Concurrent and Real-Time Programming

Theory Lecture Notes - A.Y. 2022/2023

Lorenzo Saccaro

compiled on June 14, 2023

1 Operating Systems review

1.1 Input/Output (I/O) in computers

I/O requires a **Communication bus**, the main elements are:

- **Data lines**: where the content is put
- **Address lines**: to pass where to/from write/read
- **Handshaking lines**: for synchronization

Separate memory and I/O buses (Fig. 1) are possible while modern architectures define a common bus (Fig. 2) called **Memory mapped I/O**. In this case, the address range of the connected devices must be disjointed from the address range for memory.

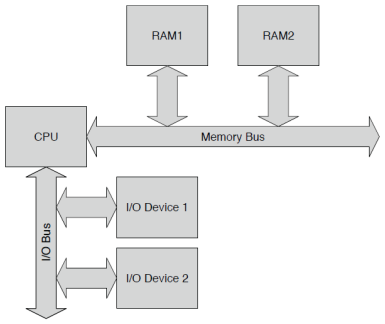

Fig. 1: Bus architecture with a separate I/O bus

Every device itself has a set of **registers** at given 'memory' addresses (every device has its own range). Each register performs a given functionality that depends on the given device type. In general, the following types of registers are present:

- **Status registers**: its bits bring information on current device status
- **Command registers**: written, they issue commands to the device
- **Configuration registers**
- **Data registers**

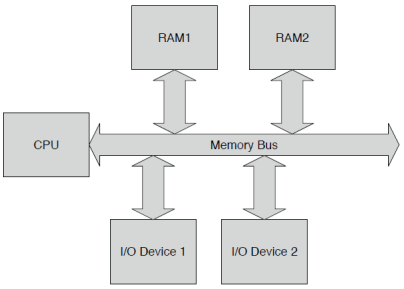

Fig. 2: Bus architecture for memory mapped I/O

1.2 PCI/PCI-e bridges

Connecting all the external I/O devices to the memory bus is not ideal (performance degradation + risk of malfunctions), therefore one or more separate buses are present. A common bus for I/O in computers is the **Peripheral Component Interconnect (PCI)** bus:

- It is a **parallel bus**. The same lines are shared among the connected components
- At most 2 devices can be connected to the same PCI bus
- At system startup, connected devices have to provide the required address ranges

PCI express (PCI-e) represents an evolution of PCI using **serial links** instead of the shared parallel bus. It is further characterized by the number of lanes used for serial communication (x1, x2, x4, x8, x16 etc.).

PCI and PCIe are then connected to the memory bus via **bridges** (Fig. 3): a bridge is a device itself with its own set of registers. It is 'programmed' at startup starting from those connected to the memory bus. Every bridge recognizes a bus access in the assigned (by the architecture designer) range and **translates** (adds an offset to the memory addresses) it in a new bus cycle on the connected bus (no direct talk to device but only between buses).

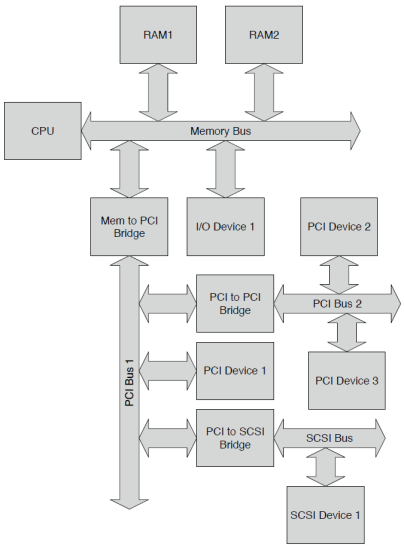

Fig. 3: Bus architecture with two PCI buses and one SCSI bus

1.3 Synchronizing I/O

In order to perform correct I/O operations, the computer needs **to know when the device is ready to provide/consume a new data item** (i.e. the bits contained in the data register are the ones I am supposed to read?).

The simplest method is to define a bit in the status register that specifies whether the device is ready. The computer will repeatedly read the status register until it detects that the device is ready, and then read/write the corresponding data register. This is called **polling**: a very basic method for synchronization, useful for very fast device transferring few data. Otherwise, it may lead to a **waste of computer resources**, e.g.:

- 9600 serial communication link, max 1200 B/s → new byte every 0.83 ms
- Assuming 100 ns for memory access (no cache effects) → on average 8000 read ops to exchange a byte → 99.99% CPU time is wasted

Other techniques exist that allow the **CPU to perform independent work while the device is processing I/O**. A new mechanism is required in order to inform asynchronously the availability of the device → **new handshaking lines in the communication bus + hardware and software functionalities in the CPU and the Operative System (OS)**.

1.4 Interrupt management

When a device needs to communicate an asynchronous event (e.g. new data ready to be read/written) it raises one of the **interrupt lines**. The processor as soon as it finishes executing the current machine instruction will serve the interrupt request by executing a specific routine, the **Interrupt Service Routine (ISR)**.

Since in a computer there could be multiple devices, more than one interrupt could be issue at the same time. For this reason, a **priority** is associated with interrupts (separate lines for each priority level). A new interrupt request will be served immediately if its priority is higher than the current one, otherwise it will be served as soon as the previous ISR has terminated and there no pending interrupts with priority greater or equal to the current one.

When the processor starts serving an interrupt **it must not lose information about the program currently in execution**: which is described by the associated memory contents (program itself + data) and by the content of the **processor registers**, including the **Program Counter (PC)** (which records the address of the current machine instruction) and the **Status Register (SR)** (which contains information on the current processor status). Processor register that needs to be used during the ISR are saved in the stack at the beginning of the routine, but this is not possible for the PC (it has just changed and set to the address of the first instruction of the routine) and for the SR (it will be updated to show that an ISR is being served). For this reason, the **processor itself saves the PC e SR** (in the stack or in designated registers) and are restored at the end of the ISR.

Since a specific ISR has to be associated with every possible source of interrupt, computer architectures define a vector of addresses in memory, called **Vector Table**, containing the start addresses of the ISRs for all the I/O devices able to generate an interrupt request. The offset of a given ISR within the vector table is called the **Interrupt Vector Number (IVN)** (it is communicated by the device so that the processor calls the right ISR). When the processor starts serving an interrupt, it performs a cycle on the bus called **Interrupt Acknowledge Cycle (IACK)**: the processor communicates the priority of the interrupt being served and the device which issued the interrupt at the specified priority returns the IVN.

If **two different devices** issued interrupt request with the **same priority** at the **same time**, the device closest to the processor in the bus will be served first. This achieved by defining a bus line in **Daisy Chain** configuration: which is propagated from every device to the next one along the bus only if it has not answered to an IACK.
So, a device will respond to a IACK only if both conditions are satisfied:

1. It has generated a request for interrupt at the specified priority
2. It has received a signal over the daisy chain

In that case it will not propagate the daisy chain to the next device.
The IVN returned by a device depends on the current organization of the vector table and so it must be a programmable parameter. Devices (which request interrupts) have two registers for the definition of the interrupt priority and the IVN.

To recap, the sequence of actions are (Fig. 4):

1. The device issues an interrupt request;
2. The processor saves the context, puts the current values of the PC and of the SR on the stack;
3. The processor issues an IACK on the bus;
4. The device responds by putting the IVN over the data lines on the bus;
5. The processor uses the IVN as an offset in the vector table and loads the ISR address in the PC.

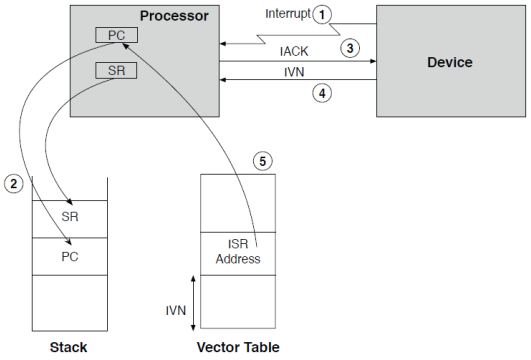

Fig. 4: The Interrupt Sequence

1.5 Required SW actions for properly handling interrupts

The SW device driver is the code for managing a given device. It is not directly accessible by users but is it plugged within the OS. The driver shall specify and 'connect' the code to be executed when the device issues am interrupt. During device installation (at startup, performed by BIOS) **the driver updates the vector table** by inserting the ISR address and **set the IVN** (in a designated register) to be returned when an IACK is issued.

