call blocks the calling thread until the execution of `t` is terminated. There are three variants of this method:

- `void join()`: the calling thread is blocked until the target thread is terminated;
- `void join (long timeout)`: the calling thread is blocked until the target thread is terminated or the given time interval `timeout` has passed; the time interval is given in milliseconds;
- `void join (long timeout, int nanos)`: the behavior is similar to `void join (long timeout)`; the additional parameter allows a more exact specification of the time interval using an additional specification in nanoseconds.

The calling thread will not be blocked, if the target thread has not yet been started. The method

```
boolean isAlive()
```

of the Thread class gives information about the execution status of a thread: the method returns `true`, if the target thread has been started but has not yet been terminated; otherwise, `false` is returned. The `join()` and `isAlive()` methods have no effect on the calling thread. A name can be assigned to a specific thread by using the methods:

```
void setName (String name);
String getName ();
```

An assigned name can later be used to identify the thread. A name can also be assigned at thread creation by using the constructor `Thread (String name)`. The Thread class defines static methods which affect the calling thread or provide information about the program execution:

```
static Thread currentThread();
static void sleep (long milliseconds);
static void yield();
static int enumerate (Thread[] th_array);
static int activeCount();
```

Since these methods are static, they can be called without using a target Thread object. The call of `currentThread()` returns a reference to the Thread object of the calling thread. This reference can later be used to call nonstatic methods of the Thread object. The method `sleep()` blocks the execution of the calling thread until the specified time interval has passed; at this time, the thread again becomes ready for execution and can be assigned to an execution core or processor. The method `yield()` is a directive to the JVM to assign another thread with the same priority to the processor. If such a thread exists, then the scheduler of the JVM can bring this

thread to execution. The use of `yield()` is useful for JVM implementations without a time-sliced scheduling, if threads perform long-running computations which do not block. The method `enumerate()` yields a list of all active threads of the program. The return value specifies the number of `Thread` objects collected in the parameter array `th_array`. The method `activeCount()` returns the number of active threads in the program. The method can be used to determine the required size of the parameter array before calling `enumerate()`.

Example: Fig. 6.23 gives an example of a class for performing a matrix multiplication with multiple threads. The input matrices are read in into `in1` and `in2` by the main thread using the static method `ReadMatrix()`. The thread creation is performed by the constructor of the `MatMult` class, such that each thread computes one row of the result matrix. The corresponding computations are specified in the `run()` method. All threads access the same matrices `in1`, `in2` and `out` that have been allocated by the main thread. No synchronization is required, since each thread writes to a separate area of the result matrix `out`. □

6.2.2 Synchronization of Java Threads

The threads of a Java program access a shared address space. Suitable synchronization mechanisms have to be applied to avoid race conditions when a variable is accessed by several threads concurrently. Java provides `synchronized` blocks and methods to guarantee mutual exclusion for threads accessing shared data. A `synchronized` block or method avoids a concurrent execution of the block or method by two or more threads. A data structure can be protected by putting all accesses to it into `synchronized` blocks or methods, thus ensuring mutual exclusion. A synchronized increment operation of a counter can be realized by the following method `incr()`:

```
public class Counter {
    private int value = 0;
    public synchronized int incr() {
        value = value + 1;
        return value;
    }
}
```

Java implements the synchronization by assigning to each Java object an implicit mutex variable. This is achieved by providing the general class `Object` with an implicit mutex variable. Since each class is directly or indirectly derived from the class `Object`, each class inherits this implicit mutex variable, and every object instantiated from any class implicitly possess its own mutex variable. The activation of a `synchronized` method of an object `Ob` by a thread `t` has the following effects:

```

import java.lang.*;
import java.io.*;
class MatMult extends Thread {
    static int in1[] []; static int in2[] []; static int out[] [];
    static int n=3; int row;
    MatMult (int i) {
        row=i;
        this.start();
    }
    public void run() {
        //compute a row of the result matrix
        int i,j;
        for(i=0;i<n;i++) {
            out[row][i]=0;
            for (j=0;j<n;j++)
                out[row][i]=out[row][i]+in1[row][j]*in2[j][i];
        }
    }
    public static void ReadMatrix (int in[] []) {
        //read the input matrix
        int i,j;
        BufferedReader br=new BufferedReader(new
            InputStreamReader(System.in));
        System.out.println("Enter the Matrix : ");
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                try {
                    in[i][j]=Integer.parseInt(br.readLine());
                } catch(Exception e){ }
    }
    public static void PrintMatrix (int out[] []) {
        //print the result matrix
        int i,j;
        System.out.println("OUTPUT :");
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                System.out.println(out[i][j]);
    }
    public static void main(String args[]) {
        int i,j;
        in1=new int[n][n]; in2=new int[n][n];
        out=new int[n][n];
        ReadMatrix(in1); ReadMatrix(in2);
        MatMult mat[] = new MatMult[n];
        for(i=0;i<n;i++)
            mat[i]=new MatMult(i);
        try {
            for(i=0;i<n;i++)
                mat[i].join();
        } catch(Exception e){ }
        PrintMatrix(out);
    }
}

```

Fig. 6.23 Parallel matrix multiplication in Java.

- When starting the `synchronized` method, `t` implicitly tries to lock the mutex variable of `Ob`. If the mutex variable is already locked by another thread `s`, thread `t` is blocked. The blocked thread becomes ready for execution again when the mutex variable is released by the locking thread `s`. The called `synchronized` method will only be executed after successfully locking the mutex variable of `Ob`.
- When `t` leaves the `synchronized` method called, it implicitly releases the mutex variable of `Ob`, so that it can be locked by another thread.

A `synchronized` access to an object can be realized by declaring all methods accessing the object as `synchronized`. The object should only be accessed with these methods to guarantee mutual exclusion.

In addition to `synchronized` methods, Java provides `synchronized` blocks: such a block is started with the keyword `synchronized` and the specification of an arbitrary object that is used for the synchronization in parenthesis. Instead of an arbitrary object, the synchronization is usually performed with the object whose method contains the `synchronized` block. The above method for the incrementation of a counter variable can be realized using a `synchronized` block as follows:

```
public int incr() {
    synchronized (this) {
        value = value + 1; return value;
    }
}
```

The synchronization mechanism of Java can be used for the realization of **fully synchronized objects** (also called **atomic objects**); these can be accessed by an arbitrary number of threads without any additional synchronization. To avoid race conditions, the synchronization has to be performed within the methods of the corresponding class of the objects. This class must have the following properties:

- all methods must be declared `synchronized`;
- no public entries are allowed that can be accessed without using a local method;
- all entries are consistently initialized by the constructors of the class;
- the objects remain in a consistent state also in case of exceptions.

Figure 6.24 demonstrates the concept of fully synchronized objects for the example of a class `ExpandableArray`; this is a simplified version of the predefined synchronized class `java.util.Vector`, see also [128]. The class implements an adaptable array of arbitrary objects, i.e., the size of the array can be increased or decreased according to the number of objects to be stored. The adaptation is realized by the method `add()`: if the array data is fully occupied when trying to add a new object, the size of the array will be increased by allocating a larger array and using the method `arraycopy()` from the `java.lang.System` class to copy the content of the old array into the new array. Without the synchronization included, the class could not be used concurrently by more than one thread safely. A conflict could occur, if, e.g., two threads tried to perform an `add()` operation at the same time.

```

import java.lang.*;
import java.util.*;
public class ExpandableArray {
    private Object[] data;
    private int size = 0;
    public ExpandableArray(int cap) {
        data = new Object[cap];
    }
    public synchronized int size() {
        return size;
    }
    public synchronized Object get(int i)
        throws NoSuchElementException {
        if (i < 0 || i >= size)
            throw new NoSuchElementException();
        return data[i];
    }
    public synchronized void add(Object x) {
        if (size == data.length) { // array too small
            Object[] od = data;
            data = new Object[3 * (size + 1) / 2];
            System.arraycopy(od, 0, data, 0, od.length);
        }
        data[size++] = x;
    }
    public synchronized void removeLast()
        throws NoSuchElementException {
        if (size == 0)
            throw new NoSuchElementException();
        data[--size] = null;
    }
}

```

Fig. 6.24 Example for a fully synchronized class.

6.2.2.1 Deadlocks

The use of fully synchronized classes avoids the occurrence of race conditions, but may lead to deadlocks when threads are synchronized with different objects. This is illustrated in Fig. 6.25 for a class `Account` which provides a method `swapBalance()` to swap account balances, see [128]. A deadlock can occur when `swapBalance()` is executed by two threads *A* and *B* concurrently: For two account objects *a* and *b*, if *A* calls `a.swapBalance(b)` and *B* calls `b.swapBalance(a)` and *A* and *B* are executed on different processors or cores, a deadlock occurs with the following execution order:

- **time T_1 :** thread *A* calls `a.swapBalance(b)` and locks the mutex variable of object *a*;
- **time T_2 :** thread *B* calls `b.swapBalance(a)` for object *a* and executes this function;

```

public class Account {
    private long balance;
    synchronized long getBalance() {return balance;}
    synchronized void setBalance(long v) {
        balance = v;
    }
    synchronized void swapBalance(Account other) {
        long t = getBalance();
        long v = other.getBalance();
        setBalance(v);
        other.setBalance(t);
    }
}

```

Fig. 6.25 Example for a deadlock situation.

- **time T_2 :** thread B calls `b.swapBalance(a)` and locks the mutex variable of object b ;
- **time T_3 :** thread A calls `b.getBalance()` and blocks because the mutex variable of b has previously been locked by thread B ;
- **time T_3 :** thread B calls `getBalance()` for object b and executes this function;
- **time T_4 :** thread B calls `a.getBalance()` and blocks because the mutex variable of a has previously been locked by thread A .

The execution order is illustrated in Fig. 6.26. After time T_4 , both threads are blocked: thread A is blocked, since it could not acquire the mutex variable of object b . This mutex variable is owned by thread B and only B can free it. Thread B is blocked, since it could not acquire the mutex variable of object a . This mutex variable is owned by thread A and only A can free it. Thus, both threads are blocked and none of them can proceed; a deadlock has occurred.

