

Chapter 4. IPC: synchronization

The decision about what code to run next is made by the operating system (i.e., its scheduler), or by the hardware interrupts that force the processor to jump to an associated interrupt routine. To the scheduler of the OS, all tasks are just “numbers” in scheduling queues; and interrupts “talk” to their own interrupt service routine only. So, scheduler and interrupts would be sufficient organizational structure in a system where all tasks just live next to each other, without need for cooperation. This, of course, is not sufficient for many applications. For example, an interrupt service routine collects measurements from a peripheral device, this data is processed by a dedicated control task, the results are sent out via another peripheral device to an actuator, and displayed for the user by still another task.

Hence, the need exists for *synchronization* of different tasks (What is the correct sequence and timing to execute the different tasks?), as well as for *data exchange* between them. Synchronization and data exchange are complementary concepts, because the usefulness of exchanged data often depends on the correct synchronization of all tasks involved in the exchange. Both concepts are collectively referred to as *Interprocess communication* (“IPC”).

The role of the operating system in matters of IPC is to offer a sufficiently rich set of IPC-supporting primitives. These should allow the tasks to engage in IPC without having to bother with the details of their implementation and with hardware dependence. This is not a minor achievement of the operating system developers, because making these IPC primitives safe and easy to use requires a lot of care and insight. In any case, the current state-of-the-art in operating systems’ IPC support is such that they still don’t offer much more than just *primitives*. Hence, programmers have to know how to apply these primitives appropriately when building software systems consisting of multiple concurrent tasks; this often remains a difficult because error-prone design and implementation job. Not in the least because no *one-size-fits-all* solution can exist for all application needs.

4.1. IPC terminology

The general *synchronization* and *data exchange* problems involve (at least) two tasks, which we will call the “sender” and the “receiver”. (These tasks are often also called “writer” and “reader”, or “producer” and “consumer”.) For *synchronization*, “sender” and “receiver” want to make sure they are both in (or *not* in) specified parts of their code at the same time. For *data exchange*, “sender” and “receiver” want to make sure they can exchange data efficiently, without having to know too much of each other (“decoupling”, Chapter 14), and according to several different *policies*, such as blocking/non-blocking, or with/without data loss.

Data exchange has a natural direction of flow, and, hence, the terminology “sender” and “receiver” is appropriate. Synchronization is often without natural order or direction of flow, and, hence, the terminology “sender” and “receiver” is less appropriate in this context, and “(IPC) client” might be a more appropriate because symmetric terminology. Anyway, the exact terminology doesn’t matter too much. Unless we want to be more specific, we will use the generic system calls `send()` and `receive()` to indicate the IPC primitives used by sender and receiver, respectively.

4.1.1. Blocking/Non-blocking

IPC primitives can have different effects on *task scheduling*:

- *Blocking*. When executing the `send()` part of the IPC, the sender task is blocked (i.e., non-available for scheduling) until the receiver has accepted the IPC in a `receive()` call. And similarly the other way around. If both the sender and the receiver block until *both of them* are in their `send()` and `receive()` commands, the IPC is called *synchronous*. (Other names are: *rendez-vous*, or *handshake*.) Synchronous IPC is the easiest to design with, and is very similar to building hardware systems.
- *Non-blocking (asynchronous)*. Sender and receiver are not blocked in their IPC commands. This means that there is incomplete synchronization: the sender doesn't know when the receiver will get its message, and the receiver cannot be sure the sender is still in the same state as when it sent the message.
- *Blocking with time out*. The tasks wait in their IPC commands for at most a specified maximum amount of time.
- *Conditional blocking*. The tasks block in their IPC commands only if a certain condition is fulfilled.

Of course, blocking primitives should be used with care in real-time sections of a software system.

4.1.2. Coupling

IPC primitives can use different degrees of *coupling*:

- *Named connection*: sender and receiver know about each other, and can *call each other by name*. That means that the sender fills in the unique identifier of the receiver in its `send()` command, and vice versa. This can set up a connection between both tasks, in a way very similar to the telephone system, where one has to dial the number of the person one wants to talk to.

The connection can be *one-to-one*, or *one-to-many* (i.e., the single sender sends to more than one receiver, such as for broadcasting to a set of named correspondents), or *many-to-one* (for example, many tasks send logging commands to an activity logging task), or *many-to-many* (for example, video conferencing).

- *Broadcast*: the sender sends its message to all “listeners” (without explicitly calling them by name) on the (sub-branch of the) *network* to which it is connected. The listeners receive the message if they want, without the sender knowing exactly which tasks have really used its message.
- *Blackboard*: while a broadcast is a message on a network-like medium (i.e., the message is not stored in the network for later use), a blackboard IPC *stores* the messages from different senders. So, receivers can look at them at any later time.
- *Object request broker (ORB)*: the previous types of IPC all imply a rather high level of *coupling* between sender and receiver, in the sense that they have to know explicitly the identity of their communication partner, of the network branch, or of the blackboard. The current trend towards more *distributed* and *dynamically reconfigurable* computer systems calls for more *loosely-coupled* forms of IPC. The ORB concept has been developed to cover these needs: a sender *component* registers its

interface with the ORB; interested receivers can ask the broker to forward their requests to an appropriate sender (“server”) component, without the need to know its identity, nor its address.

4.1.3. Buffering

IPC primitives can use different degrees of *buffering*, ranging from the case where the operating system stores and delivers all messages, to the case where the message is lost if the receiver is not ready to receive it.

Not all of the above-mentioned forms of IPC are equally appropriate for *real-time* use, because some imply too much and/or too indeterministic overhead for communication and resource allocation.