During the execution of an ISR no other activity, OS ones included, **can be executed**. The OS itself makes heavy use of interrupts to take control of the computer regardless the operation carried out by user code (e.g. infinite loop). On average, during normal operation, 100s-1000s of interrupts occurs every second.

1.6 Direct Memory Access (DMA)

There are cases in which the use of interrupts for I/O synchronization would require too much computing resources. Mouse: 30 update/s, 400 CPU cycles to handle interrupt and run ISR $\rightarrow$ 12,000 cycle/s spent, for a 1 GHz processor this corresponds to $12 \times 10^3/10^9 = 12 \times 10^{-6} = 0.0012\%$ of CPU load.
HDD: 4 MB/s transfer rate, interrupt every 16 bytes of data available. 400 CPU cycle to handle interrupt and run ISR $\rightarrow 4 \times 10^6/16 = 250,000$ interrupts/s $\rightarrow 400 \times 250,000 = 100,000,000$ CPU cycles, for a 1 GHz processor this corresponds to 10% of CPU load. Not acceptable.

Interrupt driven data transfer is therefore **not the best solution for high throughput devices**. In this case **Direct Memory Access (DMA)** is used. The device is responsible for the transfer of data from/to memory, bypassing the CPU: in this case the device becomes **owner of the bus** in order to initiate a data transfer cycle: this happens after a bus ownership transfer cycle is issued. Before the data transfer, the **start address of data target in memory** and the **number of data items** to transfer must be communicated by the device driver. For this operation, the **Memory Address Register (MAR)** and the **Word Counter (WC)** device registers are used.

Whenever a data block is available the device shall take bus ownership and transfer the block. The bus ownership will likely be exchanged multiple times between the processor and the device(s) during the overall transfer. In the meantime, the processor operates independently and the only interference is the possible request for bus ownership for memory access (normally negligible). When the DMA terminates, the CPU is informed via an interrupt.

1.7 User and Kernel mode

Most processors define at least two level of execution: **user mode** and **kernel** mode. When in user mode, a program is not allowed to execute some machine instruction, called **Privileged Instructions** or to access sets of memory addresses. Most of the OS code is executed in kernel mode.

How can programs switch from user to kernel mode? For security reasons there is not a specific machine instruction that perform this switch. The **processor only switches to kernel mode when serving an ISR** and when it returns the interrupted program resumes and the execution mode is switched to user mode (unless there is another ISR pending). This makes sense since ISR interact with devices and are part of the device driver which is a software component integrated in the OS (different devices are integrated in the OS using the device layer abstraction, i.e. a set of rules to which the device code must adhere: **insmod** command integrates binary of driver in vector table).

In order to switch from user mode to kernel mode under program control, the CPU instruction set includes the **Software Interrupt** instruction whose effect is exactly the **same of a hardware interrupt**, with the **exclusion of the IACK** and **the vector number** associated with the software interrupt **is passed as an argument in the instruction itself**.

This is the way the OS code is organized: the non-privileged code is available as a library, and the privileged one as a set of ISR associated with a set of software interrupt numbers.
As an example, let's consider the execution steps of the `printf()` of the C library:

1. The program, running within a given process, calls routine `printf()`, provided by the C runtime library. Arguments are passed on the stack and the start address of the `printf` routine is put in the PC;
2. The `printf` code carries out the required formatting of the passed string and the other optional arguments, and then calls the OS specific system service for writing the formatted string on the screen;
3. The system routine executes initially in user mode, makes some preparatory work then needs to switch in kernel mode. It issues a software interrupt, where the passed IVN specifies the offset in the vector table of the corresponding ISR to be executed in kernel mode;
4. The ISR is eventually activated by the processor in response to the software interrupt. This routine is provided by the OS and is executed in kernel mode;
5. After some work to prepare the required data structures, the ISR routine will interact with the output device. To do this, it will call specific routines of the device driver;
6. The activated driver code will write appropriate values in the device registers to start transferring the string to the video device. In the meantime, the calling process in put in wait state;
7. A sequence of interrupts will be likely generated by the device to handle the transfer of the bytes of the string to be printed on the screen;
8. When the whole string has been printed on the screen, the calling process will be resumed by the OS and `printf` will return.

1.8 Virtual Memory

Virtual memory, supported by most general-purpose operating systems, is a mechanism by which the memory addresses used by the programs (in user mode) do not correspond to the addresses the CPU uses to access the RAM memory in the same instructions.

The address translation is performed by a component of the processor called the **Memory Management Unit (MMU)**. The details of the translation may vary but the basic mechanism always relies on a data structure called **Page Table**. The memory address managed by the user program, called **Virtual (Logical) Address**, is translated by the MMU diving its N bits in the K least significant bits and the remaining N-K bits. The most significant N-K bits are used as an index in the Page Table, which is composed as an array of numbers, each L bits long. The entry in the Page Table corresponding to the given index is then paired with the least significant K bits to form a number of L+K bits that represent the **Physical Address** which will be used to read the physical memory (Fig. 5). This corresponds to providing a logical organization of the virtual address range in a set of **Memory Pages** each 2^K bytes long: the most significant N-K bits provide the memory page number and the least significant K specify the **offset** within the memory page.

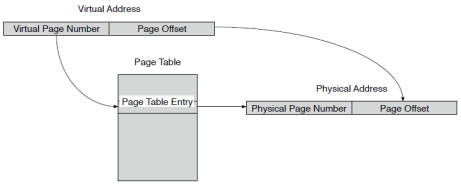

Fig. 5: The Virtual Memory address translation

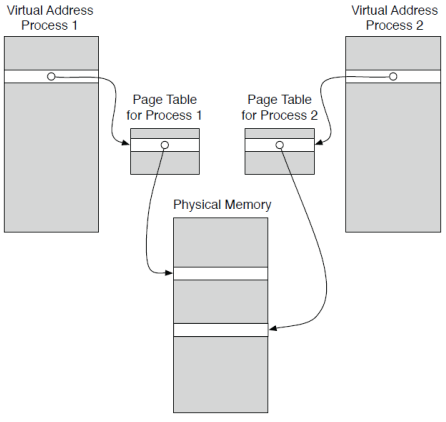

Fig. 6: The usage of virtual address translation to avoid memory conflicts

Why go through all this? Let's consider two processes running the same program. Since a program is composed of a sequence of machine instruction handling data in processor registers and in memory, if no virtual memory were supported, the two instances of the same program run by two different processes would interfere with each other since they would access the same memory locations (they are running the **same** program). This is solved using the virtual memory mechanism, by providing two different mappings to the two processes so that the same virtual address page is mapped onto two different physical pages (Fig. 6).

Since the address translation is driven by the content of the page table, the OS whenever it assigns the processor to one process it has also to set the corresponding page table entry. The page table contents become therefore part of the set of information, called **Process Context**, which needs to be restored by the OS in a context switch, that is whenever a process regains the usage of the processor.

1.9 Shared Memory management

The address translation mechanism can be used also to let separate processes share a section of memory. **Interprocess communication (IPC)** can be achieved with virtual memory by involving the OS so that it can set up the content of the page table in order to allow the sharing of one or more physical memory pages among different processes (Fig. 7). Memory sharing is always orchestrated by the OS under request from the involved actors.

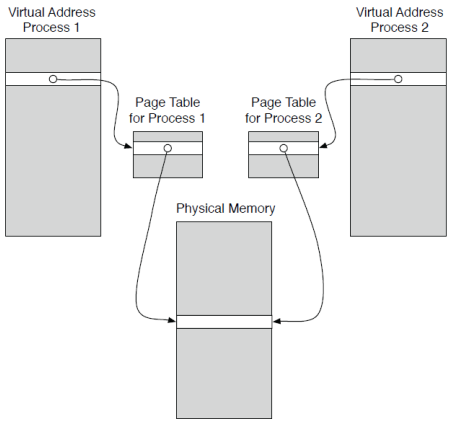

Fig. 7: Using the Page Table translation to map possibly different virtual addresses onto the same physical memory page

1.10 Memory swapping

Additional information can be defined in each **Page Table Entry (PTE)**, such as protection bits that define whether the page is accessible in user mode. In case an illegal page is accessed, an exception is generated by the MMU HW itself.