Deadlocks typically occur, if different threads try to lock the mutex variables of the same objects in different orders. For the example in Fig. 6.25, thread A tries to lock first a and then b , whereas thread B tries to lock first b and then a . In this situation, a deadlock can be avoided by a backoff strategy or by using the same locking order for each thread, see also Sect. 6.1.2. A unique ordering of objects can be obtained by using the Java method `System.identityHashCode()` which refers to the default implementation `Object.hashCode()`, see [128]. But any other unique

time	operation Thread A	operation Thread B	owner mutex a	owner mutex b
T_1	<code>a.swapBalance(b)</code>		A	—
T_2	<code>t = getBalance()</code>	<code>b.swapBalance(a)</code>	A	B
T_3	blocked with respect to b	<code>t = getBalance()</code>	A	B
T_4		blocked with respect to a	A	B

Fig. 6.26 Execution order to cause a deadlock situation for the class in Fig. 6.25.

```
public void swapBalance(Account other) {
    if (other == this) return;
    else if (System.identityHashCode(this) <
             System.identityHashCode(other))
        this.doSwap(other);
    else other.doSwap(this);
}
protected synchronized void doSwap(Account other) {
    long t = getBalance();
    long v = other.getBalance();
    setBalance(v);
    other.setBalance(t);
}
```

Fig. 6.27 Deadlock-free implementation of `swapBalance()` from Fig. 6.25.

object ordering can also be used. Thus, we can give an alternative formulation of `swapBalance()` which avoids deadlocks, see Fig. 6.27. The new formulation also contains an alias check to ensure that the operation is only executed if different objects are used. The method `swapBalance()` is not declared `synchronized` any more.

For the synchronization of Java methods, several issues should be considered to make the resulting programs efficient and safe:

- Synchronization is expensive. Therefore, `synchronized` methods should only be used for methods that can be called concurrently by several threads and that may manipulate common object data.
If an application ensures that a method is always executed by a single thread at each point in time, then a synchronization can be avoided to increase efficiency.
- Synchronization should be restricted to critical regions to reduce the time interval of locking. For larger methods, the use of `synchronized` blocks instead of `synchronized` methods should be considered.
- To avoid unnecessary sequentializations, the mutex variable of the same object should not be used for the synchronization of different, noncontiguous critical sections.
- Several Java classes are internally synchronized; Examples are `Hashtable`, `Vector`, and `StringBuffer`. No additional synchronization is required for objects of these classes.
- If an object requires synchronization, the object data should be put into `private` or `protected` instance fields to inhibit nonsynchronized accesses from outside. All object methods accessing the instance fields should be declared as `synchronized`.
- For cases in which different threads access several objects in different orders, deadlocks can be prevented by using the same lock order for each thread.

Fig. 6.28 Synchronization class with variable lock granularity.

```
public class MyMutex {
    protected Thread OwnerThread = null;
    public void getMyMutex() {
        while (!tryGetMyMutex()) {
            try { Thread.sleep(100); }
            catch (InterruptedException e) { }
        }
    }
    public synchronized boolean tryGetMyMutex() {
        if (OwnerThread == null) {
            OwnerThread = Thread.currentThread();
            return true;
        }
        else return false;
    }
    public synchronized void freeMyMutex() {
        if (OwnerThread == Thread.currentThread())
            OwnerThread = null;
    }
}
```

6.2.2.2 Synchronization with variable lock granularity

To illustrate the use of the synchronization mechanism of Java, we consider a synchronization class with a variable lock granularity, which has been adapted from [147].

The new class `MyMutex` allows the synchronization of arbitrary object accesses by explicitly acquiring and releasing objects of the class `MyMutex`, thus realizing a lock mechanism similar to mutex variables in Pthreads, see Sect. 6.1.2, page 292. The new class also enables the synchronization of threads accessing different objects. The class `MyMutex` uses an instance field `OwnerThread` which indicates which thread has currently acquired the synchronization object. Figure 6.28 shows a first draft of the implementation of `MyMutex`.

The method `getMyMutex` can be used to acquire the explicit lock of the synchronization object for the calling thread. The lock is given to the calling thread by assigning `Thread.currentThread()` to the instance field `OwnerThread`. The synchronized method `freeMyMutex()` can be used to release a previously acquired explicit lock; this is implemented by assigning `null` to the instance field `OwnerThread`. If a synchronization object has already been locked by another thread, `getMyMutex()` repeatedly tries to acquire the explicit lock after a fixed time interval of 100 ms. The method `getMyMutex()` is not declared synchronized. The synchronized method `tryGetMyMutex()` is used to access the instance field `OwnerThread`. This protects the critical section for acquiring the explicit lock by using the implicit mutex variable of the synchronization object. This mutex variable is used both for `tryGetMyMutex()` and `freeMyMutex()`.

Fig. 6.29 Implementation variant of `getMyMutex()`.

```
public void getMyMutex() {
    for ( ; ; ) {
        synchronized(this) {
            if (OwnerThread == null) {
                OwnerThread = Thread.currentThread();
                break;
            }
        }
        try { Thread.sleep(100); }
        catch (InterruptedException e) { }
    }
}
```

Fig. 6.30 Implementation of a counter class with synchronization by an object of class `MyMutex`.

```
public class Counter {
    private int value;
    private MyMutex flag = new MyMutex();
    public int incr() {
        int res;
        flag.getMyMutex();
        value = value + 1;
        res = value;
        flag.freeMyMutex();
        return res;
    }
}
```

If `getMyMutex()` would have been declared `synchronized`, the activation of `getMyMutex()` by a thread T_1 would lock the implicit mutex variable of the synchronization object of the class `MyMutex` before entering the method. If another thread T_2 holds the explicit lock of the synchronization object, T_2 cannot release this lock with `freeMyMutex()`, since this would require to lock the implicit mutex variable which is held by T_1 . Thus, a deadlock would result. The use of an additional method `tryGetMyMutex()` can be avoided by using a `synchronized` block within `getMyMutex()`, see Fig. 6.29.

Objects of the new synchronization class `MyMutex` can be used for the explicit protection of critical sections. This can be illustrated for a counter class `Counter` to protect the counter manipulation, see Fig. 6.30.

6.2.2.3 Synchronization of static methods

The implementation of `synchronized` blocks and methods based on the implicit object mutex variables works for all methods that are activated with respect to an object. Static methods of a class are not activated with respect to an object, and thus there is no implicit object mutex variable. Nevertheless, static methods can

```

public class MyStatic {
    public static synchronized void staticMethod(MyStatic obj) {
        // here, the class mutex is used
        obj.nonStaticMethod();
        synchronized(obj) {
            // here, additionally, the object mutex is used
        }
    }
    public synchronized void nonStaticMethod() {
        // using the object mutex
    }
}

```

Fig. 6.31 Synchronization of static methods.

also be declared `synchronized`. In this case, the synchronization is implemented by using the implicit mutex variable of the corresponding class object of the class `java.lang.Class` (Class mutex variable). An object of this class is automatically generated for each class defined in a Java program.

Thus, static and nonstatic methods of a class are synchronized by using different implicit mutex variables. A static `synchronized` method can acquire both the mutex variable of the `Class` object and of an object of this class by using an object of this class for a `synchronized` block or by activating a `synchronized` nonstatic method for an object of this class. This is illustrated in Fig. 6.31. Similarly, a `synchronized` nonstatic method can also acquire both the mutex variables of the object and of the `Class` object by calling a `synchronized` static method. For an arbitrary class `C1`, the `Class` mutex variable can be directly used for a `synchronized` block by using

```
synchronized (C1.class) { /*Code*/ }
```

6.2.3 Wait and Notify

In some situations, it is useful for a thread to wait for an event or condition. As soon as the event occurs, the thread executes a predefined action. The thread waits as long as the event does not occur or the condition is not fulfilled. The event can be signaled by another thread; similarly, another thread can make the condition to be fulfilled. `Pthreads` provides condition variables for these situations. Java provides a similar mechanism via the methods `wait()` and `notify()` of the predefined `Object` class. These methods are available for each object of any class which is explicitly or implicitly derived from the `Object` class. Both methods can only be used within `synchronized` blocks or methods. A typical usage pattern for `wait()` is:

```
synchronized (lockObject) {
    while (!condition) { lockObject.wait(); }
    Action();
}
```

The call of `wait()` blocks the calling thread until another thread calls `notify()` for the same object. When a thread blocks by calling `wait()`, it releases the implicit mutex variable of the object used for the synchronization of the surrounding synchronized method or block. Thus, this mutex variable can be acquired by another thread.

Several threads may block waiting for the same object. Each object maintains a list of waiting threads. When another thread calls the `notify()` method of the same object, one of the waiting threads of this object is woken up and can continue running. Before resuming its execution, this thread first acquires the implicit mutex variable of the object. If this is successful, the thread performs the action specified in the program. If this is not successful, the thread blocks and waits until the implicit mutex variable is released by the owning thread by leaving a synchronized method or block.

The methods `wait()` and `notify()` work similarly as the operations `pthread_cond_wait()` and `pthread_cond_signal()` for condition variables in Pthreads, see Sect. 6.1.3, page 300. The methods `wait()` and `notify()` are implemented using an implicit waiting queue for each object; this waiting queue contains all blocked threads waiting to be woken up by a `notify()` operation. The waiting queue does not contain those threads that are blocked waiting for the implicit mutex variable of the object.

The Java language specification does not specify which of the threads in the waiting queue is woken up if `notify()` is called by another thread. The method `notifyAll()` can be used to wake up all threads in the waiting queue; this has a similar effect as `pthread_cond_broadcast()` in Pthreads. The method `notifyAll()` also has to be called in a synchronized block or method.

6.2.3.1 Producer-Consumer pattern

The Java waiting and notification mechanism described above can be used for the implementation of a producer–consumer pattern using an item buffer of fixed size. Producer threads can put new items into the buffer and consumer threads can remove items from the buffer. Figure 6.32 shows a thread-safe implementation of such a buffer mechanism adapted from [128] using the `wait()` and `notify()` methods of Java. When creating an object of the class `BoundedBufferSignal`, an array array of a given size `capacity` is generated; this array is used as buffer. The indices `putptr` and `takeptr` indicate the next position in the buffer to put or take the next item, respectively; these indices are used in a circular way.

The class provides a `put()` method to enable a producer to enter an item into the buffer and a `take()` method to enable a consumer to remove an item from


```

public class BoundedBufferSignal {
    private final Object[] array;
    private int putptr = 0;
    private int takeptr = 0;
    private int numel = 0; // number of items in buffer
    public BoundedBufferSignal (int capacity)
        throws IllegalArgumentException {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        array = new Object[capacity];
    }
    public synchronized int size() {return numel; }
    public int capacity() {return array.length;}
    public synchronized void put(Object obj)
        throws InterruptedException {
        while (numel == array.length)
            wait(); // buffer full
        array [putptr] = obj;
        putptr = (putptr +1) % array.length;
        if (numel++ == 0)
            notifyAll(); // wake up all threads
    }
    public synchronized Object take()
        throws InterruptedException {
        while (numel == 0)
            wait(); // buffer empty
        Object x = array [takeptr];
        takeptr = (takeptr +1) % array.length;
        if (numel-- == array.length)
            notifyAll(); // wake up all threads
        return x;
    }
}

```

Fig. 6.32 Realization of a thread-safe buffer mechanism using Java `wait()` and `notify()`.

the buffer. A buffer object can have one of three states: full, partially full, and empty. Figure 6.33 illustrates the possible transitions between the states when calling `take()` or `put()`. The states are characterized by the following conditions:

state	condition	put possible	take possible
full	<code>size == capacity</code>	no	yes
partially full	<code>0 < size < capacity</code>	yes	yes
empty	<code>size == 0</code>	yes	no

If the buffer is full, the execution of the `put()` method by a producer thread will block the executing thread; this is implemented by a `wait()` operation. If the



Fig. 6.33 Illustration of the states of a thread-safe buffer mechanism.