4.2. Race conditions and critical sections

Often, two or more tasks need access to the same data or device, for writing and/or reading. The origin of most problems with *resource sharing* (or *resource allocation*) in multi-tasking and multi-processor systems is the fact that operations on resources can usually not be performed *atomically*, i.e., as if they were executed as one single, non-interruptible instruction that takes zero time. Indeed, a task that interfaces with a resource can at any instant be pre-empted, and hence, when it gets re-scheduled again, it cannot just take for granted that the data it uses now is in the same state (or at least, a state that is consistent with the state) before the pre-emption. Consider the following situation:

```
data number_1;
data number_2;

task A
{  data A_number;

    A_number = read(number_1);
    A_number = A_number + 1;
    write(number_2,A_number);
}

task B
{  if ( read(number_1) == read(number_2) )
    do_something();
    else
    do_something_else();
}
}
```

task B takes different actions based on the (non-)equality of *number_1* and *number_2*. But task B can be pre-empted in its *if* statement by task A, exactly at the moment that task B has already read

`number_1`, but not yet `number_2`. This means that it has read `number_1` before the pre-emption, and `number_2` after the pre-emption, which violates the validity of the test.

The `if` statement is one example of a so-called *critical section*: it is critical to the validity of the code that the *access to the data* used in that statement (i.e., `number_1` and `number_2`) be executed *atomically*, i.e., un-interruptable by anything else. (Most) *machine code* instructions of a given processor execute atomically; but instructions in higher-level programming languages are usually translated into a sequence of many machine code instructions, such that atomicity cannot be guaranteed.

There are three generic types of critical sections:

- *Access to the same data from different tasks*, as illustrated by the example above.
- *Access to a service*. For example, allocation of a resource, execution of a “transaction” on a database. The service typically has to process a sequence of queries, and these have to succeed as a whole, or fail as a whole.
- *Access to procedure code*. Application tasks often run exactly the same code (for example the control algorithm in each of the joints of a robot), but on other data, and some parts of that code should be executed by one task at a time only.

Of course, many applications involve combinations of different resource sharing needs.

The problem in all above-mentioned examples of access to shared resources is often called a *race condition*: two or more tasks compete (“race”) against each other to get access to the shared resources. Some of these race conditions have been given a special name:

- *Deadlock*. `task A` has locked a resource and is blocked waiting for a resource that is locked by `task B`, while `task B` is blocked waiting for the resource that is locked by `task A`.
- *Livelock*. This situation is similar to the deadlock, with this difference: both tasks are not blocked but are actively trying to get the resource, in some form of *busy waiting*.
- *Starvation*. In this situation, some tasks never get the chance to allocate the resource they require, because other tasks always get priority.

The four conditions that have to be satisfied in order to (potentially!) give rise to a deadlock are:

1. Locks are only *released voluntarily* by tasks. So, a task that needs two locks might obtain the first lock, but block on the second one, so that it is not able anymore to voluntarily release the first lock.
2. Tasks can only get in a deadlock if they need *more than one lock*, and have to obtain them in a (non-atomic) *sequential* order.
3. The resources guarded by locks can only be *allocated to one single task*. (Or to a finite number of tasks.)
4. Tasks try to obtain locks that other tasks have already obtained, and these tasks form a *circular list*. For example, `task A` is waiting for `task B` to release a lock, `task B` is waiting for `task C` to release a lock, and `task C` is waiting for `task A`.

As soon as *one* of these four conditions is not satisfied, a deadlock can not occur. Moreover, these conditions are *not sufficient* for deadlocks to occur: they just describe the conditions under which it is *possible* to have deadlocks.

The literature contains many examples of deadlock *avoidance* and *prevention* algorithms. Deadlock avoidance makes sure that all four necessary conditions are never satisfied at the same time; deadlock prevention allows the possibility for a deadlock to occur, but makes sure that this possibility is never realized. Both kinds of algorithms, however, often require some form of “global” knowledge about the states of all tasks in the system. Hence, they are too indeterministic for real-time execution, and not suitable for *component-based* design (because the requirement for global knowledge is in contradiction with the *loose coupling* strived for in component systems (see Chapter 14).

There are some guaranteed deadlock avoidance algorithms, that are reasonably simple to implement. For example, a deadlock cannot occur if *all programs* always take locks in the same order. This requires a globally known and ordered lists of locks, and coding discipline from the programmers. Other prevention algorithms use some of the following approaches: only allow each task to hold one resource; pre-allocate resources; force release of a resource before a new request can be made; ordering all tasks and give them priority according to that order.

Race conditions occur on a single processor system because of its multi-tasking and interrupt functionalities. But they show up even more on *multi-processor systems*: even if one CPU is preventing the tasks that it runs from accessing a resource concurrently, a task on another CPU might interfere.

4.3. Signals

Signals are one of the IPC synchronization primitives used for *asynchronous notification*: one task fires a signal, which *can* cause other tasks to start doing things. The emphasis is on “asynchronous” and on “can”:

- *Asynchronous*: the tasks that react to signals are in a completely arbitrary state, unrelated with the signaling task. Their reaction to the signal also need not be instantaneous, or synchronized, with the signaling task. The task that sends the signal, and the tasks that use the signal, need not share any memory, as in the case of semaphores or mutexes. This makes signals about the only synchronization primitive that is straightforward to scale over a *network*.
- *Can*: the signaling task fires a signal, and continues with its job. Whether or not other tasks do something with its signal is not of its concerns. The operating system takes care of the delivery of the signal, and if nobody wants it, it is just lost.

In most operating systems, signals

- are *not queued*. A task’s signal handler has no means to detect whether it has been signaled more than once.
- *carry no data*.

- *have no deterministic delivery time or order.* A task that gets signaled is not necessarily scheduled immediately.
- *have no deterministic order.* A task that gets signaled multiple times has no way to find out in which temporal order the signals were sent.

So, these are reasons to avoid signals for *synchronization between two running tasks*, [BrinchHansen73]. In other words: *notification* in itself is not sufficient for *synchronization*. Synchronization needs two tasks that do something together, while taking notice of each other, and respecting each other's activities. Later sections of the text present IPC primitives that are better suited for synchronization than signals.