It is also possible to handle a **virtual address range** that is **larger than the physical one**. In this case a bit in the PTE shall specify whether the page is resident in memory: if true, the physical page number is contained in the PTE, else, the disk address of the corresponding page on disk.

When a page non-resident in memory is accessed, the MMU shall issue a **Page Fault Exception**. It does not signal an error condition, but it triggers a sequence of actions orchestrated by the OS:

- The current process is suspended;
- A free page in memory is retrieved;
- An I/O operation (DMA read) is started for copying the page from the disk into the memory page;
- The PTE content is updated to reflect the new configuration;
- The suspended process is made ready again.

When **the number of free memory pages goes under a given threshold, a swapping action begins** in order to copy one or more memory pages into the corresponding disk blocks.

1.11 Caching

Ram access is costly (10-100 ns) in respect of internal register access: for this reason, the sections of memory most frequently accessed are stored in a **local fast memory cache** (L1, L2). When the processor accesses a part of memory that is not already in the cache it **loads a chuck of memory around the accessed address into the cache**, hoping that it will soon be used again. The chunks of memory handled by the cache are called **cache lines**. The size of these chunks is called the cache line size (32, 64, 128 bytes are common values). A cache can only hold a limited number of lines, determined by the cache size (e.g. a 64 KB cache with 64 B lines has 1024 cache lines).

1.12 Locality in memory access

Both swapping and caching take advantage from the **locality** in memory access. When a given address is used, it is likely that **the next access will be at address close to the previous one**: sequential programs imply sequential memory accesses in instruction fetch and array normally accessed in loops with increasing/decreasing indexes.

Therefore, if in memory access the data item is not found in the cache (**cache miss**) an **entire cache line is copied from the RAM into the cache** so that the next access will likely find the data item in the cache (**cache hit**). An example is how the C language organizes matrix memory (row by row).

1.13 Translation Lookaside Buffer (TLB)

Associative memory with the same cache technology is used to speed up address translation. A line in the **Translation Lookaside Buffer (TLB)** is searched (in parallel) with a tag corresponding to the virtual page number (this is done in HW, ≈ns). If found, the physical page number is appended to the physical address, otherwise the MMU is accessed and the TLB updated (Fig. 8).

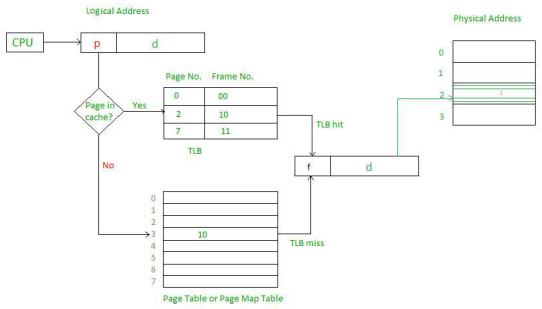

Fig. 8: Virtual address translation with TLB

1.14 Pipelining

The execution of a machine instruction is composed of several stages:

- Instruction Fetch
- Instruction Decode
- Arguments readout
- Execution
- Output write

When a given stage has finished execution, it is ready to process the next instruction, while the former one is being processed. For N stages the ideal speedup is N (Fig. 9).

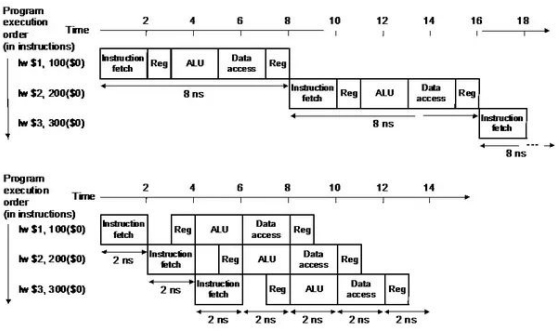

Fig. 9: Instruction pipelining

There are conditions preventing the **ideal** pipeline execution:

- **Data Hazard**: when an instruction depends on the instruction just preceding it in the program.
  For example:
  ```
  ADD #2, R1, MEM1
  SUB MEM1, R2, R4
  ```
  (In this case the result of the ADD instruction must be stored on memory before the following instruction can read arguments.)
- **Branch Hazard**: the address of the next instruction after a conditional branch is not known until the branch condition has been evaluated
- **Cache misses**: may suspend an instruction for several clock cycle (memory access)

To mitigate **data hazards** and the effect of **cache misses** the processor **reschedules on the fly the order of the instructions**, while preserving the correctness of the program.

For example:
```
MOVE #2, R1;
ADD R2, R3, MEM1;
SUB MEM1, R2, R3;
```
can be arranged as:
```
ADD R2, R3, MEM1;
MOVE #2, R1;
SUB MEM1, R2, R3;
```

To mitigate **branch hazard**, modern processors perform **branch prediction**. After fetching a conditional branch instruction, the next instruction to fetch depends on the evaluation of the branch instruction. Nevertheless, the address of the next instruction is assumed to be the most recently used in the same instruction (or the next one if it is the first time that branch instruction is executed).
A cache-like table is maintained keeping the recent history of the branch instructions so that a decision is taken soon. A line is searched where the instruction address corresponds to the current fetch address.
If found, the second element is taken as address for the next fetch stage. When the branch condition is evaluated at a later stage and the assumption was not correct, the pipeline is reverted (with a delay of some clock cycles) and the next correct instruction is fetched. This operation is performed **entirely by the HW**.

1.15 Code Optimization techniques

The following optimization techniques are mostly carried out by the compiler:

- **Register allocation**: use as far as possible CPU registers to hold the content of the program variables. Moves away from the stack-based call frame organization.
- **Code reduction**: perform at compile time all the possible computation and use more efficient operations (e.g. replace multiplications with sums), move away from loops operations that need to be performed only once
- **Loop unfolding**: when convenient unfold loops in order to reduce the overhead of branch condition check and to reduce branch hazards (jump instruction is expensive)

- **Code reordering**: in order to reduce the effects of data hazard

Can the programmer help the compiler optimizer in some way?

Very sophisticated speculation is carried out by the compiler optimizer, especially when the program semantics is defined using program variables. **When pointers and memory access are used instead, the compiler cannot speculate** what is going to happen to the memory locations that may be changed from external actors. Therefore, is up to the programmer to make good usage of memory organization, **favoring as far as possible localized memory access in order to make best usage of caching**. As a rule of thumbs, using a large number of variables in place of pointers (e.g. mapping a sliding memory region in a set of variables) allows the optimizer to make a much better work.

2 Processes, Threads and Scheduling

2.1 Handling multiple programs via processes

The process abstraction is the base mechanism to allow concurrent execution of multiple programs on a single processor (Fig. 10). Modern machine host multicore processors and therefore the compute activity is **a mix of true parallel and OS managed multiprogramming**.

We will address: what actions are required to transfer processor control from one program to another? What is the lifeline of a process? How the OS can take control of the processor in order to schedule processes? What factors are considered in the choice of the running process?

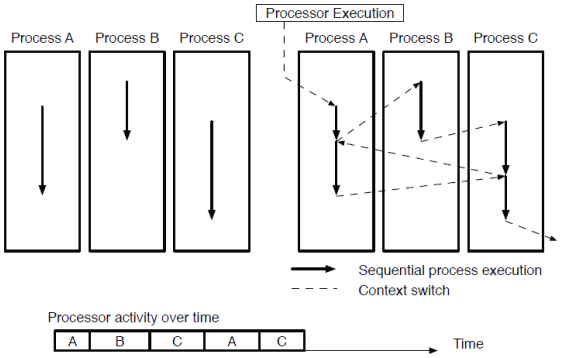

Fig. 10: Multiprogramming: abstract model of three sequential processes (left) and their execution on a single-processor system (right)

2.2 Process Context

A running **program brings a set of information that change over time** (Fig. 11), therefore **a snapshot** of the associated information **must be saved by the OS when the program loses processor ownership**.

Main components of the process context are:

1. **Program code**: also called the text of a program;
2. **Processor state**: program counter (PC), general purpose registers, status words, etc.
3. **Data memory**: program's global variables and the procedure call stack that, for many programming languages, also holds local variables.
4. **OS resources**: the state of all OS resources assigned and being used by the process: open files, I/O devices... .