`put()` method is executed for a previously empty buffer, all waiting (consumer) threads will be woken up using `notifyAll()` after the item has been entered into the buffer. If the buffer is empty, the execution of the `take()` method by a consumer thread will block the executing thread using `wait()`. If the `take()` method is executed for a previously full buffer, all waiting (producer) threads will be woken up using `notifyAll()` after the item has been removed from the buffer. The implementation of `put()` and `take()` ensures that each object of the class `BoundedBufferSignal` can be accessed concurrently by an arbitrary number of threads without race conditions.

6.2.3.2 Modification of the `MyMutex` class

The methods `wait()` and `notify()` can be used to improve the synchronization class `MyMutex` from Fig. 6.28 by avoiding the active waiting in the method `getMyMutex()`, see Fig. 6.34 (according to [147]).

Additionally, the modified implementation realizes a nested locking mechanism which allows multiple locking of a synchronization object by the same thread. The number of locks is counted in the variable `lockCount`; this variable is initialized to 0 and is incremented or decremented by each call of `getMyMutex()` or `freeMyMutex()`, respectively. In Fig. 6.34, the method `getMyMutex()` is now also declared `synchronized`. With the implementation in Fig. 6.28, this would lead to a deadlock. But in Fig. 6.34, no deadlock can occur, since the activation of `wait()` releases the implicit mutex variable before the thread is suspended and inserted into the waiting queue of the object.

6.2.3.3 Barrier Synchronization

A barrier synchronization is a synchronization point at which each thread waits until all participating threads have reached this synchronization point. Only then the threads proceed with their execution. A barrier synchronization can be implemented in Java using `wait()` and `notify()`. This is shown in Fig. 6.35 for a class `Barrier`, see also [147]. The `Barrier` class contains a constructor which initializes a `Barrier` object with the number of threads to wait for (`t2w4`). The actual synchronization is provided by the method `waitForRest()`. This method must be called by each thread at the intended synchronization point. Within the method, each thread decrements `t2w4` and calls `wait()` if `t2w4` is > 0 . This blocks each

Fig. 6.34 Realization of the synchronization class `MyMutex` with `wait()` and `notify()` avoiding active waiting.

```
public synchronized void getMyMutex() {
    while (!tryGetMyMutex()) {
        try { wait(); }
        catch (InterruptedException e) { }
    }
}

public synchronized boolean tryGetMyMutex() {
    if (OwnerThread == null) {
        OwnerThread = Thread.currentThread();
        lockCount = 1; return true;
    }
    if (OwnerThread == Thread.currentThread()) {
        lockCount++; return true;
    }
    return false;
}

public synchronized Thread getMutexOwner() {
    return OwnerThread;
}

public synchronized void freeMyMutex() {
    if (OwnerThread == Thread.currentThread()) {
        if (--lockCount == 0) {
            OwnerThread = null;
            notify();
        }
    }
}
```

arriving thread within the `Barrier` object. The last arriving thread wakes up all waiting threads using `notifyAll()`.

Objects of the `Barrier` class can be used only once, since the synchronization counter `t2w4` is decremented to 0 during the synchronization process. An example for the use of the `Barrier` class for the synchronization of a multi-phase computation is given in Fig. 6.36, see also [147]. The program illustrates an algorithm with three phases (`doPhase1()`, `doPhase2()`, `doPhase3()`) which are separated from each other by a barrier synchronization using `Barrier` objects `bp1`, `bp2`, and `bpEnd`. Each of the threads created in the constructor of `ProcessIt` executes the three phases

6.2.3.4 Condition Variables

The mechanism provided by `wait()` and `notify()` in Java has some similarities to the synchronization mechanism of condition variables in Pthreads, see Sect. 6.1.3, page 300. The main difference lies in the fact that `wait()` and `notify()` are provided by the general `Object` class. Thus, the mechanism is implicitly bound to the internal mutex variable of the object for which `wait()` and `notify()`

```

public class Barrier() {
    private int t2w4;
    private InterruptedException e;
    public Barrier(int n) {
        this.t2w4 = n;
    }
    public synchronized int waitForRest()
        throws InterruptedException {
        int nThreads = --t2w4;
        if (e != null) throw e;
        if (t2w4 <= 0) {
            notifyAll(); return nThreads;
        }
        while (t2w4 > 0) {
            if (e != null) throw e;
            try { wait(); }
            catch(InterruptedException e) { this.e = e; notifyAll(); }
        }
        return nThreads;
    }
}

```

Fig. 6.35 Realization of a barrier synchronization in Java with `wait()` and `notify()`.

are activated. This facilitates the use of this mechanism by avoiding the explicit association of a mutex variable as needed when using the corresponding mechanism in Pthreads. But the fixed binding of `wait()` and `notify()` to a specific mutex variable also reduces the flexibility, since it is not possible to combine an arbitrary mutex variable with the waiting queue of an object.

When calling `wait()` or `notify()`, a Java thread must be the owner of the mutex variable of the corresponding object; otherwise an exception `IllegalMonitorStateException` is raised. With the mechanism of `wait()` and `notify()`, it is not possible to use the same mutex variable for the synchronization of the waiting queues of different objects. This would be useful, e.g., for the implementation of producer and consumer threads with a common data buffer, see, e.g., Fig. 6.18. But `wait()` and `notify()` can be used for the realization of a new class which mimics the mechanism of condition variables in Pthreads. Figure 6.37 shows an implementation of such a class `CondVar`, see also [147, 128]. The class `CondVar` provides the methods `cvWait()`, `cvSignal()` and `cvBroadcast()` which mimic the behavior of `pthread_cond_wait()`, `pthread_cond_signal()` and `pthread_cond_broadcast()`, respectively. These methods allow the use of an arbitrary mutex variable for the synchronization. This mutex variable is provided as a parameter of type `MyMutex` for each of the methods, see Fig. 6.37.

Thus, a single mutex variable of type `MyMutex` can be used for the synchronization of several condition variables of type `CondVar`. When calling `cvWait()`, a thread will be blocked and put in the waiting queue of the corresponding object of type `CondVar`. The internal synchronization within `cvWait()` is performed with

Fig. 6.36 Use of the `Barrier` class for the realization of a multi-phase algorithm.

```
public class ProcessIt implements Runnable {
    String is[];
    Barrier bpStart, bp1, bp2, bpEnd;
    public ProcessIt(String sources[]) {
        is = sources;
        bpStart = new Barrier(sources.length);
        bp1 = new Barrier(sources.length);
        bp2 = new Barrier(sources.length);
        bpEnd = new Barrier(sources.length);
        for (int i = 0; i < sources.length; i++)
            new Thread(this).start();
    }
    public void run() {
        try {
            int i = bpStart.waitForRest();
            doPhase1(is[i]);
            bp1.waitForRest();
            doPhase2(is[i]);
            bp2.waitForRest();
            doPhase3(is[i]);
            bpEnd.waitForRest();
        }
        catch (InterruptedException e)
        {}
    }
    public static void main(String args[]) {
        ProcessIt pi = new ProcessIt(args);
    }
}
```

the internal mutex variable of this object. The class `CondVar` also allows a simple porting of Pthreads programs with condition variables to Java programs.

Fig. 6.38 shows as example the realization of a buffer mechanism with producer and consumer threads by using the new class `CondVar`, see also [128]. A producer thread can insert objects into the buffer by using the method `put()`. A consumer thread can remove objects from the buffer by using the method `take()`. The condition objects `notFull` and `notEmpty` of type `CondVar` use the same mutex variable `mutex` for synchronization.

6.2.4 Extended Synchronization Patterns

The synchronization mechanisms provided by Java can be used to implement more complex synchronization patterns which can then be used in parallel application programs. This will be demonstrated in the following for the example of a semaphore mechanism, see page 157.

```

public class CondVar {
    private MyMutex syncVar; /* use MyMutex for synchronization */
    public CondVar() {
        this(new MyMutex());
    }
    public CondVar(MyMutex sv) {
        syncVar = sv;
    }
    public void cvWait() throws InterruptedException {
        cvWait(syncVar, 0);
    }
    public void cvWait(MyMutex sv) throws InterruptedException {
        cvWait(sv, 0);
    }
    public void cvWait(int millis) throws InterruptedException {
        cvWait(syncVar, millis);
    }
    public void cvWait(MyMutex sv, int millis)
        throws InterruptedException {
        int i = 0;
        InterruptedException exception;
        synchronized (this) {
            if (sv.getMutexOwner() != Thread.currentThread())
                throw new IllegalMonitorStateException ("thread not owner");
            while (sv.getMutexOwner() == Thread.currentThread()) {
                i++; sv.freeMyMutex();
            }
            try { if (millis == 0) wait(); else wait(millis); }
            catch (InterruptedException e) { exception = e; }
        }
        for (; i > 0; i--) sv.getMyMutex();
        if (exception != null) throw exception;
    }
    public void cvSignal() {
        cvSignal(syncVar);
    }
    public synchronized void cvSignal(MyMutex sv) {
        if (sv.getMutexOwner() != Thread.currentThread())
            throw new IllegalMonitorStateException ("thread not owner");
        notify();
    }
    public void cvBroadcast() {
        cvBroadcast(syncVar);
    }
    public synchronized void cvBroadcast(MyMutex sv) {
        if (sv.getMutexOwner() != Thread.currentThread())
            throw new IllegalMonitorStateException ("thread not owner");
        notifyAll();
    }
}

```

Fig. 6.37 Class CondVar for the realization of the Pthreads condition variable mechanism using the Java signaling mechanism.

```

class PThreadsStyleBuffer {
    private final MyMutex mutex = new MyMutex();
    private final CondVar notFull = new CondVar(mutex);
    private final CondVar notEmpty = new CondVar(mutex);
    private int count = 0;
    private int takePtr = 0;
    private int putPtr = 0;
    private final Object[] array;

    public PThreadsStyleBuffer(int capacity) {
        array = new Object[capacity];
    }

    public void put(Object x) throws InterruptedException {
        mutex.getMyMutex();
        try {
            while (count == array.length)
                notFull.cvWait();

            array[putPtr] = x;
            putPtr = (putPtr + 1) % array.length;
            ++count;
            notEmpty.cvSignal();
        }
        finally {
            mutex.freeMyMutex();
        }
    }

    public Object take() throws InterruptedException {
        Object x = null;
        mutex.getMyMutex();
        try {
            while (count == 0)
                notEmpty.cvWait();

            x = array[takePtr];
            array[takePtr] = null;
            takePtr = (takePtr + 1) % array.length;
            --count;
            notFull.cvSignal();
        }
        finally {
            mutex.freeMyMutex();
        }
        return x;
    }
}

```

Fig. 6.38 Implementation of a buffer mechanism for producer and consumer threads.