POSIX has standardized signals and their connection to threads. The OS offers a number of pre-defined signals (such as “kill”), and task can ask the operating system to connect a handler (i.e., a function) to a particular signal on its behalf. The handler is “registered”, using the system call `sigaction()`. The task also asks the OS to receive or block a specific subset of all available signals; this is its “signal mask”. Whenever a signal is received by the operating system, it executes the registered handlers of all tasks that have this signal in their mask. The task can also issue a `sigwait(signal)`, which makes it sleep until the `signal` is received; in this case, the signal handler is *not executed*. Anyway, signals are a bit difficult to work with, as illustrated by this quote from the `signal` man page:

For `sigwait` to work reliably, the signals being waited for must be blocked in all threads, not only in the calling thread, since otherwise the POSIX semantics for signal delivery do not guarantee that it's the thread doing the `sigwait` that will receive the signal. The best way to achieve this is block those signals before any threads are created, and never unblock them in the program other than by calling `sigwait`.

The masks are also set on a *per-thread* basis, but the signal handlers are shared between all threads in a process. Moreover, the implementation of signals tend to differ between operating systems, and the POSIX standard leaves room for interpretation of its specification. For example, it doesn't say anything about the *order* in which blocked threads must be woken up by signals. So, these are reasons why many developers don't use signals too much.

POSIX has a specification for so-called “real-time signals” too. Real-time signals are queued, they pass a 4-byte data value to their associated signal handler, and they are guaranteed to be delivered in numerical order, i.e., from lowest signal number to highest. For example, RTLinux implements POSIX real-time signals, and offers 32 different signal levels. (See the file `include/rtl_sched.h` in the RTLinux source tree.) And RTAI also offers a 32 bit unsigned integer for events, but in a little different way: the integer is used to allow signalling *multiple* events: each bit in the integer is an event, and a task can ask to be notified when a certain AND or OR combination of these bits becomes valid. (See the file `include/rtai_bits.h` in the RTAI source tree.)

4.4. Exceptions

Exceptions are signals that are sent (“raised”) *synchronously*, i.e., by the task that is currently running. (Recall that signals are *asynchronous*, in the sense that a task can receive a signal at any arbitrary moment in its lifetime.) Exceptions are, roughly speaking, a signal from a task to itself. As operating system primitive, an exception is a software interrupt (see Section 3.1) used to handle non-normal cases in the execution of a task: numerical errors; devices that are not reachable or deliver illegal messages; etc. The software interrupt gives rise to the execution of an exception handler, that the task (or the

operating system, or another task) registered previously. In high-level programming languages, an exception need not be a software interrupt, but it is a function call to the language's *runtime* support, that will take care of the exception handling.

4.5. Atomic operations

The concept of an *atomic operation* is very important in interprocess communication, because the operating system must guarantee that the taking or releasing a lock is done without interruption. That can only be the case if the *hardware* offers some form of atomic operation on bits or bytes. Atomic operations come in various forms: in the hardware, in the operating system, in a language's run-time, or in an application's support library, but always, the hardware atomic operation is at the bottom of the atomic service. This Section focuses on the *hardware* support that is commonly available.

Most processors offer an atomic machine instruction to *test a bit* (or a byte or a word). In fact, the operation not just *tests* the bit, but also *sets* the bit if that bit has not already been set. Hence, the associated assembly instruction is often called `test_and_set()`, or something similar. Expressed in pseudo-code, the `test_and_set()` would look like this:

```
int test_and_set(int *lock){
    int temp = *lock;
    *lock = 1;
    return temp;
}
```

Another atomic instruction offered by (a fewer number of) processors is `compare_and_swap(address, old, new)`: it compares a value at a given memory address with an "old" value given as parameter, and overwrites it with a "new" value if the compared values are the same; in this case, it returns "true". If the values are not equal, the new value is copied over the old value. Examples of processors with a `compare_and_swap()` are the Alpha, ia32/ia64, SPARC and the M68000/PowerPC. (Look in the Linux source tree for the `__HAVE_ARCH_CMPXCHG` macro to find them.)

The `compare_and_swap()` operation is appropriate for the implementation of the synchronization needed in, for example, *swinging pointers* (see Section 4.10): in this case, the parameters of the `compare_and_swap(address, old, new)` are the address of the pointer and its old and new values.

The following pseudo-implementation is simplest to understand the semantics of the `compare_and_swap()`:

```
int compare_and_swap(address, old, new) {
    get_lock();
    if (*address == old) {
        *address == new;
        release_lock();
    }
}
```

```

    return (1);
} else {
    release_lock();
    return (0);
};

```

The `compare_and_swap()` can, however, be implemented without locks, using the following pair of atomic instructions: `load_linked()` and `store_conditional()`. Together, they implement an atomic read-modify-write cycle. The idea is that the `load_linked()` instruction marks a memory location as “reserved” (but does not lock it!) and if no processor has tried to change the contents of that memory location when the `store_conditional()` takes place, the store will succeed, otherwise it will fail. If it fails, the calling task must decide what to do next: retry, or do something else.

This pair of instructions can be used to implement `compare_and_swap()` in an obvious way, and without needing a lock:

```

int compare_and_swap(address, old, new) {
    temp = load_linked(address);
    if (old == temp) return store_conditional(address, new);
    else return;
}

```

The test `old == temp` need not take place in a critical section, because both arguments are *local* to this single task.

There are some *important caveats* with the `compare_and_swap()` function:

- It only compares the *values* at a given memory location, but does not detect whether (or how many times) this value has changed! That is: a memory location can be changed twice and have its original value back. To overcome this problem, a more extensive atomic operation is needed, the `double_word_compare_and_swap()`, which also checks a *tag* attached to the pointer, and that increments the tag at each change of the value of the pointer. This operation is not very common in processors!
- It is *not multi-processor safe*: (TODO: why exactly?)