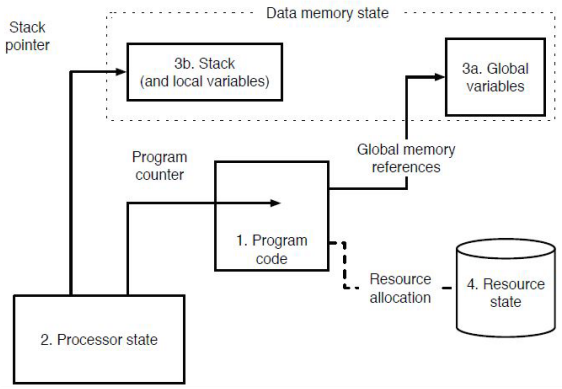

Fig. 11: Graphical representation of the process state components

Registers and OS resources are saved in a data structure owned by the OS called the **Process Control Block (PCB)**. It would be however not possible to copy the content of the memory used by the process, and indeed memory contents are indeed left where they are. It is therefore necessary that physical memory locations are independent from memory locations as seen by the process (virtual addresses). In this case, the **Page Table entries** (possibly referring to swapped pages) **used by the process are saved in the PCB**. This includes:

- The program code
- The program stack for local variable
- The static memory content for global and static variables
- The memory dynamically allocated in the Heap

The amount of information to be copied in the PCB can be significant if the process is using a large amount of memory and potentially affect the speed of the process context. TLB must be flushed upon context switch.

The use of virtual memory ensures protection against wrong memory access in process code. Without it, user programs may harm other processes' memory or, even worse, OS data structures.

Using virtual memory, the process is given a memory area (the pages mapped in the corresponding PTEs) and any access within this area is legal (program bugs included). If the process accesses a location whose address is not mapped, an exception (interrupt-like) is generated by the MMU HW and the OS regains control and normally the process is aborted. In any case, **OS and other process memory integrity is preserved**.

2.3 The Process states

During their life, **processes can be in one of several different states**. They go from one state to another depending on their own behavior, OS decision or external events. **At any instant, the OS has the responsibility of keeping track of the current state of all processes under its control**.

Process states and transition rules can be described with the **Process State Diagram (PSD)** (Fig. 12), a directed graph in which nodes represent states and edges transitions. At any instant, a process can be in one of the following states:

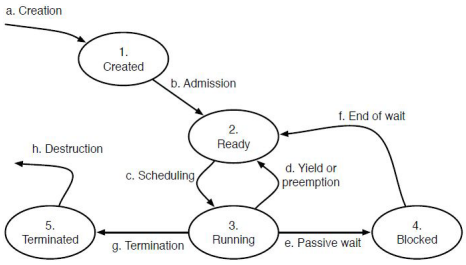

Fig. 12: Process State Diagram

1. A process is in the **Created** state immediately after creation. It has a valid PCB associated to it, but it does not yet compete with the other processes.
2. A process is **Ready** when it is willing to execute (and competes with other processes to do so), but at the moment no processor is available, therefore no progress is made.
3. A process being actively executed by a processor is in the **Running** state (upper limit is equal to number of processors).
4. When a process is waiting for an external event to occur (e.g. an I/O operation) or for synchronization with other processes (will see it later) is put in the **Blocked** state, and it does not compete for execution.
5. When a process terminates, before its destruction, it is put in the **Terminated** state: it cannot be executed but its PCB is still available to other processes (e.g. to see if it terminated spontaneously or due to an error).

Let's examine now the possible **state transitions**:

a. **Creation**: another process creates it and the OS correctly initialized its PCB (and checks some minimum resources requirements)
b. **Admission**: transition from Created to Ready state performed by OS (different behavior between general purpose and real-time systems)
c. **Scheduling**: transition from the Ready to Running state is controlled by the OS scheduler according to its scheduling algorithm and it is transparent to the process that experience it
d. The transition from Running to Ready can be due to two distinct reasons:
   - **Preemption**: decided by the OS scheduler, the process is forced to relinquish the processor even if it is still willing to execute. It can occur anytime and anywhere during execution.
   - **Yield**: requested by the process itself to ask the system to reconsider its scheduling decision and possibly hand over the processor to another process. It may occur only at specific location in the code and at the time the process requests it
e. **Passive wait**: when a process in Running states voluntarily hands over the processor and enters the Blocked state for a synchronous I/O request or interprocess communication, waiting for an external event to occur.
f. **End of wait**: when the event the process is waiting for eventually occurs, it moved from the Blocked state back to the Ready state.
g. **Termination**: a process may go from the Running to the Terminated state when it ends its execution or an unrecoverable error occurs.
h. **Destruction**: a process and its PCB are ultimately removed from the system with a final transition out of the Terminated state (it happens automatically or after explicit request, care about using PID number that could be invalid or refer to the wrong process)

2.4 Threads

In many applications, there are **several distinct concurrent activities** that are nonetheless related to each other, for example, because they **have a common goal**. In this case, **implementing them as distinct processes may be difficult** because the **different processes must share resources** such as memory structures and open files. It may therefore be useful to manage all these activities as a group and share system resources (files, devices, network connections...).

This can be done conveniently by envisaging **multiple flows of control, or threads, within a single process** (Fig. 13). All of them **refer to the same address space and thus share memory**. In particular **they share the page table** (same virtual address space) and **open files**, alongside with part of the memory contents such as **global variables** (static) and **the program code** itself.

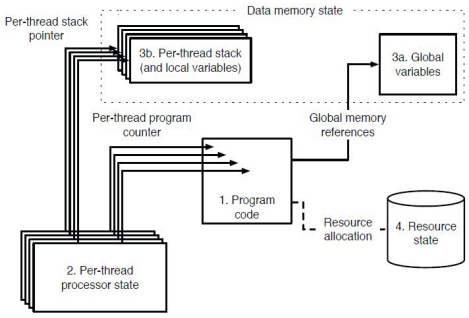

Fig. 13: Process sate components in a multithreading system

Other information is **thread specific**: every thread is running a different program (different routines of the same main program). This information includes: **processor registers**, including the **PC** and **local program variables in a per-thread stack**.

Variables allocated on the per-thread stack are those declared as **local variable in the C routines. Different threads can execute the same routine without affecting other local variables** (Fig. 14).

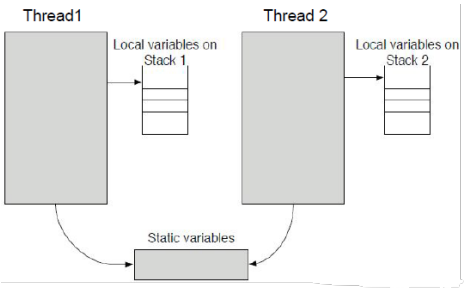

Fig. 14: Thread memory model

Variables declared **outside routines** and as **static in the C code** are shared among threads. This offers an easy way for **sharing memory structures among threads in the same process**. However, shared memory alone is in general not enough to ensure correct communication and synchronization among threads.

A drawback of multithreading is that in case a single thread fails due to an error, all other threads and the related process are killed (e.g. multithreading not a good idea for a web server, if a single connection fails everything crashes, also for memory security reasons, not access data I am not supposed to).

2.5 The Scheduler

The scheduler is the OS component that supervises the selection of the process or thread to become running. Threads and processes play an equivalent role and in the following they shall be referred as **tasks**. The selection of the task to become running is based on its **priority**: the ready task at the highest priority shall be selected.

Scheduler action may be requested when:
An interrupt occurs, possibly changing the state of a pending I/O operation and making a waiting process ready.
Any process issues an OS call (e.g. I/O ops) possibly changing the state of the calling process from running to wait. The involved OS code shall make a call to `schedule()` at the end of the corresponding action. An important interrupt source is the **Timer Interrupt**, issued normally at 60Hz, that enables the update of dynamic priorities and time slices.