A semaphore mechanism can be implemented in Java by using `wait()` and `notify()`. Figure 6.39 shows a simple implementation, see also [128, 147]. The method `acquire()` waits (if necessary), until the internal counter of the semaphore

Fig. 6.39 Implementation of a semaphore mechanism.

```
public class Semaphore {
    private long counter;
    public Semaphore(long init) {
        counter = init;
    }
    public void acquire()
        throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
        synchronized (this) {
            try {
                while (counter <= 0) wait();
                counter--;
            }
            catch (InterruptedException ie) {
                notify(); throw ie;
            }
        }
    }
    public synchronized void release() {
        counter++;
        notify();
    }
}
```

object has reached at least the value 1. As soon as this is the case, the counter is decremented. The method `release()` increments the counter and uses `notify()` to wake up a waiting thread that has been blocked in `acquire()` by calling `wait()`. A waiting thread can only exist, if the counter had the value 0 before incrementing it. Only in this case, it can be blocked in `acquire()`. Since the counter is only incremented by one, it is sufficient to wake up a single waiting thread. An alternative would be to use `notifyAll()`, which wakes up all waiting threads. Only one of these threads would succeed in decrementing the counter, which would then have the value 0 again. Thus, all other threads that had been woken up would be blocked again by calling `wait`.

The semaphore mechanism shown in Fig. 6.39 can be used for the synchronization of producer and consumer threads. A similar mechanism has already been implemented in Fig. 6.32 by using `wait()` and `notify()` directly. Fig. 6.41 shows an alternative implementation with semaphores, see [128]. The producer stores the objects generated into a buffer of fixed size, the consumer retrieves objects from this buffer for further processing. The producer can only store objects in the buffer, if the buffer is not full. The consumer can only retrieve objects from the buffer, if the buffer is not empty. The actual buffer management is done by a separate class `BufferArray` which provides methods `insert()` and `extract()` to insert and retrieve objects, see Fig. 6.40. Both methods are `synchronized`, so multiple threads can access objects of this class without conflicts. The class `BufferArray` does not provide a mechanism to control buffer overflow.

Fig. 6.40 Class `BufferArray` for buffer management.

```
public class BufferArray {
    private final Object[] array;
    private int putptr = 0;
    private int takeptr = 0;
    public BufferArray (int n) {
        array = new Object[n];
    }
    public synchronized void insert (Object obj) {
        array[putptr] = obj;
        putptr = (putptr + 1) % array.length;
    }
    public synchronized Object extract() {
        Object x = array[takeptr];
        array[takeptr] = null;
        takeptr = (takeptr + 1) % array.length;
        return x;
    }
}
```

The class `BoundedBufferSema` in Fig. 6.41 provides the methods `put()` and `take()` to store and retrieve objects in a buffer. Two semaphores `putPermits` and `takePermits` are used to control the buffer management. At each point in time, these semaphores count the number of permits to store (producer) and retrieve (consumer) objects. The semaphore `putPermits` is initialized to the buffer size, the semaphore `takePermits` is initialized to 0. When storing an objects by using `put()`, the semaphore `putPermits` is decremented with `acquire()`; if the buffer is full, the calling thread is blocked when doing this. After an object has been stored in the buffer with `insert()`, a waiting consumer thread (if present) is woken up by calling `release()` for the semaphore `takePermits`. Retrieving an object with `take()` works similarly with the role of the semaphores exchanged.

In comparison to the implementation in Fig. 6.32, the new implementation in Fig. 6.41 uses two separate objects (of type `Semaphore`) for buffer control. Depending on the specific situation, this can lead to a reduction of the synchronization overhead: in the implementation from Fig. 6.32 *all* waiting threads are woken up in `put()` and `take()`. But only one of these can proceed and retrieve an object from the buffer (consumer) or store an object into the buffer (producer). All other threads are blocked again. In the implementation from Fig. 6.41, only one thread is woken up.

6.2.5 Thread Scheduling in Java

A Java program may consist of several threads which can be executed on one or several of the processors of the execution platform. The threads which are ready for

Fig. 6.41 Buffer management with semaphores.

```

public class BoundedBufferSema {
    private final BufferArray buff;
    private final Semaphore putPermits;
    private final Semaphore takePermits;
    public BoundedBufferSema(int capacity)
        throws IllegalArgumentException {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        buff = new BufferArray(capacity);
        putPermits = new Semaphore(capacity);
        takePermits = new Semaphore(0);
    }
    public void put(Object x)
        throws InterruptedException {
        putPermits.acquire();
        buff.insert(x);
        takePermits.release();
    }
    public Object take()
        throws InterruptedException {
        takePermits.acquire();
        Object x = buff.extract();
        putPermits.release();
        return x;
    }
}

```

execution compete for execution on a free processor. The programmer can influence the mapping of threads to processors by assigning priorities to the threads. The minimum, maximum, and default priorities for Java threads are specified in the following fields of the Thread class:

```

public static final int MIN_PRIORITY // normally 1
public static final int MAX_PRIORITY // normally 10
public static final int NORM_PRIORITY // normally 5

```

A *large* priority value corresponds to a *high* priority. The thread which executes the `main()` method of a class has by default the priority `Thread.NORM_PRIORITY`. A newly created thread has by default the same priority as the generating thread. The current priority of a thread can be retrieved or dynamically changed by using the methods

```

public int getPriority();
public int setPriority(int prio);

```

of the Thread class. If there are more executable threads than free processors, a thread with a larger priority is usually favored by the scheduler of the JVM. The exact mechanism for selecting a thread for execution may depend on the implementation of a specific JVM. The Java specification does not define an exact scheduling mechanism

to increase flexibility for the implementation of the JVM on different operating systems and different execution platforms. For example, the scheduler might always bring the thread with the largest priority to execution, but it could also integrate an aging mechanism to ensure that threads with a lower priority will be mapped to a processor from time to time to avoid starvation and implement fairness.

Since there is no exact specification for the scheduling of threads with different priorities, priorities cannot be used to replace synchronization mechanisms. Instead, priorities can only be used to express the relative importance of different threads to bring the most important thread to execution in case of doubt.

When using threads with different priorities, the problem of **priority inversion** can occur, see also Sect. 6.1.11, page 332. A priority inversion happens if a thread with a high priority is blocked to wait for a thread with a low priority, e.g., because this thread has locked the same mutex variable that the thread with the high priority tries to lock. The thread with a low priority can be inhibited from proceeding its execution and releasing the mutex variable as soon as a thread with a medium priority is ready for execution. In this constellation, the thread with high priority can be prevented from execution in favor of the thread with a medium priority.

The problem of priority inversion can be avoided by using **priority inheritance**, see also Sect. 6.1.11: if a thread with high priority is blocked, e.g., because of an activation of a `synchronized` method, then the priority of the thread that currently controls the critical synchronization object will be increased to the high priority of the blocked thread. Then, no thread with medium priority can inhibit the thread with high priority from execution. Many JVMs use this method, but this is not guaranteed by the Java specification.

6.2.6 Package `java.util.concurrent`

The `java.util.concurrent` package provides additional synchronization mechanisms and classes which are based on the standard synchronization mechanisms described in the previous section, like `synchronized` blocks, `wait()`, and `notify()`. The package is available for Java platforms starting with the Java2 platform (Java2 Standard Edition 5.0, J2SE 5.0).

The additional mechanisms provide more abstract and flexible synchronization operations, including atomic variables, lock variables, barrier synchronization, condition variables, and semaphores as well as different thread-safe data structures like queues, hash maps, or array lists. The additional classes are similar to those described in [128]. In the following, we give a short overview of the package and refer to [76] for a more detailed description.

6.2.6.1 Semaphore mechanism

The class `Semaphore` provides an implementation of a counting semaphore, which is similar to the mechanism given in Fig. 6.17. Internally, a `Semaphore` object maintains a counter which counts the number of permits.

The most important methods of the `Semaphore` class are:

```
void acquire();
void release();
boolean tryAcquire()
boolean tryAcquire(int permits, long timeout,
                   TimeUnit unit)
```

The method `acquire()` asks for a permit and blocks the calling thread if no permit is available. If a permit is currently available, the internal counter for the number of available permits is decremented and control is returned to the calling thread.

The method `release()` adds a permit to the semaphore by incrementing the internal counter. If another thread is waiting for a permit of this semaphore, this thread is woken up. The method `tryAcquire()` asks for a permit to a semaphore object. If a permit is available, a permit is acquired by the calling thread and control is returned immediately with return value `true`. If no permit is available, control is also returned immediately, but with return value `false`; thus, in contrast to `acquire()`, the calling thread is not blocked. There exist different variants of the method `tryAcquire()` with varying parameters allowing the additional specification of a number of permits to acquire (parameter `permits`), a waiting time (parameter `timeout`) after which the attempt of acquiring the specified number of permits is given up with return value `false`, as well as a time unit (parameter `unit`) for the waiting time. If not enough permits are available when calling a timed `tryAcquire()`, the calling thread is blocked until one of the following events occurs:

- the number of requested permits becomes available because other threads call `release()` for this semaphore; in this case, control is returned to the calling thread with return value `true`;
- the specified waiting time elapses; in this case, control is returned with return value `false`; no permit is acquired in this case, also if some of the requested permits would have been available.

6.2.6.2 Barrier synchronization

The class `CyclicBarrier` provides an implementation of a barrier synchronization. The prefix *cyclic* refers to the fact that an object of this class can be re-used again after all participating threads have passed the barrier. The constructors of the class

```
public CyclicBarrier(int n);  
public CyclicBarrier(int n, Runnable action);
```

allow the specification of a number `n` of threads that must pass the barrier before execution continues after the barrier. The second constructor allows the additional specification of an operation `action` that is executed as soon as all threads have passed the barrier. The most important methods of `CyclicBarrier` are `await()` and `reset()`. By calling `await()` a thread waits at the barrier until the specified number of threads have reached the barrier. A barrier object can be reset into its original state by calling `reset()`.