While the hardware support for *locks* is quite satisfactory, there is no support for *transaction rollback*. Transaction rollback means that the software can undo the effects of a sequence of actions, in such a way that the complete sequence takes place as a whole, or else is undone without leaving any trace. Transaction rollback is a quite advanced feature, and not supported by operating systems; it’s however a primary component of high-end database servers.

4.6. Semaphore, mutex, spinlock, read/write lock, barrier

Race conditions can occur because the access to a shared resource is not well synchronized between different tasks. One solution is to allow tasks to get a *lock* on the resource. The simplest way to lock is to disable all interrupts and disable the scheduler when the task wants the resource. This is certainly quite effective for the running task, but also quite drastic and far from efficient for the activity of all other tasks. Hence, programmers should not use these methods lightly if they want to maintain real multi-tasking in the system. So, this text focuses on locking mechanisms that do *not* follow this drastic approach. Basically, programmers can choose between two types of locking primitives (see later sections for more details):

1. One based on *busy waiting*. This method has overhead due to wasting CPU cycles in the busy waiting, but it avoids the overhead due to bookkeeping of queues in which tasks have to wait.
2. One based on the concept of a *semaphore*. This method has no overhead of wasting CPU cycles, but it does have the overhead of task queue bookkeeping and context switches.

A generic program that uses locks would look like this:

```
data number_1;
data number_2;
lock lock_AB;

task A
{
    data A_number;

    get_lock(lock_AB);
    A_number = read(number_1);
    A_number = A_number + 1;
    write(number_2,A_number);
    release_lock(lock_AB);
}

task B
{
    get_lock(lock_AB);
    i = ( read(number_1) == read(number_2) );
    release_lock(lock_AB);
    if ( i )
        do_something();
    else
        do_something_else();
}
}
```

The `get_lock()` and `release_lock()` function calls do not belong to any specific programming language, library or standard. They have just been invented for the purpose of illustration of the idea. When either `task A` or `task B` reaches its so-called *critical section*, it requests the lock; it gets the lock if the lock is not taken by the other task, and can enter the critical section; otherwise, it waits (“blocks”, “sleeps”) till the other task releases the lock at the end of its critical section. A blocked task cannot be scheduled for execution, so locks are to be used with care in real-time applications: the application programmer should be sure about the *maximum* amount of time that a task can be delayed because of

locks held by other tasks; and this maximum should be less than that specified by the timing constraints of the system.

The `get_lock()` should be executed *atomically*, in order to avoid a race condition when both tasks try to get the lock at the same time. (Indeed, the lock is in this case an example of a shared resource, so locking is prone to all race conditions involved in allocation of shared resources.) The atomicity of getting a lock seems to be a vicious circle: one needs a lock to guarantee atomicity of the execution of the function that must give you a lock. Of course, (only) the use of an atomic machine instruction can break this circle. Operating systems implement the `get_lock()` function by means of a atomic `test_and_set()` machine instruction (see Section 4.5) on a variable associated with the lock.

Another effective (but not necessarily efficient!) implementation of a lock is as follows (borrowed from the Linux kernel source code):

```
int flags;

save_flags(flags);    // save the state of the interrupt vector
cli();                // disable interrupts
    // ... critical section ...
restore_flags(flags); // restore the interrupt vector to
                    // its original state
sti();                // enable interrupts
```

(Note that, in various implementations, `restore_flags()` implicitly uses `sti()`.)

The implementation described above is not always efficient because: (i) in SMP systems the `cli()` turns off interrupts on *all* CPUs (see Section 3.1), and (ii) if a `test_and_set()` can do the job, one should use it, because the disabling of the interrupts and the saving of the flags generate a lot of overhead.

The lock concept can easily lead to unpredictable latencies in the scheduling of a task: the task can sleep while waiting for a lock to be released; it doesn't have influence on how many locks other tasks are using, how deep the locks are *nested*, or how well-behaved other tasks use locks. *Both* tasks involved in a synchronization using a lock have (i) to agree about which lock they use to protect their common data (it must be in their common address space!), (ii) to be disciplined enough to release the lock, and (iii) to keep the critical section as short as possible. Hence, the locks-based solution to *access or allocation constraints* is equally *indirect and primitive* as the priority-based solution to *timing constraints*: it doesn't protect the *data* directly, but synchronizes the *code* that accesses the data. As with scheduling priorities, locks give disciplined(!) programmers a means to reach deterministic performance measures. But even discipline is not sufficient to guarantee consistency in large-scale systems, where many developers work more or less independently on different parts.

Locks are inevitable for task *synchronization*, but for some common *data exchange* problems there exist *lock-free* solutions (see Section 4.10). The problem with using locks is that they make an application vulnerable for the *priority inversion* problem (see Section 4.8). Another problem occurs when the CPU on which the task holding the lock is running, suddenly fails, or when that task enters a trap and/or exception (see Section 3.1), because then the lock is not released, or, at best its release is delayed.

4.6.1. Semaphore

The name “semaphore” has its origin in the railroad world, where it was the (hardware) signal used to (dis)allow trains to access sections of the track: when the semaphore was lowered, a train could proceed and enter the track; when entering, the semaphore was raised, preventing other trains from entering; when the train in the critical section left that section, the semaphore was lowered again.

Edsger Dijkstra introduced the semaphore concept in the context of computing in 1965, [Dijkstra65]. A semaphore is an *integer number* (initialized to a positive value), together with a set of function calls to *count* `up()` and `down()`. POSIX names for `up()` and `down()` are `sem_wait()` and `sem_signal()`. POSIX also introduces the *non-blocking* functions `sem_post()` (set the semaphore) and `sem_trywait()` (same as `sem_wait()` but instead of blocking, the state of the semaphore is given in the function’s return value).

A task that executes a `sem_wait()` blocks if the count is zero or negative. The count is decremented when a task executes a `sem_signal()`; if this makes the semaphore value non-negative again, the semaphore unblocks one of the tasks that were blocking on it.