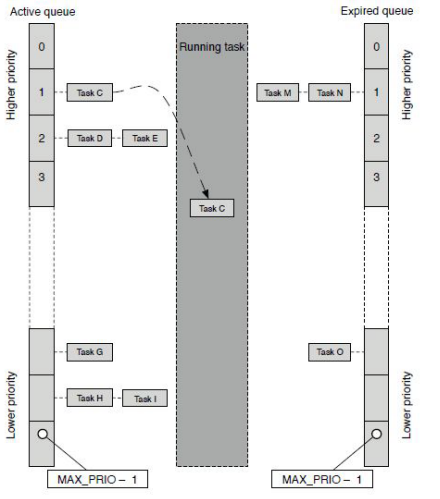

Fig. 15: Data organization of the Linux $\mathcal{O}(1)$ scheduler

The **scheduler organizes ready tasks in queues based on their priority** (Fig. 15). Every task that is not declared as FIFO (First In First Out) task is assigned a **time slice in order to ensure fairness among task of the same (highest) priority**.

`scheduler_tick()` is called at every timer interrupt. It decreases the current time slice of the currently running task. Whenever the time slice reaches zero, the task is moved to the expired queue and the processor will be assigned to the next task in the active queue (via a call to `schedule()`).

If the active queue is empty for a given priority, it shall be swapped with the corresponding expired queue. Selection of the highest priority queue is performed in $\mathcal{O}(1)$ time using a bitmap.

2.6 Task Priority

In Linux 140 priority levels are defined (lower number → higher priority). Priorities 0 to 100 are **fixed** while the others are **dynamic**. These can be dynamically changed by the OS based on a given **niche** value.

Dynamic priority adjustment aims at providing improved fairness, in practice a more **fluid** user interaction. Priority adjustment is carried out during Timer Interrupts, decreasing a counter associated with the currently running task. When the counter reaches 0, the priority of the task is lowered; with a similar mechanism the priority of the waiting tasks is increased. The rationale behind is to **let tasks that tend to use less CPU** (e.g. lots of I/O ops) **a higher priority**. In this way the computer shall not be blocked even if the highest priority task is running an infinite loop.

Dynamic task priority is important in improving user interaction, avoiding annoying blocks in task execution (because of a CPU consuming higher priority task). Conversely, it is not possible to ensure that an important task to which the highest priority has been assigned will retain its priority → its **latency** may increase.

Latency is defined as **the time between the instant in which an event occurs that makes the task ready and the time the task gains processor ownership**. Events may be: the termination of an I/O ops, the availability of a new input, the termination of a given interval (cyclic tasks), the occurrence of an interrupt signaling a condition to be served. Several factors affect task latency, among which: the presence of tasks at higher or equal priority, the number of available cores and the OS latency.

3 Models of Parallel Processing

3.1 Task Interaction

In both general general-purpose and real-time systems, **processes** do not live by themselves and **are not independent to each other**. Rather, several processes are brought together to form the application software, and they must therefore cooperate to solve the problem at end. **Processes must therefore be able to communicate, that is, exchange information in a meaningful way**. We have already seen that it is possible to share some memory among processes in a controlled way by making part of their address space refer to the same physical memory region or share global variables if threads within the same process.

However, this is only part of the story. In order to implement a correct and meaningful data exchange **processes must also synchronize their action in some way**. For instance, they must not try to use a certain data item if it has not been set up properly. Otherwise, inconsistent data may be accessed, e.g. a linked list in shared memory requires proper link setting that may be corrupted when two tasks try to concatenate a new element concurrently. Errors due to the **unexpected interleaving of actions over shared data structures** are often referred to as **race conditions**. They could happen in a single core machine or only in certain conditions that tests do not cover and/or in different machines.

3.2 A race condition example

Suppose a very simple function `void inck(void)` that only contains the statement `k = k+1`. **No real-world CPU is actually able to increment k in a single, indivisible step**, at least when the code is compiled into ordinary assembly instructions.

A simplified computer based on the Von Neumann architecture performs a sequence of three distinct steps (Fig. 16):

1. Load the value of k from memory into an internal processor register; from the processor point of view, this is an external operation because it involves both the processor itself and memory. **The load operation is not destructive, that is, k retains its current value after it has been performed**.
2. Increment the value loaded from memory by one. Unlike the previous one, this operation is internal in the processor.
3. Store the new value of k into memory with an external operation involving a memory bus transaction like the first one. It is important to notice that the new value of k can be observed from outside the processor only at this point, not before. In other words, **if we look at the memory, k retains its original value until this final step has been completed**.

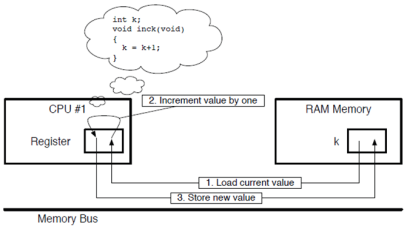

Fig. 16: Simplified representation of how the CPU increments a memory-resident variable k

The following sequence of events **may occur** (Fig. 17):

1.1. CPU #1 loads the value of k from memory and stores it into one of its registers, R1. Since k currently contains 0, R1 will also contain 0.

1.2. CPU #1 increments its register R1. The new value of R1 is therefore 1.

2.1. Now CPU #2 takes over, loads the value of k from memory, and stores it into one of its registers, R2. Since CPU #1 has not stored the updated value of R1 back to memory yet, CPU #2 still gets the value 0 from k and R2 will also contain 0

2.2. CPU #2 increments its register R2, its new value is therefore 1

2.3. CPU #2 stores the contents of R2 back to memory in order to update k, that is, it stores 1 into k.

1.3. CPU #1 does the same: it stores the contents of R1 back to memory, that is, it stores 1 into k.

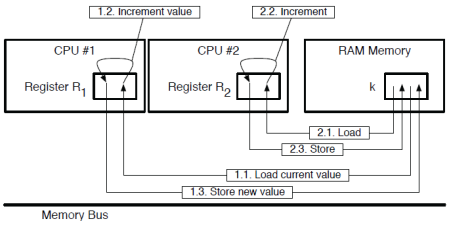

Fig. 17: Concurrently incrementing a shared variable in a careless way leads to a race condition

Looking closely at the sequence of events just described, it easy to notice that taking an (obviously correct) piece of sequential code and using it for concurrent programming in a careless way did not work as expected. In the particular example just made, the final value of k is clearly incorrect: the initial value of k was 0, two distinct processes incremented it by one, but its final value is 1 instead of 2, as it should have been. Even worse: **the result of the concurrent update is not only incorrect but it is incorrect only sometimes**. Finally, **the same problem may arise with two processes/threads on the same processor**.

3.3 The need of mutual exclusion

A general solution is to protect **critical sections**, i.e. section of code accessing shared information by means of **locks**, thus ensuring that only a task at a time is allowed to execute a section of code 'protected' by that lock.

A task that wants to access a shared object, by means of a certain critical region, must perform the following sequence of steps:

- **Acquire some sort of lock** associated with the shared object **and wait if it is not immediately available**.
- **Use the shared object**.
- **Release the lock**, so that other processes can acquire it and be able to access the same object in the future

In any case, four conditions must be satisfied in order to have an acceptable solution:

1. It must really work, that is, **it must prevent any two processes from simultaneously executing code within critical regions pertaining to the same shared object**.
2. **Any process** that is busy doing internal operations, that is, **is not currently executing within a critical region, must not prevent other processes from entering their critical regions**, if they so decide.
3. **If a process wants to enter a critical region, it must not have to wait forever to do so**. This condition guarantees that the process will eventually make progress in its execution (starvation).
4. **It must work regardless of any low-level details about the hardware or software architecture**. For example, the correctness of the solution must not depend on the number of processes in the system, the number of physical processors, or their relative speed.

3.4 A Naïve Implementation

In this example (Fig. 18), the lock variable is implemented as an integer variable called `lock` that can assume either the value 0 (no processes are currently accessing the set of shared variables associated with the lock) or 1 (one process is accessing the set of shared variables...). Before entering a critical section, a process P must:

1. Check whether the lock variable is 1. If this is the case, another process is currently accessing the set of shared variables protected by the lock. Therefore, P must wait and perform the check again later. This is done by the `while` loop shown in the figure.

2. When P eventually finds that the lock variable is 0, it breaks the while loop and is allowed to enter its critical region. Before doing this, it must set the lock variable to 1 to prevent other processes from entering, too.

When P is abandoning a critical region, it must reset the lock variable to 0. This may have two possible effects:

1. If one or more processes are already waiting to enter, one of them will find the lock variable at 0, will set it to 1 again, and will be allowed to enter its critical region.
2. If no processes are waiting to enter, the lock variable will stay at 0 for the time being until the critical region entry code is executed again.