6.2.6.3 Lock Mechanisms

The package `java.util.concurrent.locks` contains interfaces and classes for locks and for waiting for the occurrence of conditions. The interface `Lock` defines locking mechanisms which go beyond the standard `synchronized` methods and blocks and are not limited to the synchronization with the implicit mutex variables of the objects used. The most important methods of `Lock` are

```
void lock();  
boolean tryLock();  
boolean tryLock(long time, TimeUnit unit);  
void unlock();
```

The method `lock()` tries to lock the corresponding lock object. If the lock has already been set by another thread, the executing thread is blocked until the locking thread releases the lock by calling `unlock()`. If the lock object has not been set by another thread when calling `lock()`, the executing thread becomes owner of the lock without waiting.

The method `tryLock()` also tries to lock a lock object. If this is successful, the return value is `true`. If the lock object is already set by another thread, the return value is `false`; in contrast to `lock()`, the calling thread is not blocked in this case. For the method `tryLock()`, additional parameters can be specified to set a waiting time after which control is resumed also if the lock is not available, see `tryAcquire()` of the class `Semaphore`. The method `unlock()` releases a lock which has previously been set by the calling thread.

The class `ReentrantLock()` provides an implementation of the interface `Lock`. The constructors of this class

```
public ReentrantLock();  
public ReentrantLock(boolean fairness);
```

```
import java.util.concurrent.locks.*;
public class NewClass {
    private ReentrantLock lock = new ReentrantLock();
    //...
    public void method() {
        lock.lock();
        try {
            //...
        } finally { lock.unlock(); }
    }
}
```

Fig. 6.42 Illustration of the use of `ReentrantLock` objects.

allow the specification of an additional fairness parameter `fairness`. If this is set to `true`, the thread with the longest waiting time can access the lock object if several threads are waiting concurrently for the same lock object. If the fairness parameter is not used, no specific access order can be assumed. Using the fairness parameter can lead to an additional management overhead and hence to a reduced throughput. A typical usage of the class `ReentrantLock` is illustrated in Fig. 6.42.

6.2.6.4 Signal mechanism

The interface `Condition` from the package `java.util.concurrent.lock` defines a signal mechanism with condition variables which allows a thread to wait for a specific condition. The occurrence of this condition is shown by a signal of another thread, similar to the functionality of condition variables in Pthreads, see Sect. 6.1.3, page 300. A condition variable is always bound to a lock object, see interface `Lock`. A condition variable to a lock object can be created by calling the method

```
Condition newCondition().
```

This method is provided by all classes which implement the interface `Lock`. The condition variable returned by the method is bound to the lock object for which the method `newCondition()` has been called. For condition variables, the following methods are available:

```
void await();
void await(long time, TimeUnit unit);
void signal();
void signalAll();
```

```

import java.util.concurrent.locks.*;
public class BoundedBufferCondition {
    private Lock lock = new ReentrantLock();
    private Condition notFull = lock.newCondition();
    private Condition notEmpty = lock.newCondition();
    private Object[] items = new Object[100];
    private int putptr=0, takeptr=0, count=0;
    public void put (Object x)
        throws InterruptedException {
        lock.lock();
        try {
            while (count == items.length)
                notFull.await();
            items[putptr] = x;
            putptr = (putptr +1) % items.length;
            ++count;
            notEmpty.signal();
        } finally { lock.unlock(); }
    }
    public Object take()
        throws InterruptedException {
        lock.lock();
        try {
            while (count == 0)
                notEmpty.await();
            Object x = items[takeptr];
            takeptr = (takeptr +1) % items.length;
            --count;
            notFull.signal();
            return x;
        } finally {lock.unlock();}
    }
}

```

Fig. 6.43 Realization of a buffer mechanism by using condition variables.

The method `await()` blocks the executing thread until it is woken up by another thread by `signal()`. Before blocking, the executing thread releases the lock object as an atomic operation. Thus, the executing thread has to be the owner of the lock object before calling `await()`. After the blocked thread is woken up again by a `signal()` of another thread, it first must try to set the lock object again. Only after this is successful, the thread can proceed with its computations.

There is a variant of `await()` which allows the additional specification of a waiting time. If this variant is used, the calling thread is woken up after the time interval has elapsed, and if no `signal()` of another thread has arrived in the meantime. By calling `signal()`, a thread can wake up another thread which is waiting for a condition variable. By calling `signalAll()`, *all* waiting threads of the condition variable are woken up. The use of condition variables for the realization of a buffer

mechanism is illustrated in Fig. 6.43. The condition variables are used in a similar way as the semaphore objects in Fig. 6.41.

6.2.6.5 Atomic Operations

The package `java.util.concurrent.atomic` provides atomic operations for simple data types, allowing a lock-free access to single variables. An example is the class `AtomicInteger` which comprises the following methods

```
boolean compareAndSet (int expect, int update);  
int getAndIncrement();
```

The first method sets the value of the variable to the value `update`, if the variable previously had the value `expect`. In this case, the return value is `true`. If the variable has not the expected value, the return value is `false`; no operation is performed. The operation is performed *atomically*, i.e., during the execution, the operation cannot be interrupted.

The second method increments the value of the variable atomically and returns the previous value of the variable as a result. The class `AtomicInteger` provides plenty of similar methods.

6.2.6.6 Task-based execution of programs

The package `java.util.concurrent` also provides a mechanism for a task-based formulation of programs. A task is a sequence of operations of the program which can be executed by an arbitrary thread. The execution of tasks is supported by the interface `Executor`:

```
public interface Executor {  
    void execute (Runnable command);  
}
```

where `command` is the task which is brought to execution by calling `execute()`. A simple implementation of the method `execute()` might merely activate the method `command.run()` in the current thread. More sophisticated implementations may queue `command` for execution by one of a set of threads. For multicore processors, several threads are typically available for the execution of tasks. These threads can be combined in a thread pool where each thread of the pool can execute an arbitrary task.

Compared to the execution of each task by a separate thread, the use of task pools typically leads to a smaller management overhead, particularly if the tasks consist of only a few operations. For the organization of thread pools, the class `Executors` can be used. This class provides methods for the generation and management of thread pools. Important methods are:


```
static ExecutorService newFixedThreadPool(int n);  
static ExecutorService newCachedThreadPool();  
static ExecutorService newSingleThreadExecutor();
```

The first method generates a thread pool which creates new threads when executing tasks until the maximum number *n* of threads has been reached. The second method generates a thread pool for which the number of threads is dynamically adapted to the number of tasks to be executed. Threads are terminated, if they are not used for a specific amount of time (60 s). The third method generates a single thread which executes a set of tasks. To support the execution of task-based programs the interface `ExecutorService` is provided. This interface inherits from the interface `Executor` and comprises methods for the termination of thread pools. The most important methods are

```
void shutdown();  
List<Runnable> shutdownNow();
```

The method `shutdown()` has the effect, that the thread pool does not accept further tasks for execution. Tasks which have already been submitted are still executed before the shutdown. In contrast, the method `shutdownNow()` additionally stops the tasks which are currently in execution; the execution of waiting tasks is not started. The set of waiting tasks is provided in form of a list as return value. The class `ThreadPoolExecutor` is an implementation of the interface `ExecutorService`.

Figure 6.44 illustrates the use of a thread pool for the realization of a web server which waits for connection requests of clients at a `ServerSocket` object. If a client request arrives, it is computed as a separate task by submitting this task with `execute()` to a thread pool. Each task is generated as a `Runnable` object. The operation `handleRequest()` to be executed for the request is specified as `run()` method. The maximum size of the thread pool is set to 10.

6.3 OpenMP

OpenMP is a portable standard for the programming of shared memory systems. The OpenMP API (application program interface) provides a collection of compiler directives, library routines, and environmental variables. The compiler directives can be used to extend the sequential languages Fortran, C, and C++ with single program multiple data (SPMD) constructs, tasking constructs, work-sharing constructs, and synchronization constructs. The use of shared and private data is supported. The library routines and the environmental variable control the runtime system.

The OpenMP standard was designed in 1997 and is owned and maintained by the OpenMP Architecture Review Board (ARB). Since then many vendors have

Fig. 6.44 Draft of a task-based web server.

```
import java.io.IOException;
import java.net.*;
import java.util.concurrent.*;

public class TaskWebServer {
    static class RunTask implements Runnable {
        private Socket myconnection;
        public RunTask (Socket connection) {
            myconnection = connection;
        }
        public void run() {
            // handleRequest(myconnection);
        }
    }
    public static void main (String[] args)
        throws IOException {
        ServerSocket s = new ServerSocket(80);
        ExecutorService pool =
            Executors.newFixedThreadPool(10);
        try {
            while (true) {
                Socket connection = s.accept();
                Runnable task = new RunTask(connection)
                pool.execute(task);
            }
        } catch (IOException ex) {
            pool.shutdown();
        }
    }
}
```

included the OpenMP standard in their compilers. Currently, most compilers support the Version 2.5 from May 2005 [149]. The most recent update is Version 3.1 from July 2011 [150], which is, e.g., supported by the gcc4.7 compiler and the Intel Fortran and C/C++ compilers. Information about OpenMP and the standard definition can be found at the following website: <http://www.openmp.org>.

The programming model of OpenMP is based on cooperating **threads** running simultaneously on multiple processors or cores. Threads are created and destroyed in a **fork-join** pattern. The execution of an OpenMP program begins with a single thread, the initial thread, which executes the program sequentially until a first **parallel** construct is encountered. At the parallel construct, the initial thread creates a team of threads consisting of a certain number of new threads and the initial thread itself. The initial thread becomes the master thread of the team. This fork operation is performed implicitly. The program code inside the parallel construct is called a **parallel region** and is executed in parallel by all threads of the team. The parallel execution mode can be an SPMD style; but an assignment of different tasks to different threads is also possible. OpenMP provides directives for different execution modes, which

will be described below. At the end of a parallel region there is an implicit barrier synchronization, and only the master thread continues its execution after this region (implicit join operation). Parallel regions can be nested and each thread encountering a parallel construct creates a team of threads as described above.

The memory model of OpenMP distinguishes between shared memory and private memory. All OpenMP threads of a program have access to the same shared memory. To avoid conflicts, race conditions, or deadlocks, synchronization mechanisms have to be employed, for which the OpenMP standard provides appropriate library routines. In addition to shared variables, the threads can also use private variables in the *threadprivate* memory, which cannot be accessed by other threads.

An OpenMP program needs to include the header file `<omp.h>`. The compilation with appropriate options translates the OpenMP source code into multithreaded code. This is supported by several compilers. The Version 4.2 of GCC and newer versions support OpenMP; the option `-fopenmp` has to be used. Intel's C++ compiler Version 8 and newer versions also support the OpenMP standard and provide additional Intel-specific directives. A compiler supporting OpenMP defines the variable `_OPENMP` if the OpenMP option is activated.