So, the number of tasks that a semaphore allows to pass without blocking is equal to the positive number with which it is initialized; the number of blocked tasks is indicated by the absolute value of a negative value of the semaphore count.

The semaphore *S* must also be created (`sem_init(S, initial_count)`) and deleted (`sem_destroy(S)`) somewhere. The *initial_count* is the number of allowed *holders* of the semaphore lock. Usually, that number is equal to 1, and the semaphore is called a *binary semaphore*. The general case is called a *counting semaphore*. Most operating systems offer both, because their implementations differ only in the initialization of the semaphore’s count.

From an implementation point of view, the minimum data structure of a semaphore has two fields:

```
struct semaphore {
    int count; // keeps the counter of the semaphore.
    queue Q;  // lists the tasks that are blocked on the semaphore.
}
```

And (non-atomic!) pseudo code for `sem_wait()` and `sem_signal()` (for a *binary* semaphore) basically looks like this (see, for example, `upscheduler/rtai_sched.c` of the RTAI code tree for more detailed code):

```
semaphore S;

sem_wait(S)
{
    if (S.count > 0) then S.count = S.count - 1;
    else block the task in S.Q;
}
```

```
sem_signal(S)
{
    if (S.Q is non-empty) then wakeup a task in S.Q;
    else S.count = S.count + 1;
}
```

So, at each instant in time, a negative $S.count$ indicates the fact that at least one task is blocked on the semaphore; the absolute value of $S.count$ gives the number of blocked tasks.

The semantics of the semaphore as a lock around a critical section is exactly as in its historical railway inspiration. However, a semaphore can also be used for different *synchronization* goals: if task A just wants to *synchronize* with task B, (irrespective of the fact whether or not it needs to exclude task B from entering a shared piece of code), both tasks can use the `sem_wait()` and `sem_signal()` function calls.

Here is a pseudo code example of two tasks task A and task B that synchronize their mutual job by means of a semaphore:

```
semaphore S;

task A:
main()
{ ...
  do_first_part_of_job();
  sem_signal(S);
  do_something_else_A();
  ...
}

task B:
main()
{ ...
  do_something_else_B();
  sem_wait(S);
  do_second_part_of_job();
  ...
}
```

Finally, note that a semaphore is a lock for which the normal behaviour of the locking task is to go to sleep. Hence, this involves the overhead of context switching, so don't use semaphores for critical sections that should take only a very short time; in these cases *spinlocks* are a more appropriate choice (Section 4.6.3).

4.6.2. Mutex

A mutex (MUTual EXclusion) is often defined as a synonym for a binary semaphore. However, binary semaphore and mutex have an important semantic distinction: a semaphore can be “signaled” and “waited for” by *any* task, while only the task that has *taken* a mutex is allowed to release it. So, a mutex has an *owner*, as soon as it has been taken. This semantics of a mutex corresponds nicely to its envisaged use as a lock that gives *only one task* access to a critical section, excluding all others. That is, the task entering the critical section *takes* the mutex, and *releases* it when it exits the critical section. When another task tries to take the mutex when the first one still holds it, that other task will *block*. The

operating systems unblocks one waiting task as soon as the first task releases the mutex. This mutually exclusive access to a section of the code is often also called *serialization*.

A POSIX mutex, for example, is a (counting) semaphore with *priority inheritance* implied (see Section 4.9). The basic POSIX API for mutexes is:

```
pthread_mutex_t lock;
int pthread_mutex_init( // Initialise mutex object:
    pthread_mutex_t *mutex,
    const pthread_mutexattr_t *mutex_attr
);

// Destroy mutex object.
int pthread_mutex_destroy(pthread_mutex_t *mutex);

// Non blocking mutex lock:
int pthread_mutex_trylock(pthread_mutex_t *mutex);

// Blocking mutex lock:
int pthread_mutex_lock(pthread_mutex_t *mutex);

// Mutex unlock:
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

A *recursive mutex* (or *recursive semaphore*) is a mutex that can be locked repeatedly by the owner. Otherwise the thread that holds a mutex and would try to take the mutex again would lock itself, hence leading to a deadlock. This recursive property is useful for complex mutual exclusion situations, such as in *monitors*, Section 15.2.

The POSIX API requires to indicate explicitly that a mutex should be recursive:

```
pthread_mutexattr_settype(&mutex, PTHREAD_MUTEX_RECURSIVE);
```

Some operating systems (e.g., VxWorks) use the recursive mutex mode as the default. Some offer a so-called *fast* mutex: such a mutex is locked and unlocked in the fastest manner possible on the given operating system (i.e., it doesn't perform any error checks). A fast mutex can only be locked one single time by `pthread_mutex_lock()`, and *all* subsequent calls cause the calling thread to block until the mutex is freed; also the thread that holds the mutex is locked, which causes a deadlock. So, be careful with using fast mutexes.

Many programmers tend to think that a semaphore is necessarily a more primitive RTOS function than a mutex. This is not necessarily so, because one can implement a *counting semaphore* with a mutex and a condition variable (Section 4.7):

```
int sem_wait(sem_t *sem)
{
    pthread_mutex_lock(&sem->mutex);
```

```

while (sem->count == 0) pthread_cond_wait(&sem->cond, &sem->mutex);
sem->count--;
pthread_mutex_unlock(&sem->mutex);
return(0);
}

```

4.6.3. Spinlocks

A “spinlock” is the appropriate lock mechanism for multi-processor systems, and for use in all kinds of contexts (kernel call, interrupt service routine, etc.). They are phasing out the use of “hard” exclusion methods such as `cli()` and `sti()`, because: (i) these are too “global”, in the sense that they don’t specify the context in which the lock is needed; (ii) it is usually not necessary to disable interrupts in order to protect two tasks from entering a critical section. However, you can not do all kinds of things when running inside a critical section locked by a spinlock! For example, do nothing that can take a “long” time, or that can sleep. Use semaphores or mutexes for this kind of locks.

The task that wants to get a spinlock tries to get a lock that is shared by all processors. If it doesn’t get the lock, it keeps trying (“*busy waiting*”) till it succeeds:

```

int spinlock(spinlock_t l){
    while test_and_set(l) {};
};

```

So, it’s clear why you shouldn’t do things that take a long time within a spinlock context: another task could be busy waiting for you all the time! An example of a spinlock in the Linux kernel is the “Big Kernel Lock” (Section 1.2): the BKL is a *recursive* spinlock, i.e., it can be locked multiple times recursively. That means that you (possibly in two separate tasks) can lock it twice in a row, but you also have to release it twice after that.

Spinlocks come in three versions:

1. `spin_lock` and `spin_unlock`: the classical mutual exclusion version, allowing interrupts to occur while in the critical section.
2. `spin_lock_irq` and `spin_unlock_irq`: as above, but with interrupts disabled.
3. `spin_lock_irqsave` and `spin_unlock_irqrestore`: as above, but saving the current state flag of the processor.

All of them work on (the address of) variables of the type `spinlock_t`. One should call `spin_lock_init()` before using the lock. The spinlock versions that disable interrupts do *not* disable interrupts on the *other* CPUs than the one the calling task is running on, in order not to bring down the throughput of the whole multi-processor system. An example (Linux specific!) of the usage (not the implementation!) of a spinlock with local interrupt disabling is given here:

```

spinlock_t l = SPIN_LOCK_UNLOCKED;
unsigned long flags

```

```
spin_lock_irqsave(&l, flags);
/* critical section ... */
spin_unlock_irqrestore(&l, flags);
```

So, both the concurrency and the multi-processor issues are dealt with. On a uni-processor system, this should translate into:

```
unsigned long flags;
save_flags(flags);
cli();
/* critical section ... */
restore_flags(flags);
```

Note: the POSIX function `pthread_spin_lock()` has this semantics of disabling interrupts.

Spinlocks are a trade-off between (i) disabling all interrupts on all processors (costly, safe, but what you don't want to do on a multi-processor system or a pre-emptable kernel), and (ii) wasting time in busy waiting (which is the only alternative that remains). So, spinlocks work if the programmer is disciplined enough to use them with care, that is for guaranteed *very* short critical sections. In principle, the latency induced by a spinlock is *not* deterministic, which is in contradiction to its use for real-time. But they offer a good solution in the case that the scheduling and context switching times generated by the use of locks, are larger than the time required to execute the critical section the spinlock is guarding.

There is a reason why atomic *test-and-set* operations are not optimal on multi-processor systems built from typical PC architecture processors: the *test-and-set* performed by one processor can make parts of the caches on the other processors invalid because part of the operation involves *writing* to memory. And this cache invalidating lowers the benefits to be expected from caching. But the following implementation can help a bit:

```
int spinlock(spinlock_type l){
    while test_and_set(l) { // enter wait state if l is 1
        while (l == 1) {} // stay in wait state until l becomes 0
    };
};
```

The difference with the previous implementation is that the `test_and_set()` requires a read *and* a write operation (which *has* to block memory access for other CPUs), while the `test l == 1` requires only a read, which can be done from cache.

4.6.4. Read/write locks

Often, data has only to be protected against concurrent writing, not concurrent reading. So, many tasks can get a read lock at the same time for the same critical section, but only one single task can get a write lock. Before this task gets the write lock, all read locks have to be released. Read locks are often useful to access complex data structures like linked lists: most tasks only read through the lists to find the element they are interested in; changes to the list are much less common. (See also Section 5.5.)

Linux has a reader/writer spinlock (see below), that is used similarly to the standard spinlock, with the exception of separate reader/writer locking:

```
rwlock_t rwlock = RW_LOCK_UNLOCKED; // initialize

read_lock(&rwlock);
/* critical section (read only) ... */
read_unlock(&rwlock);

write_lock(&rwlock);
/* critical section (read and write) ... */
write_unlock(&rwlock);
```

Similarly, Linux has a *read/write semaphore*.

4.6.5. Barrier

Sometimes it is necessary to synchronize a lot of threads, i.e., they should wait until *all* of them have reached a certain “barrier.” A typical implementation initializes the barrier with a counter equal to the number of threads, and decrements the counter whenever one of the threads reaches the barrier (and blocks). Each decrement requires synchronization, so the barrier cost scales linearly in the number of threads.

POSIX (1003.1-2001) has a `pthread_barrier_wait()` function, and a `pthread_barrier_t` type. RTAI has something similar to a barrier but somewhat more flexible, which it calls “*bits*” (see file `bits/rtai_bits.c` in the RTAI source tree), and what some other operating systems call *flags* or *events*. The *bits* is a 32 bit value, that tasks can share to encode any kind of AND or OR combination of binary flags. It can be used as a barrier for a set of tasks, by initializing the bits corresponding to each of the tasks to “1” and letting each task that reaches the barrier reset its bit to “0”. This is similar to a semaphore (or rather, an array of semaphores), but it is not “counting”.

4.7. Condition variable for synchronization within mutex

Condition variables have been introduced for two reasons (which amount basically to one single reason):

1. It allows to make a task sleep until a certain *application-defined logical criterium* is satisfied.
2. It allows to make a task sleep *within* a critical section. (Unlike a semaphore.) This is in fact the same reason as above, because the critical section is needed to evaluate the application-defined logical criterium atomically.

The solution to this problem is well known, and consists of the *combination* of three things:

1. A *mutex lock* (see Section 4.6.2).
2. A *boolean expression*, which represents the above-mentioned logical criterium.

3. A *signal* (see Section 4.3), that other tasks can fire to wake up the task blocked in the condition variable, so that it can re-check its boolean expression.

The lock allows to check the boolean expression “atomically” in a critical section, and to wait for the signal within that critical section. It’s the operating system’s responsibility to release the mutex behind the back of the task, when it goes to sleep in the wait, and to take it again when the task is woken up by the signal.

There exists a POSIX standard for condition variables. Here are some of the major prototypes for the `pthread_cond_wait()` system call, used to make a task wait for its wake-up signal:

```
#include <pthread.h>

// Initialise condition attribute data structure:
int pthread_condattr_init(pthread_condattr_t *attr);

// Destroy condition attribute data structure:
int pthread_condattr_destroy(pthread_condattr_t *attr);

// Initialise conditional variable:
int pthread_cond_init(
    pthread_cond_t *cond,
    const pthread_condattr_t *cond_attr
);

// Destroy conditional variable:
int pthread_cond_destroy(pthread_cond_t *cond);

// Wait for condition variable to be signaled:
int pthread_cond_wait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex
);

// Wait for condition variable to be signaled or timed-out:
int pthread_cond_timedwait(
    pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime
);

// Restart one specific waiting thread:
int pthread_cond_signal(pthread_cond_t *cond);

// Restart all waiting threads:
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Others system calls that take the same arguments are: `pthread_cond_init()` (initialize the data structure with which a condition variable is built), `pthread_cond_signal()` (signal the fact that a condition variable has changed state), `pthread_cond_broadcast()` (signals the state change to *all* tasks that are waiting for the signal, and wakes them all), `pthread_cond_timedwait()` (wait for the signal, or for a timer to expire, whichever comes first).

The `sem_wait()` of Section 4.6.2 shows a typical application of a condition variable. We repeat the code here for convenience:

```
int sem_wait(sem_t *sem)
{
    pthread_mutex_lock(&sem->mutex);
    while (sem->count == 0) pthread_cond_wait(&sem->cond, &sem->mutex);
    sem->count--;
    pthread_mutex_unlock(&sem->mutex);
    return(0);
}
```

The semaphore has a mutex `sem->mutex`, a condition signal `sem->cond`, and its particular boolean expression, namely its `count` being zero or not. The checking of this condition, as well as the possible decrement of the `count`, must be done in a critical section, in order to synchronize access to the semaphore with other tasks. The `pthread_cond_wait()` function makes the calling task block on the condition variable if the boolean expression evaluates to false. The operating system releases the mutex when the task must block, so that other tasks can use the semaphore. When the condition is signaled (this is done by the complementary function `sem_signal()`, which is not given here but that executes a `pthread_cond_broadcast()`), the calling task is woken up and its mutex is activated (all in one atomic operation) such that the woken-up task can safely access the critical section, i.e., check its boolean expression again. The above-mentioned atomicity is guaranteed by the operating system, which itself uses some more internal locks in its implementation of the `pthread_cond_wait()` call.

It is essential that tasks that wake up from waiting on a condition variable, *re-check* the boolean expression for which they were waiting, because nothing guarantees that it is still true at the time of waking up. Indeed, a task can be scheduled a long time after it was signaled. So, it should also be prepared to wait again. This leads to the almost inevitable `while` loop around a `pthread_cond_wait()`.

The `pthread_cond_broadcast()` should be the default way to signal the condition variable, and not `pthread_cond_signal()`. The latter is only an *optimization* in the case that one knows for sure that only one waiter must be woken up. However, this optimization violates the *loose coupling* principle of good software design (Chapter 14): if the application is changed somewhat, the “optimization” of before could well become a bottleneck, and solving the situation involves looking for the `pthread_cond_signal()` calls that can be spread over various files in the application.

However, blindly using `pthread_cond_broadcast()` can also have a negative effect, called the “*thundering herd*” problem: `pthread_cond_broadcast()` can wake up a large number of tasks, and in the case that only one task is needed to process the broadcast, all other woken-up tasks will immediately go to sleep again. That means the scheduler is hidden under a “herd” of unnecessary wake-up and sleep calls. So, Linux and other operating systems introduced policies that programmers can use to give some tasks the priority in wake-ups.

Both semaphores/mutexes and condition variables can be used for *synchronization* between tasks. However, they have some basic differences:

1. Signaling a semaphore has *always* an effect on the semaphore's internal count. Signaling a condition variable can sometimes have no effect at all, i.e., when no task is waiting for it.
2. A condition variable can be used to check an *arbitrary complex* boolean expression.
3. According to the POSIX rationale, a condition variable can be used to make a task wait *indefinitely long*, but spinlocks, semaphores and mutexes are meant for shorter waiting periods. The reason is that `pthread_mutex_lock()` is not a *cancelling point*, while the `pthread_cond_wait()` is.
4. A condition variable is nothing more than a notification to a task that the condition it was waiting for *might* have changed. And the woken-up task *should* check that condition again before proceeding. This check-on-wake-up policy is not part of the semaphore primitive.

4.8. Priority inversion

Priority scheduling and locks are, in fact, contradictory OS primitives: priority scheduling wants to run the highest priority job first, while a mutex excludes *every* other job (so, also the highest priority job) from running in a critical section that is already entered by another job. And these contradictory goals lead to tricky trade-offs. For example, everybody coding multi-tasking systems using priority-based task scheduling and locking primitives should know about the “priority inversion” danger: in some situations, the use of a lock prevents a task to proceed because it has to wait for a lower-priority task. The reason is that a low-priority task (i) is in a critical section for which it holds the lock that blocks the high-priority task, and (ii) it is itself pre-empted by a medium-priority task that has nothing to do with the critical section in which the high- and low-priority tasks are involved. Hence, the name “priority inversion”: a medium-priority job runs while a high-priority task is ready to proceed. The simplest case is depicted in Figure 4-1. In that Figure, `task H` is the high-priority task, `task M` the medium-priority task, and `task L` the low-priority task. At time instant $T1$, `task L` enters the critical section it shares with `task H`. At time $T2$, `task H` blocks on the lock issued by `task L`. (Recall that it cannot pre-empt `task L` because that task has the lock on their common critical section.) At time $T3$, `task M` pre-empts the lower-priority task `task L`, and *at the same time* also the higher-priority task `task H`. At time $T4$, `task M` stops, and `task L` gets the chance again to finish the critical section code at time $T5$ when, at last, `task H` can run.