This naive approach to the problem does not work even if we consider only two processes P_1 and P_2. This is due to the fact that, as described above, **the critical region entry code is composed of a sequence of two steps that are not executed atomically**. The following **interleaving** is therefore possible:

1.1. P_1 executes the entry code and checks the value of lock, finds that lock is 0, and immediately escapes from the while loop.

2.1. Before P_1 had the possibility of setting lock to 1, P_2 executes the entry code, too. Since the value of lock is still 0, it exits from the while loop as well.

2.2 P_1 sets lock to 1

1.2 P_2 sets lock to 1

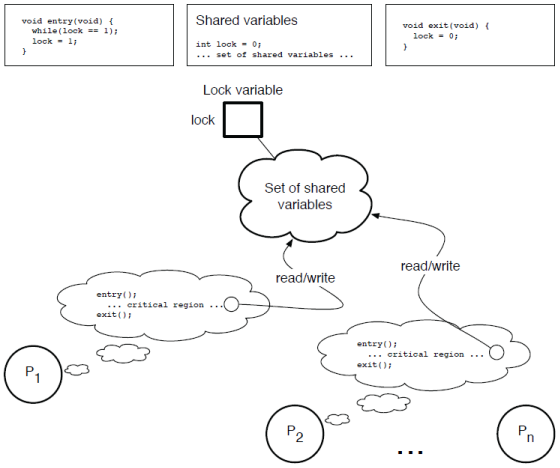

Fig. 18: Concurrently incrementing a shared variable in a careless way leads to a race condition

Both P_1 and P_2 execute their critical code and violate the mutual exclusion constraint. Using lock variables in this way, the mutual exclusion problem has merely been shifted from one "place" to another.

3.5 Hardware assisted solution

Several processor provides a `test_and_set` instruction. This instruction has the address p of a memory word as argument and **atomically** (no interleaving allowed) performs the following three steps (Fig. 19):

1. It reads the value v of the memory word pointed by p
2. It stores 1 into the memory word pointed by p
3. It puts v into a register

An alternative solution uses **XCHG instruction**, which atomically exchanges the contents of a register with the contents of a memory word. **Both instruction ensure atomicity even in a multicore architecture because they internally lock the memory bus**.

The approach just described is **correct with respect to conditions** (3.3) **1 and 2 but does not fully satisfy conditions 3 and 4**.

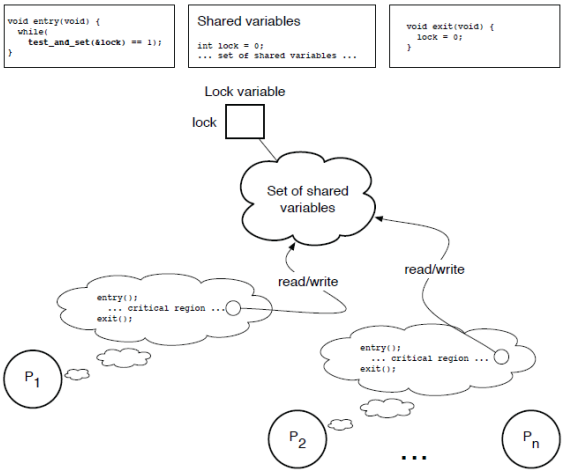

Fig. 19: Hardware-assisted lock variables work correctly

By intuition, if one of the processes is noticeably slower that the others because, for example, it is executed on a slower processor in a multiprocessor system it is placed at a disadvantage when it executes the critical region entry code. In fact, it checks the lock variable less frequently than the others, and this lowers its probability of finding lock at 0. This partially violates condition 4.
In extreme cases, the execution of the while loop may be so slow that other processes may succeed in taking turns entering and exiting their critical regions, so that the "slow" process never finds lock at 0 and is never allowed to enter its own critical region. This is in contrast with condition 3.

3.6 Software solution: the Peterson algorithm

Peterson's algorithm can be generalized to work with an arbitrary (but fixed) number of processes, for simplicity we will only consider the simplest scenario, involving only two processes P_0 and P_1 as shown in Fig. 20. It is also assumed that the memory access is **atomic**.

The critical region entry and exit functions take a parameter pid that uniquely identifies the invoking process and will be either 0 (for P_0) or 1 (for P_1). The set of shared, access-control variables becomes slightly more complicated, in particular:

- There is now one flag for each process, implemented as an array `flag[2]` of two flags. Each **flag will be 1 if the corresponding process wants to, or succeeded in entering its critical sectio**n, and 0 otherwise. The initial value of both flags is 0.
- The variable `turn` **is used to enforce the two processes to take turns if both want to enter their critical region concurrently**. Its value is a process identifier and can therefore be either 0 or 1. It is initially set to 0 to make sure it has a legitimate value even if this initial value is irrelevant to the algorithm.

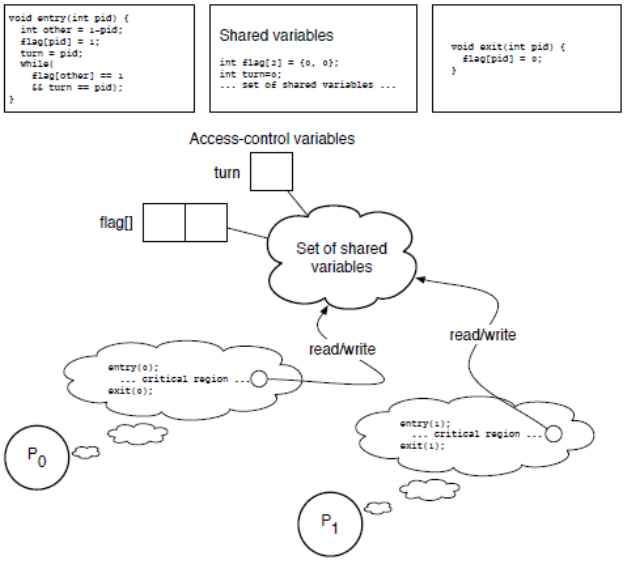

Fig. 20: Peterson's software-based mutual exclusion for two processes.

Let's start with the formal proof:

Lemma. *pid in critical section* $\implies$ *flag[pid] == 1*

Proof. pid in critical section $\implies$ pid in critical section **or** pid in entry after instruction flag[pid] = 1 $\implies$ flag[pid] == 1 (invariant) ■

Theorem. *At most one task is in critical section*

Proof. At the time the first task just enters in critical section, the other task is either:

1. not calling entry() or in any case before instruction turn = pid
2. calling entry and after instruction turn = pid

In case 1 if the other task is enters the while loop it finds the condition false: flag[other] = 1 (lemma) and turn == pid (turn is not changed by the other task in critical section).

In case 2 turn == pid, otherwise the first task would not have entered the critical section. For the same reason as before, the while loop finds the condition false. ■

To better understand the algorithm, let's start by considering the case when the two processes do not execute the critical region entry code concurrently but sequentially and P_0 executes first:

- P_0 sets its own flag, `flag[0]`, to 1. It should be noted that P_0 works on `flag[0]` because `pid` is 0 in the instance of `enter()` it invokes.
- It sets `turn` to its own process identifier `pid`, that is, 0.
- It evaluates the predicate of the `while` loop. In this case, the righthand part of the predicate is true (because `turn` has just been set to `pid`), but the left-hand part is false because the other process is not currently trying to enter its critical section.

As a consequence, P_0 immediately abandons the `while` loop and is granted access to its critical section. The final state of the shared variables after the execution of `enter(0)` is:

- `flag[0]` == 1 (it has just been set by P_0)
- `flag[1]` == 0 (P_1 is not trying to enter its critical region)
- `turn` == 0 (P_0 has been the last process to start executing `enter()`)

If, at this point, P_1 tries to enter its critical region by executing `enter(1)`, the following sequence of events take place:

1. P_1 sets its own flag, `flag[1]`, to 1.
2. It sets `turn` to its own process identifier, 1.
3. It evaluates the predicate of the `while` loop. Both parts of the predicate are true because:
   - The assignment `other = 1-pid` implies that the value of other represents the process identifier of the "other" process. Therefore, `flag[other]` refers to `flag[0]`, the flag of P_0, and this flag is currently set to 1.
   - The right-hand part of the predicate, `turn == pid`, is also true because turn has just been set this way by P_1, and P_0 does not modify it in any way.