An OpenMP program can also be compiled into sequential code by a translation without the OpenMP option. The translation ignores all OpenMP directives. However, for the translation into correct sequential code special care has to be taken for some OpenMP runtime functions. The variable `_OPENMP` can be used to control the translation into sequential or parallel code.

6.3.1 Compiler directives

In OpenMP, parallelism is controlled by compiler directives. For C and C++, OpenMP directives are specified with the `#pragma` mechanism of the C and C++ standards. The general form of an OpenMP directive is

```
#pragma omp directive [clauses [ ] ...]
```

written in a single line. The clauses are optional and are different for different directives. Clauses are used to influence the behavior of a directive. In C and C++, the directives are case sensitive and apply only to the next code line or to the block of code (written within brackets `{` and `}`) immediately following the directive.

6.3.1.1 Parallel region

The most important directive is the `parallel` construct mentioned before with syntax

```
#pragma omp parallel [clause [clause] ... ]
{ // structured block ... }
```

The `parallel` construct is used to specify a program part that should be executed in parallel. Such a program part is called a *parallel region*. A team of threads is created to execute the parallel region in parallel. Each thread of the team is assigned a unique thread number, starting from zero for the master thread up to the number of threads minus one. The parallel construct ensures the creation of the team but does not distribute the work of the parallel region among the threads of the team. If there is no further explicit distribution of work (which can be done by other directives), all threads of the team execute the same code on possibly different data in an SPMD mode. One usual way to execute on different data is to employ the thread number also called *thread id*. The user-level library routine

```
int omp_get_thread_num()
```

returns the thread id of the calling thread as integer value. The number of threads remains unchanged during the execution of one parallel region but may be different for another parallel region. The number of threads can be set with the clause

```
num_threads(expression)
```

The user-level library routine

```
int omp_get_num_threads()
```

returns the number of threads in the current team as integer value, which can be used in the code for SPMD computations. At the end of a parallel region there is an implicit barrier synchronization and the master thread is the only thread which continues the execution of the subsequent program code.

The clauses of a parallel directive include clauses which specify whether data will be private for each thread or shared among the threads executing the parallel region. Private variables of the threads of a parallel region are specified by the `private` clause with syntax

```
private(list_of_variables)
```

where `list_of_variables` is an arbitrary list of variables declared before. The `private` clause has the effect that for each private variable a new version of the original variable with the same type and size is created in the memory of each thread belonging to the parallel region. The private copy can be accessed and modified only by the thread owning the private copy. Shared variables of the team of threads are specified by the `shared` clause with the syntax

```
shared(list_of_variables)
```

where `list_of_variables` is a list of variables declared before. The effect of this clause is that the threads of the team access and modify the same original variable in the shared memory. The default clause can be used to specify whether variables in a parallel region are *shared* or *private* by default. The clause

```
default(shared)
```

```

#include <stdio.h>
#include <omp.h>

int npoints, iam, np, mypoints;
double *x;

int main() {
    scanf("%d", &npoints);
    x = (double *) malloc(npoints * sizeof(double));
    initialize();
    #pragma omp parallel shared(x,npoints) private(iam,np,mypoints)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        mypoints = npoints / np;
        compute_subdomain(x, iam, mypoints);
    }
}

```

Fig. 6.45 OpenMP program with `parallel` construct.

causes all variables referenced in the construct to be shared except the private variables which are specified explicitly. The clause

`default(none)`

requires each variable in the construct to be specified explicitly as *shared* or *private*. The following example shows a first OpenMP program with a parallel region, in which multiple threads perform an SPMD computation on shared and private data.

Example: The program code in Fig. 6.45 uses a `parallel` construct for a parallel SPMD execution on an array `x`. The input values are read in the function `initialize()` by the master thread. Within the parallel region the variables `x` and `npoints` are specified as *shared* and the variables `iam`, `np` and `mypoints` are specified as *private*. All threads of the team of threads executing the parallel region store the number of threads in the variable `np` and their own thread id in the variable `iam`. The private variable `mypoints` is set to the number of points assigned to a thread. The function `compute_subdomain()` is executed by each thread of the team using its own private variables `iam` and `mypoints`. The actual computations are performed on the *shared* array `x`. □

A nesting of parallel regions by calling a `parallel` construct within a parallel region is possible. However, the default execution mode assigns only one thread to the team of the inner parallel region. The library function

`void omp_set_nested(int nested)`

with a parameter `nested` $\neq 0$ can be used to change the default execution mode to more than one thread for the inner region. The actual number of threads assigned to the inner region depends on the specific OpenMP implementation.

6.3.1.2 Parallel loops

OpenMP provides constructs which can be used within a parallel region to distribute the work across threads that already exist in the team of threads executing the parallel region. The loop construct causes a distribution of the iterates of a **parallel loop** and has the syntax

```
#pragma omp for [clause [clause] ... ]
for (i = lower_bound; i op upper_bound; incr_expr) {
    { // loop iterate ... }
}
```

The use of the `for` construct is restricted to loops which are parallel loops, in which the iterates of the loop are independent of each other and for which the total number of iterates is known in advance. The effect of the `for` construct is that the iterates of the loop are assigned to the threads of the parallel region and are executed in parallel. The index variable `i` should not be changed within the loop and is considered as private variable of the thread executing the corresponding iterate. The expression `lower_bound` and `upper_bound` is integer expressions, whose values should not be changed during the execution of the loop. The operator `op` is a boolean operator from the set `{<, <=, >, >=}`. The increment expression `incr_expr` can be of the form

```
++i, i++, --i, i--, i += incr, i -= incr,
i = i + incr, i = incr + i, i = i - incr,
```

with an integer expression `incr` that remains unchanged within the loop. The parallel loop of a `for` construct should not be finished with a `break` command. The parallel loop ends with an implicit synchronization of all threads executing the loop, and the program code following the parallel loop is only executed if all threads have finished the loop. The `nowait` clause given as clause of the `for` construct can be used to avoid this synchronization.

The specific distribution of iterates to threads is done by a scheduling strategy. OpenMP supports different scheduling strategies specified by the `schedule` parameters of the following list:

- `schedule(static, block_size)` specifies a static distribution of iterates to threads which assigns blocks of size `block_size` in a *round-robin* fashion to the threads available. When `block_size` is not given, blocks of almost equal size are formed and assigned to the threads in a blockwise distribution.
- `schedule(dynamic, block_size)` specifies a dynamic distribution of blocks to threads. A new block of size `block_size` is assigned to a thread as soon as the thread has finished the computation of the previously assigned block. When `block_size` is not provided, blocks of size one, i.e., consisting of only one iterate, are used.
- `schedule(guided, block_size)` specifies a dynamic scheduling of blocks with decreasing size. For the parameter value `block_size = 1`, the new block

assigned to a thread has a size which is the quotient of the number of iterates not assigned yet and the number of threads executing the parallel loop. For a parameter value `block_size = k > 1`, the size of the blocks is determined in the same way, but a block never contains fewer than k iterates (except for the last block which may contain fewer than k iterates). When no `block_size` is given, the blocks consist of one iterate each.

- `schedule(auto)` delegates the scheduling decision to the compiler and/or runtime system. Thus, any possible mapping of iterates to threads can be chosen.
- `schedule(runtime)` specifies a scheduling at runtime. At runtime the environmental variable `OMP_SCHEDULE`, which has to contain a character string describing one of the formats given above, is evaluated. Examples are

```
setenv OMP_SCHEDULE "dynamic, 4"
setenv OMP_SCHEDULE "guided"
```

When the variable `OMP_SCHEDULE` is not specified, the scheduling used depends on the specific implementation of the OpenMP library.

A `for` construct without any `schedule` parameter is executed according to a default scheduling method also depending on the specific implementation of the OpenMP library. The use of the `for` construct is illustrated with the following example coding a matrix multiplication.

Example: The code fragment in Fig. 6.46 shows a multiplication of 100×100 matrix `MA` with a 100×100 matrix `MB` resulting in a matrix `MC` of the same dimension. The parallel region specifies `MA`, `MB`, and `MC` as shared variables and the indices `row`, `col`, `i` as private. The two parallel loops use `static` scheduling with blocks of `row`. The first parallel loop initializes the result matrix `MC` with 0. The second parallel loop performs the matrix multiplication in a nested `for` loop. The `for` construct applies to the first `for` loop with iteration variable `row`, and thus the iterates of the parallel loop are the nested loops of the iteration variables `col` and `i`. The static scheduling leads to a row-block-wise computation of the matrix `MC`. The first loop ends with an implicit synchronization. Since it is not clear that the first and the second parallel loops have exactly the same assignment of iterates to threads, a `nowait` clause should be avoided to guarantee that the initialization is finished before the multiplication starts. □

The nesting of the `for`-construct within the same `parallel` construct is not allowed. The nesting of parallel loops can be achieved by nesting `parallel` constructs, so that each `parallel` construct contains exactly one `for` construct. This is illustrated in the following example.

Example: The program code in Fig. 6.47 shows a modified version of the matrix multiplication in the last example. Again, the `for`-construct applies to the `for` loop with the iteration index `row`. The iterates of this parallel loop start with another `parallel` construct which contains a second `for`-construct applying to the loop with iteration index `col`. This leads to a parallel computation, in which each entry

```

#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
    read_input(MA, MB);
    #pragma omp parallel shared(MA,MB,MC,size) private(row,col,i)
    {
        #pragma omp for schedule(static)
        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                MC[row][col] = 0.0;
        }
        #pragma omp for schedule(static)
        for (row = 0; row < size; row++) {
            for (col = 0; col < size; col++)
                for (i = 0; i < size; i++)
                    MC[row][col] += MA[row][i] * MB[i][col];
        }
    }
    write_output(MC);
}

```

Fig. 6.46 OpenMP program for a parallel matrix multiplication using a parallel region with two inner `for` constructs.

of `MC` can be computed by a different thread. There is no need for a synchronization between initialization and computation. □

The OpenMP program in Fig. 6.47 implements the same parallelism as the Pthreads program for matrix multiplication in Fig. 6.1, see page 293. A difference between the two programs is that the Pthreads program starts the threads explicitly. The thread creation in the OpenMP program is done implicitly by the OpenMP library which deals with the implementation of the nested loop and guarantees the correct execution. Another difference is that there is a limitation for the number of threads in the Pthreads program. The matrix size 8×8 in the Pthreads program from Fig. 6.1 leads to a correct program. A matrix size 100×100 , however, would lead to the start of 10000 threads, which is too large for most Pthreads implementation. There is no such limitation in the OpenMP program.