Figure 4-1. Priority inversion.

The best-known practical case of a priority inversion problem occurred during the Mars Pathfinder mission in 1997. (More information about this story can be found at <http://www.kohala.com/start/papers.others/pathfinder.html> or http://research.microsoft.com/~mbj/Mars_Pathfinder/Mars_Pathfinder.html.)

4.9. Priority inheritance and priority ceiling

Operating system programmers have tried to “solve” (not prevent) the priority inversion problem, in two

different ways:

- *Priority inheritance.* A low-priority task that holds the lock requested by a high-priority task temporarily “inherits” the priority of that high-priority task, *from the moment the high-priority task does the request.* That way, the low-priority task will not be pre-empted by medium-level priority tasks, and will be able to finish its critical section without holding up the high-priority task any longer than needed. When it releases the lock, its priority drops to its original level, while the high-priority task will now get the lock. The maximum predictable delay is the length of the critical section of the low-priority task.

Priority inheritance generates *run-time* overhead, because the scheduler has to inspect the priorities of all tasks that access a lock.

- *Priority ceiling.* Every lock gets a priority level corresponding to the priority of the highest-priority task that *can* use the lock. This level is called the *ceiling priority*. Note that it is the *lock* that gets a priority, which it gives to every task that tries the lock. So, when the low-priority task enters the critical section, it *immediately* gets the ceiling priority from the lock, such that it will not be pre-empted by any medium-level priority task. Therefore, another name of the priority ceiling protocol is *instant inheritance*.

Priority ceiling generates *compile-time* overhead, because it can already at that moment check the priorities of all tasks that will request a lock.

Priority ceiling has the pleasant property that it simplifies implementation and has small run-time overhead (only the change in priority for the task entering a critical section): the lock *never has to be tested* for being free or not, because any task that tries the lock runs at the highest priority to enter the critical section: any other task that could test the lock would run at the same ceiling priority, and hence would not have been interrupted in its critical section by the task that currently tests the lock. Indeed, both tasks live in the same priority level and are scheduled with a *SCHED_FIFO* policy. Instant inheritance also offers a solution to the “deadly embrace” (see Section 4.2) occurring when two tasks lock *nested* critical sections in opposite order: the first task to enter the outermost lock will have the appropriate priority to finish the complete set of nested critical sections.

A possible problem with priority ceiling is that it makes more processes run at higher priorities, for longer times than necessary. Indeed, the priorities of tasks are changed, *irrespective* of the fact whether another task will try to request the lock or not. This reduces the discriminating effects of using priorities in the first place, *and* it gives rise to “hidden” priority inversion: while task *L* gets its priority raised to the ceiling priority because it is involved in a lock with another task *V* that has a very high priority, a third task *H* not involved in the lock could get pre-empted by *L* although its priority is higher and *V* is dormant most of the time.

Priority ceiling and inheritance look great at first sight, and they are part of some OS standards: priority ceiling is in the POSIX standard (*POSIX_PRIO_PROTECT*), the Real-Time Specification for Java (RTSJ), OSEK, and the Ada 95 real-time specifications. Priority inheritance is also part of standards such as POSIX (*POSIX_PRIO_INHERIT*), and the RTSJ. But priority ceiling and inheritance can still not

guarantee that no inversion or indeterministic delays will occur, [Yodaiken2002], [Locke2002]. Moreover, the priority inheritance “feature” gives rise to code that is more complex to understand and certainly to predict. Also, determining *a priori* the ceiling priority for a lock is not an easy matter (the compiler must have access to *all* code that can possibly use a lock!), and can cause portability and extendability headaches.

Priority inversion is always a result of a *bad design*, so it’s much better to *prevent* race conditions instead of “solving” them. However, contrary to the deadlock prevention algorithm (Section 4.2), no similarly simple and guaranteed algorithm for priority inversion is known. So, all an operating system could do to help the programmers is signalling when priority inversion takes place, such that they can improve their design.

Most RTOSes don’t apply priority inversion solutions for every case of sender-receiver synchronization. For example Neutrino (from QNX) uses separate synchronization mechanisms for critical sections (semaphores) and sender-receiver (which is synchronous IPC in QNX). It solves priority inversion only so long as applications use a many-to-one IPC model. As soon as an application uses many-to-many IPC (via a POSIX queue) there is no more prevention of priority inversion. Many-to-many is inherently difficult because the kernel has no way to know which receiver might be ready next, so all it could do would be to raise the priority of all potential listeners (and the processes upon which they are waiting). And this would often result in a logjam as every process was raised to the same priority, invalidating exactly the major reason why priorities were introduced in the first place.

4.10. Lock-free synchronization for data exchange

Some synchronization can also be done *without* locks, and hence this is much more efficient and guaranteed to be deadlock-free, [Herlihy91], [Herlihy93]. Lock-free synchronization uses the `compare_and_swap(address, old, new)` (see Section 4.5), or similar constructs. This functionality is applicable to the manipulation of *pointers*, e.g., to interchange two buffers in one atomic operation, or to do linked list, queue or stack operations.

The following code fragment shows the basic form of this *pointer swinging*:

```
ptr = ...
do {
    old = ptr;
    new = new_value_for_pointer;
    while ( !compare_and_swap(ptr, old, new) );
```

If the `compare_and_swap()` returns “false”, the swinging of the pointers should not be done, because some other task has done something with the pointer in the meantime.

Recall the possible problem with `compare_and_swap()`: it only compares the *values* of the addresses, not whether this value has been changed! This means that a double change of the pointer (back to its original value) will not be detected. This occurs quite frequently, i.e., any time when memory space is re-used, e.g., in a stack or a linked list.

Another problem of using `compare_and_swap` for lock-free synchronization is that it is not always the most efficient method available, because it involves the *copying* of a complete data structure before that data structure can be updated without a lock.