P_1 is therefore trapped in the while loop and will stay there until P_0 exits its critical region and invokes `exit(0)`, setting `flag[0]` back to 0. In turn, this will make the left-hand part of the predicate being evaluated by P_1 false, break its busy waiting loop, and allow P_1 to execute its critical region. After that, P_0 cannot enter its critical region again because it will be trapped in `enter(0)`.

We showed that the mutual exclusion algorithm works satisfactorily when P_0 and P_1 execute `enter()` sequentially. Now, we must convince ourselves that, **for every possible interleaving of the two, concurrent executions of `enter()`, the algorithm still works as intended**.

For what concerns the first two statements executed by P_0 and P_1, that is, `flag[0] = 1` and `flag[1] = 1`, it is easy to see that the result does not depend on the execution order because they operate on two distinct variables. In any case, after both statements have been executed, both flags will be set to 1.

On the contrary, the second pair of statements, `turn = 0` and `turn = 1`, respectively, work on the same variable turn. **The result will therefore depend on the execution order but, thanks to the memory access atomicity taken for granted at the single-variable level, the final value of turn will be either 0 or 1**, and not anything else, **even if both processes are modifying it concurrently**. The final value of `turn` only depends on which process executed its assignment **last**, and represents the identifier of that process.

Let us now consider what happens when both processes evaluate the predicate of their `while` loop:

- The left-hand part of the predicate has no effect on the overall outcome of the algorithm because both `flag[0]` and `flag[1]` have been set to one.
- The right-hand part of the predicate will be true for **one and only one** process. It will be true for at least one process because `turn` will be either 0 or 1. In addition, it cannot be true for both processes because `turn` cannot assume two different values at once and no processes can further modify it.

In summary, either P_0 or P_1 (but not both) will be trapped in the `while` loop, whereas the other process will be allowed to enter into its critical region. Due to our considerations about the value of `turn`, we can also conclude that the process that set turn last will be trapped, whereas the other will proceed. As before, the `while` loop executed by the trapped process will be broken when the other process resets its flag to 0 by invoking its critical region exit code.

The software solution is working with respect to conditions 1 and 2 (3.3). Moreover, **with respect to condition 3 and 4, it works better than the hardware-assisted one** discussed in 3.5:

- The slower process is no longer systematically put at a disadvantage. Assuming that P_0 is slower than P_1, it is still true that P_1 may initially overcome P_0 if both processes execute the critical region entry code concurrently. However, when P_1 exits from its critical region and then tries to immediately reenter it, it can no longer overcome P_0.
  When P_1 is about to evaluate its `while` loop predicate for the second time, the value of turn will in fact be 1 (because P_1 set it last), and both flags will be set. Under these conditions, the predicate will be true, and P_1 will be trapped in the loop. At the same time, as soon as `turn` has been set to 1 by P_1, P_0 will be allowed to proceed regardless of its speed because its `while` loop predicate becomes false and stays this way.
- For the same reason, and due to the symmetry of the code, if both processes repeatedly contend against each other for access to their critical region, they will take turns at entering them, so that no process can systematically overcome the other. This property also implies that, if a process wants to enter its critical region, it will succeed within a finite amount of time, that is, at the most the time the other process spends in its critical region.

3.7 From Active to Passive Wait

The solutions seen so far imply a **repeated check of a shared variable**. This mechanism is similar to the polling I/O synchronization, but may waste processor cycles. Moreover, when tasks share the same processor, **the wait loop may affect the execution of the critical section leading to unbounded access time to the critical section**.

A better solution would imply **the OS removing the processor from a task trying to access a critical section already locked until it becomes free**. This implies making a request to the OS for entering and exiting a critical section.

In a single processor system OS data structures will replace the shared variables, and **any task (process or thread) willing to access a critical section will be put in wait state if another task already gained access**. When a task exits the critical section, the OS shall be informed, and tasks waiting for that critical section will be made ready again and eventually they will try again to enter the critical section.

The implementation of the OS code **for multicore systems is more complicated because the OS code itself is distributed**. In this case **HW specific locking instruction** such as atomic exchanges shall be used internally in the OS as well as polling (**spinlock**)

In Fig. 21 we see an example of indefinite wait that may happen if two processes share the same processor: P_0 is stuck because it is waiting to acquire the lock from P_1, a lower priority process. Since P_1 priority is lower it will not be able to gain access to the processor (which is being occupied by P_0) and so, it will never be able to release the lock.

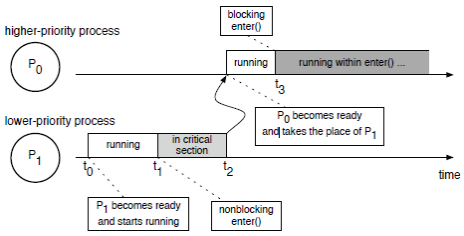

Fig. 21: Busy wait and fixed-priority assignment may interfere with each other, leading to an unbounded priority inversion.

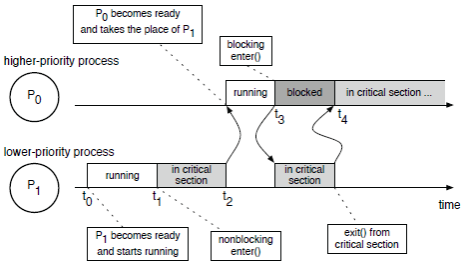

Fig. 22: Using passive wait instead of busy wait solves the unbounded priority inversion problem in simple cases.

The problem is solved, as shown in Fig. 22, because P_0 is put in wait state by the OS, thus giving the chance to P_1 to gain access to the processor and release the lock. When P_1 releases the lock, the OS will make P_0 ready again and it will try to acquire the lock again.

3.8 Semaphores

A semaphore is an object that comprises two abstract items of information:

1. **A non-negative integer value**
2. **A queue of processes passively waiting on the semaphore**

Upon initialization, a semaphore acquires an initial value specified by the programmer, and its queue is initially empty. **Neither the value nor the queue associated with a semaphore can be read or written directly after initialization**. On the contrary, **the only way to interact with a semaphore is through the following two primitives that are assumed to be executed atomically**:

1. `P(s)`, when invoked on semaphore s, checks whether the value of the semaphore is (strictly) greater than zero. If this is the case, it decrements the value by one and returns to the caller without blocking. Otherwise, it puts the calling process into the queue associated with the semaphore and blocks it by moving it into the Wait (Blocked) state of the process state diagram (Fig. 23).
2. `V(s)`, when invoked on semaphore s, checks whether the queue associated with that semaphore is empty or not. If the queue is empty, it increments the value of the semaphore by one. Otherwise, it picks one of the blocked processes found in the queue and makes it ready for execution again by moving in into the Ready state of the process state diagram.

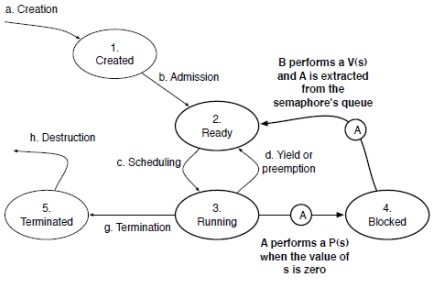

Fig. 23: Process State Diagram transitions induced by the semaphore primitives `P()` and `V()`

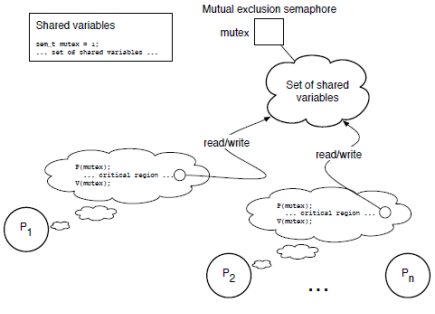

Fig. 24: A semaphore can be used to enforce mutual exclusion

In Fig. 24 we see an example of how semaphores can be used to protect a critical section.
Conditions (3.3) 1 and 2 are clearly satisfied, while condition 3 and four are satisfied only if the semaphore queuing policy is first-in first-out (FIFO), so that `V()` always wakes up the process that has been waiting on the semaphore for the longest time.