6.3.1.3 Noniterative Work-Sharing Constructs

The OpenMP library provides the `sections` construct to distribute noniterative tasks to threads. Within the `sections` construct different code blocks are indicated by the `section` construct as tasks to be distributed. The syntax for the use of a `sections` construct is the following.


```

#include <omp.h>

double MA[100][100], MB[100][100], MC[100][100];
int i, row, col, size = 100;

int main() {
    read_input(MA, MB);
    #pragma omp parallel private(row,col,i)
    {
        #pragma omp for schedule(static)
        for (row = 0; row < size; row++) {
            #pragma omp parallel shared(MA, MB, MC, size)
            {
                #pragma omp for schedule(static)
                for (col = 0; col < size; col++) {
                    MC[row][col] = 0.0;
                    for (i = 0; i < size; i++)
                        MC[row][col] += MA[row][i] * MB[i][col];
                }
            }
        }
    }
    write_output(MC);
}

```

Fig. 6.47 OpenMP programm for a parallel matrix multiplication with nested parallel loops.

```

#pragma omp sections [clause [clause] ... ]
{
    [#pragma omp section]
    { // structured block ... }
    [#pragma omp section]
    { // structured block ... }
    :
}

```

The section constructs denote structured blocks which are independent of each other and can be executed in parallel by different threads. Each structured block starts with `#pragma omp section` which can be omitted for the first block. The sections construct ends with an implicit synchronization unless a `nowait` clause is specified.

6.3.1.4 Single execution

The `single` construct is used to specify that a specific structured block is executed by only one thread of the team, which is not necessarily the master thread. This can be useful for tasks like control messages during a parallel execution. The `single` construct has the syntax

```
#pragma omp single [Parameter [Parameter] ... ]
{ // structured block ... }
```

and can be used within a parallel region. The `single` construct also ends with an implicit synchronization unless a `nowait` clause is specified. The execution of a structured block within a parallel region by the master thread only is specified by

```
#pragma omp master
{ // structured block ... }
```

All other threads ignore the construct. There is no implicit synchronization of the master threads and the other threads of the team.

6.3.1.5 Syntactic abbreviations

OpenMP provides abbreviated syntax for parallel regions containing only one `for` construct or only one `sections` construct. A parallel region with one `for` construct can be specified as

```
#pragma omp parallel for [clause [clause] ... ]
  for (i = lower_bound; i op upper_bound; incr_expr) {
    { // loop body ... }
  }
```

All clauses of the `parallel` construct or the `for` construct can be used. A parallel region with only one `sections` construct can be specified as

```
#pragma omp parallel sections [clause [clause] ... ]
{
  [#pragma omp section]
  { // structured block ... }
  [#pragma omp section]
  { // structured block ... }
  :
}
}
```

6.3.2 *Execution environment routines*

The OpenMP library provides several execution environment routines that can be used to query and control the parallel execution environment. We present a few of them. The function

```
void omp_set_dynamic (int dynamic_threads)
```

can be used to set a dynamic adjustment of the number of threads by the runtime system and is called outside a parallel region. A parameter value `dynamic_threads` $\neq 0$ allows the dynamic adjustment of the number of threads for the subsequent parallel region. However, the number of threads within the same parallel region remains constant. The parameter value `dynamic_threads` = 0 disables the dynamic adjustment of the number of threads. The default case depends on the specific OpenMP implementation. The routine

```
int omp_get_dynamic (void)
```

returns information about the current status of the dynamic adjustment. The return value 0 denotes that no dynamic adjustment is set; a return value $\neq 0$ denotes that the dynamic adjustment is set. The number of threads can be set with the routine

```
void omp_set_num_threads (int num_threads)
```

which has to be called outside a parallel region and influences the number of threads in the subsequent parallel region (without a `num_threads` clause). The effect of this routine depends on the status of the dynamic adjustment. If the dynamic adjustment is set, the value of the parameter `num_threads` is the maximum number of threads to be used. If the dynamic adjustment is not set, the value of `num_threads` denotes the number of threads to be used in the subsequent parallel region. The routine

```
void omp_set_nested (int nested)
```

influences the number of threads in nested parallel regions. The parameter value `nested` = 0 means that the execution of the inner parallel region is executed by one thread sequentially. This is the default. A parameter value `nested` $\neq 0$ allows a nested parallel execution and the runtime system can use more than one thread for the inner parallel region. The actual behavior depends on the implementation. The routine

```
int omp_get_nested (void)
```

returns the current status of the nesting strategy for nested parallel regions.

6.3.3 *Coordination and synchronization of threads*

A parallel region is executed by multiple threads accessing the same shared data, so that there is need for synchronization in order to protect critical regions or avoid

race condition, see also Chap. 3. OpenMP offers several constructs which can be used for synchronization and coordination of threads within a parallel region. The `critical` construct specifies a **critical region** which can be executed only by a single thread at a time. The syntax is

```
#pragma omp critical [(name)]
    structured block
```

An optional name can be used to identify a specific critical region. When a thread encounters a `critical` construct, it waits until no other thread executes a critical region of the same name `name` and then executes the code of the critical region. Unnamed critical regions are considered to be one critical region with the same unspecified name. The `barrier` construct with syntax

```
#pragma omp barrier
```

can be used to synchronize the threads at a certain point of execution. At such an explicit `barrier` construct all threads wait until all other threads of the team have reached the barrier and only then they continue the execution of the subsequent program code. The `atomic` construct can be used to specify that a single assignment statement is an **atomic operation**. The syntax is

```
#pragma omp atomic
statement
```

and can contain statements of the form

```
x binop= E,
x++, ++x, x--, --x,
```

with an arbitrary variable `x`, a scalar expression `E` not containing `x`, and a binary operator `binop` $\in \{+, -, *, /, \&, ^, |, <<, >>\}$. The `atomic` construct ensures that the storage location `x` addressed in the statement belonging to the construct is updated atomically, which means that the load and store operations for `x` are atomic but not the evaluation of the expression `E`. No interruption is allowed between the load and the store operation for variable `x`. However, the `atomic` construct does not enforce exclusive access to `x` with respect to a critical region specified by a `critical` construct. An advantage of the `atomic` construct over the `critical` construct is that also parts of an array variable can be specified as being atomically updated. The use of a `critical` construct would protect the entire array.

Example: The following example shows an atomic update of a single array element `a[index[i]] += b`.

```
extern float a[], *p=a, b; int index[];
#pragma omp atomic
a[index[i]] += b;
#pragma omp atomic
p[i] -= 1.0;
```

□

```
#pragma omp parallel for reduction (+: a,y) reduction (||: am)
for (i=0; i<n; i++) {
    a += b[i];
    y = sum (y, c[i]);
    am = am || b[i] == c[i];
}
```

Fig. 6.48 Program fragment for the use of the `reduction` clause.

A typical calculation which needs to be synchronized is a **global reduction** operation performed in parallel by the threads of a team. For this kind of calculation OpenMP provides the `reduction` clause, which can be used for `parallel`, `sections` and `for` constructs. The syntax of the clause is

`reduction (op: list)`

where `op` $\in \{+, -, *, \&, ^, |, \&\&, ||\}$ is a reduction operator to be applied and `list` is a list of reduction variables which have to be declared as shared. For each of the variables in `list`, a private copy is created for each thread of the team. The private copies are initialized with the neutral element of the operation `op` and can be updated by the owning thread. At the end of the region for which the `reduction` clause is specified, the local values of the reduction variables are combined according to the operator `op` and the result of the reduction is written into the original shared variable. The OpenMP compiler creates efficient code for executing the global reduction operation. No additional synchronization, such as the `critical` construct, has to be used to guarantee a correct result of the reduction. The following example illustrates the accumulation of values.

Example: Figure 6.48 shows the accumulation of values in a `for` construct with the results written into the variables `a`, `y` and `am`. Local reduction operations are performed by the threads of the team executing the `for` construct using private copies of `a`, `y` and `am` for the local results. It is possible that a reduction operation is performed within a function, such as the function `sum` used for the accumulation onto `y`. At the end of the `for` loop, the values of the private copies of `a`, `y` and `am` are accumulated according to `+` or `||`, respectively, and the final values are written into the original shared variables `a`, `y` and `am`. □

The shared memory model of OpenMP might also require to coordinate the memory view of the threads. OpenMP provides the `flush` construct with the syntax

`#pragma omp flush [(list)]`

to produce a consistent view of the memory, where `list` is a list of variables whose values should be made consistent. For pointers in the list `list` only the pointer value is updated. If no list is given, all variables are updated. An inconsistent view can occur since modern computers provide memory hierarchies. Updates are usually done in the faster memory parts, like registers or caches, which are not immediately

```

#pragma omp parallel private (iam, neighbor) shared (work, sync)
{
    iam = omp_get_thread_num();
    sync[iam] = 0;
    #pragma omp barrier
    work[iam] = do_work();
    #pragma omp flush (work)
    sync[iam] = 1;
    #pragma omp flush (sync)
    neighbor = (iam != 0) ? (iam - 1) : (omp_get_num_threads() - 1);
    while (sync[neighbor] == 0) {
        #pragma omp flush (sync)
        { }
    }
    combine (work[iam], work[neighbor]);
}

```

Fig. 6.49 Program fragment for the use of the `flush` construct.

visible to all threads. OpenMP has a specific relaxed-consistency shared memory in which updated values are written back later. But to make sure at a specific program point that a value written by one thread is actually read by another thread, the `flush` construct has to be used. It should be noted that no synchronization is provided if several threads execute the `flush` construct.

Example: Figure 6.49 shows an example adopted from the OpenMP specification [148]. Two threads i ($i = 0, 1$) compute `work[i]` of array `work` which is written back to memory by the `flush` construct. The following update on array `sync[iam]` indicates that the computation of `work[iam]` is ready and written back to memory. The array `sync` is also written back by a second `flush` construct. In the `while` loop a thread waits for the other thread to have updated its part of `sync`. The array `work` is then used in the function `combine()` only after both threads have updated their elements of `work`. □

Besides the explicit `flush` construct there is an implicit flush at several points of the program code, which are:

- a `barrier` construct,
- entry to and exit from a `critical` region,
- at the end of a `parallel` region,
- at the end of a `for`, `sections` or `single` construct without `nowait` clause,
- entry and exit of `lock` routines (which will be introduced below).

Fig. 6.50 Program fragment illustrating the use of nestable lock variables.

```
#include <omp.h>

typedef struct {
    int a, b;
    omp_nest_lock_t l;
} pair;

void incr_a (pair *p, int a) {
    p->a += a;
}

void incr_b (pair *p, int b) {
    omp_set_nest_lock (&p->l);
    p->b += b;
    omp_unset_nest_lock (&p->l);
}

void incr_pair (pair *p, int a, int b) {
    omp_set_nest_lock (&p->l);
    incr_a (p, a);
    incr_b (p, b);
    omp_unset_nest_lock (&p->l);
}

void f (pair *p) {
    extern int work1(), work2(), work3();
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair (p, work1(), work2());
        #pragma omp section
        incr_b (p, work3());
    }
}
```

6.3.3.1 Locking mechanism

The OpenMP runtime system also provides runtime library functions for a synchronization of threads with the **locking mechanism**. The locking mechanism has been described in Sect. 4.3 and in this Sect. 6 for Pthreads and Java threads. The specific locking mechanism of the OpenMP library provides two kinds of lock variables on which the locking runtime routines operate. *Simple locks* of type `omp_lock_t` can be locked only once. *Nestable locks* of type `omp_nest_lock_t` can be locked multiple times by the same thread. OpenMP lock variables should be accessed only by OpenMP locking routines. A lock variable is initialized by one of the following initialization routines

```
void omp_init_lock (omp_lock_t *lock);
void omp_init_nest_lock (omp_nest_lock_t *lock);
```

for simple and nestable locks, respectively. A lock variable is removed with the routines

```
void omp_destroy_lock (omp_lock_t *lock);
void omp_destroy_nest_lock (omp_nest_lock_t *lock);
```

An initialized lock variable can be in the states *locked* or *unlocked*. At the beginning, the lock variable is in the state *unlocked*. A lock variable can be used for the synchronization of threads by locking and unlocking. To lock a lock variable the functions

```
void omp_set_lock (omp_lock_t *lock)
void omp_set_nest_lock (omp_nest_lock_t *lock)
```

are provided. If the lock variable is available, the thread calling the lock routine locks the variable. Otherwise, the calling thread blocks. A simple lock is available when no other thread has locked the variable before without unlocking it. A nestable lock variable is available when no other thread has locked the variable without unlocking it or when the calling thread has locked the variable, i.e., multiple locks for one nestable variable by the same thread are possible counted by an internal counter. When a thread uses a lock routine to lock a variable successfully, this thread is said to *own* the lock variable. A thread owning a lock variable can unlock this variable with the routines

```
void omp_unset_lock (omp_lock_t *lock)
void omp_unset_nest_lock (omp_nest_lock_t *lock)
```

For a nestable lock, the routine `omp_unset_nest_lock ()` decrements the internal counter of the lock. If the counter has the value 0 afterwards, the lock variable is in the state *unlocked*. The locking of a lock variable without a possible blocking of the calling thread can be performed by one of the routines

```
void omp_test_lock (omp_lock_t *lock)
void omp_test_nest_lock (omp_nest_lock_t *lock)
```

for simple and nestable lock variables, respectively. When the lock is available, the routines lock the variable or increment the internal counter and return a result value $\neq 1$. When the lock is not available, the `test` routine returns 0 and the calling thread is not blocked.

Example: Figure 6.50 illustrates the use of nestable lock variables, see [148]. A data structure `pair` consist of two integers `a` and `b` and a nestable lock variable `l` which is used to synchronize the updates of `a`, `b` or the entire `pair`. It is assumed that the lock variable `l` has been initialized before calling `f()`. The increment function `incr_a()` for incrementing `a`, `incr_b()` for incrementing `b`, and `incr_pair()` for incrementing both integer variables are given. The function `incr_a()` is only called from `incr_pair()` and does not need an additional locking. The functions `incr_b()` and `incr_pair()` are protected by the lock since they can be called concurrently. □

6.4 Exercises for Chapter 6

Exercise 6.1. Modify the matrix multiplication program from Fig. 6.1 on page 293, so that a fixed number of threads is used for the multiplication of matrices of arbitrary size. For the modification, let each thread compute the number of rows of the result matrix instead of a single entry. Compute the rows that each thread must compute, such that each thread has about the same number of rows to compute. Is there any synchronization required in the program?

Exercise 6.2. Use the task pool implementation from Sect. 6.1.6 on page 306 to implement a parallel matrix multiplication. To do so, use the function `thread_mult()` from Fig. 6.1 to define a task as the computation of one entry of the result matrix and modify the function if necessary, so that it fits to the requirements of the task pool. Modify the main program, so that all tasks are generated and inserted into the task pool before the threads to perform the computations are started. Measure the resulting execution time for different numbers of threads and different matrix sizes and compare the execution time with the execution time of the implementation of the last exercise.

Exercise 6.3. Consider the read/write lock mechanism in Fig. 6.5. The implementation given does not provide operations that are equivalent to the function `pthread_mutex_trylock()`. Extend the implementation from Fig. 6.5 by specifying functions `rw_lock_rtrylock()` and `rw_lock_wtrylock()` which return `EBUSY` if the requested read or write permit cannot be granted.

Exercise 6.4. Consider the read/write lock mechanism in Fig. 6.5. The implementation given favors read request over write requests in the sense that a thread will get a write permit only if no other threads request a read permit, but read permits are given without waiting also in the presence of other read permits. Change the implementation such that write permits have priority, i.e., as soon as a write permit arrives, no more read permits are granted until the write permit has been granted and the corresponding write operation is finished. To test the new implementation write a program which starts three threads, two read threads and one write thread. The first read-thread requests five read permits one after another. As soon as it gets the read permits it prints a control message and waits for 2 s (use `sleep(2)`) before requesting the next read permit. The second read thread does the same except that it only waits 1 s after the first read permit and 2 s otherwise. The write thread first waits 5 s and then requests a write permit and prints a control message after it has obtained the write permit; then the write permit is released again immediately.

Exercise 6.5. An read/write lock mechanism allows multiple readers to access a data structure concurrently, but only a single writer is allowed to access the data structures at a time. We have seen a simple implementation of r/w-locks in Pthreads in Fig. 6.5. Transfer this implementation to Java threads by writing a new class

RWlock with entries `num_r` and `num_w` to count the current number of read and write permits given. The class `RWlock` provides methods similar to the functions in Fig. 6.5 to request or release a read or write permit.

Exercise 6.6. Consider the pipelining programming pattern and its Pthreads implementation in Sect. 6.1.7. In the example given, each pipeline stage adds 1 to the integer value received from the predecessor stage. Modify the example such that pipeline stage i adds the value i to the value received from the predecessor. In the modification, there should still be only one function `pipe_stage()` expressing the computations of a pipeline stage. This function must receive an appropriate parameter for the modification.

Exercise 6.7. Use the task pool implementation from Sect. 6.1.6 to define a parallel loop pattern. The loop body should be specified as function with the loop variable as parameter. The iteration space of the parallel loop is defined as the set of all values that the loop variable can have. To execute a parallel loop, all possible indices are stored in a parallel data structure similar to a task pool which can be accessed by all threads. For the access, a suitable synchronization must be used.

- (a) Modify the task pool implementation accordingly, such that functions for the definition of a parallel loop and for retrieving an iteration from the parallel loop are provided. The thread function should also be provided.
- (b) The parallel loop pattern from (a) performs a dynamic load balancing since a thread can retrieve the next iteration as soon as its current iteration is finished. Modify this operation, such that a thread retrieves a chunk of iterations instead of a single operation to reduce the overhead of load balancing for fine-grained iterations.
- (c) Include guided self-scheduling (GSS) in your parallel loop pattern. GSS adapts the number of iterations retrieved by a thread to the total number of iterations that are still available. if n threads are used and there are R_i remaining iterations, the next thread retrieves

$$x_i = \lceil \frac{R_i}{n} \rceil$$

iterations. For the next retrieval, $R_{i+1} = R_i - x_i$ iterations remain. R_1 is the initial number of iterations to be executed.

- (d) Use the parallel loop pattern to express the computation of a matrix multiplication where the computation of each matrix entry can be expressed as an iteration of a parallel loop. Measure the resulting execution time for different matrix sizes. Compare the execution time for the two load balancing schemes (standard and GSS) implemented.

Exercise 6.8. Consider the client-server pattern and its Pthreads implementation in Sect. 6.1.8. Extend the implementation given in this section by allowing a cancellation with deferred characteristics. To be cancellation safe, mutex variables that have been locked must be released again by an appropriate cleanup handler. When a cancellation occurs, allocated memory space should also be released. In the server

function `tty_server_routine()`, the variable `running` should be reset when a cancellation occurs. Note that this may create a concurrent access. If a cancellation request arrives during the execution of a synchronous request of a client, the client thread should be informed that a cancellation has occurred. For a cancellation in the function `client_routine()`, the counter `client_threads` should be kept consistent.

Exercise 6.9. Consider the taskpool pattern and its implementation in Pthreads in Section 6.1.6. Implement a Java class `TaskPool` with the same functionality. The task pool should accept each object of a class which implements the interface `Runnable` as task. The tasks should be stored in an array `final Runnable tasks[]`. A constructor `TaskPool(int p, int n)` should be implemented which allocates a task array of size `n` and creates `p` threads which access the task pool. The methods `run()` and `insert(Runnable w)` should be implemented according to the Pthreads functions `tpool_thread()` and `tpool_insert()` from Fig. 6.7. Additionally, a method `terminate()` should be provided to terminate the threads that have been started in the constructor. For each access to the task pool, a thread should check whether a termination request has been set.

Exercise 6.10. Transfer the pipelining pattern from Sect. 6.1.7 for which Figs. 6.8–6.11 give an implementation in Pthreads to Java. For the Java implementation, define classes for a pipeline stage as well as for the entire pipeline which provide the appropriate method to perform the computation of a pipeline stage, to send data into the pipeline, and to retrieve a result from the last stage of the pipeline.

Exercise 6.11. Transfer the client-server pattern for which Figs. 6.13–6.16 give a Pthreads implementation to Java threads. Define classes to store a request and for the server implementation explain the synchronizations performed and give reasons that no deadlock can occur.

Exercise 6.12. Consider the following OpenMP program piece:

```
int x=0;
int y=0;

void foo1() {
#pragma omp critical (x)
    { foo2(); x+=1; }
}
void foo2() {
#pragma omp critical(y)
    { y+=1; }
}
void foo3() {
#pragma omp critical(y)
    { y-=1; foo4(); }
```

```
}  
void foo4() {  
    #pragma omp critical(x)  
        { x-=1; }  
}  
int main(int argc, char **argv) {  
    int x;  
    #pragma omp parallel private(i) {  
        for (i=0; i<10; i++)  
            { foo1(), foo3(); }  
    }  
    printf(''\%d \%d \n'', x,y )  
}
```

We assume that two threads execute this piece of code on two cores of a multicore processors. Can a deadlock situation occur? If so, describe the execution order which leads to the deadlock. If not, give reasons why a deadlock is not possible.