Beside mutual exclusion, semaphores are also useful for **condition synchronization**, that is, when we want to block a process until a certain event occurs or a certain condition is fulfilled.

For example, considering the producers-consumers problem, it may be desirable **block any consumer that wants to get a data item from the shared buffer when the buffer is completely empty**, instead of raising an error indication. A blocked consumer must be unblocked as soon as a producer puts a new data item into the buffer.
Symmetrically, we might also want to **block a producer when it tries to put more data into a buffer that is already completely full**.

In order to do this, we need **one semaphore for each synchronization condition that the concurrent program must respect**. In this case, we have two conditions, and hence we need two semaphores:

1. The semaphore `empty` counts how many empty elements there are in the buffer. Its initial value is N because the buffer is completely empty at the beginning. Producers perform a `P(empty)` operation before putting more data into the buffer to update count and possibly block themselves, if there is no empty space in the buffer. After removing one data item from the buffer, consumers perform a `V(empty)` to either unblock one waiting producer or increment the count of empty elements.
2. Symmetrically, the semaphore full counts how many full elements there are in the buffer. Its initial value is 0 because there is no data in the buffer at the beginning. Consumers perform a `P(full)` before removing a data item from the buffer, and producers perform a `V(full)` after storing an additional data item into the buffer.

The corresponding code for the producers-consumers problem is shown in Fig. 25.

```
void prod(int d) {
  P(empty);
  P(mutex);
    buf[in] = d;
    in = (in+1) % N;
  V(mutex);
  V(full);
}
```

Macros/Shared variables

```
#define N 8
int buf[N];
int in=0, out=0;
sem_t mutex=1;
sem_t empty=N;
sem_t full=0;
```

```
int cons(void) {
  int c;
  P(full);
  P(mutex);
    c = buf[out];
    out = (out+1) % N;
  V(mutex);
  V(empty);
  return c;
}
```

Fig. 25: Producers–consumers problem solved with mutual exclusion and condition synchronization semaphores

In summary, even if semaphores are all the same, they can be used in two very different ways, which should not be confused:

1. A **mutual exclusion** semaphore, like mutex in the example, is used to prevent more than one process from executing within a set of critical regions pertaining to the same set of shared data. The use of a mutual exclusion semaphore is quite stereotyped: its initial value is always 1, and `P()` and `V()` are placed, like brackets, around the critical regions code.
2. A **Condition synchronization** semaphore, such as `empty` and `full` in the example, is used to ensure that certain sequences of events do or do not occur. In this particular case, we are using them to prevent a producer from storing data into a full buffer, or a consumer from getting data from an empty buffer.

3.9 Monitors

Semaphore represent a rather 'low level' approach for synchronization and lead to complicated solutions with increased risk of bugs. To address these issues, a higher-level and more structured interprocess communication mechanism, called **monitor**, was proposed by Brinch Hansen and Hoare.
It is interesting to note that, even if these proposals date back to the early '70s, they were already based on concepts that are common nowadays and known as object-oriented programming.

In its most basic form, a monitor is a composite object and contains:

- **a set of shared data**;
- **a set of methods that operate on them**.

With respect to its components, a monitor guarantees the following two main properties:

- **Information hiding**, because **the set of shared data defined in the monitor is accessible only through the monitor methods** and cannot be manipulated directly from the outside. **Monitor methods are not hidden and can be freely invoked from outside the monitor**.
- **Mutual exclusion among monitor methods**, that is, the monitor implementation, must guarantee that **only one process will be actively executing within any monitor method at any given instant**.

No direct support in C/C++ for monitors, but the Java synchronized method closely reflect the monitor approach.

We have seen that semaphores can be also used for synchronization. This is achieved also in monitors by means of **condition variables**. Condition variables can be used only by the methods of the monitor they belong to, and cannot be referenced in any way from outside the monitor boundary. They are therefore hidden exactly like the monitor's shared data.

The following two primitives are defined on a condition variable c:

- `wait(c)` blocks the invoking process and releases the monitor in a single, **atomic** action.
- `signal(c)` wakes up one of the processes blocked on c; it has **no effect if no processes are blocked on c**.

The informal reasoning behind the primitives is that, if a process starts executing a monitor method and then discovers that it cannot finish its work immediately, it invokes wait on a certain condition variable. In this way, it blocks and allows other processes to enter the monitor and perform their job. When one of those processes, usually by inspecting the monitor's shared data, detects that the first process can eventually continue, it calls signal on the same condition variable.

`wait()` and `signal()` are implemented as java object methods and in this case the condition variable is the object instance itself. Condition variables are also provided by POSIX for thread synchronization, but the Monitor abstraction is not pushed further, and mutexes will be defined to handle critical sections.

Let's consider the case shown in Fig. 26, the following sequence of events involving two processes A and B may happen:

1. Taking for granted that the monitor is initially free, that is, no processes are executing any of its methods, process A enters the monitor by calling one of its methods, and then blocks by means of a `wait(c)`.
2. At this point, the monitor is free again, and hence, another process B is allowed to execute within the monitor by invoking one of its methods. There is no race condition because process A is still blocked.
3. During its execution within the monitor, B may invoke `signal(c)` to wake up process A.

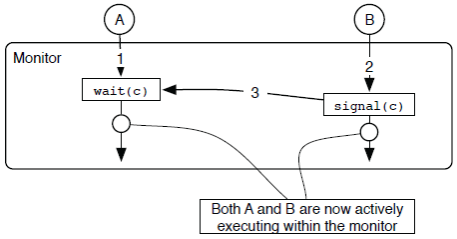

Fig. 26: After a `wait/signal` sequence on a condition variable, there is a race condition that must be adequately addressed

After this sequence of events, both **A and B are actively executing within the monitor**. Hence, they are allowed to manipulate its shared data concurrently in an uncontrolled way. In other words, **the mutual exclusion property of monitors has just been violated**.

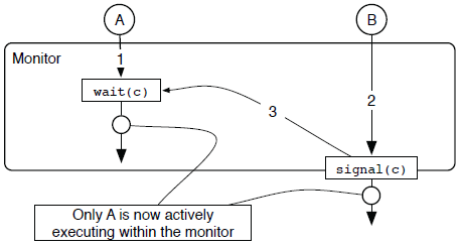

Fig. 27: An appropriate constraint on the placement of `signal` in the monitor methods solves the race condition issue after a `wait/signal` sequence

It is possible to work around the issue by constraining the placement of `signal` within the monitor methods: in particular, if a `signal` is ever invoked by a monitor method, **it must be its last action**, and implicitly makes the executing process exit from the monitor. As already discussed before, a simple scan of the source code is enough to detect any violation of the constraint. In this way, only process A will be executing within the monitor after a `wait/signal` sequence, as shown in Fig. 27, because the `signal` must necessarily be placed right at the monitor boundary. As a consequence, **process B will indeed keep running concurrently with A, but outside the monitor, so that no race condition occurs**. Of course, it can also be shown that the constraint just described solves the problem in general, and not only in this specific case.

The main advantage of this solution is that it is quite simple and efficient to implement. On the other hand, it leaves to the programmer the duty of designing the monitor methods so that `signal` only appears in the right places.

Another approach, by the POSIX standard, is based on the fact that the process just awakened after a `wait`, like process A in Fig. 28, **must acquire the monitor's mutual exclusion lock before proceeding, whereas the signaling process** (B in the figure) **continues immediately**. This approach **solves the race condition by postponing the signaled process, instead of the signaling one**. When the signaling process B eventually leaves the monitor or blocks in a `wait`, one of the processes waiting to start or resume executing in the monitor is chosen for execution. This may be:

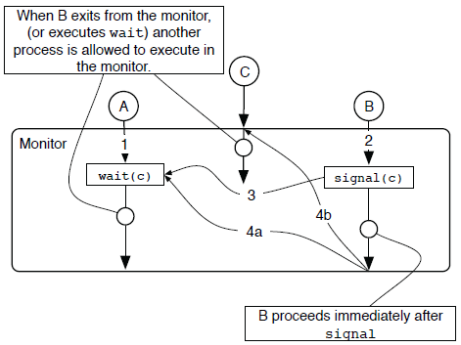

Fig. 28: The POSIX approach to eliminate the race condition after a `wait/signal` sequence is to force the process just awakened from a `wait` to reacquire the monitor mutual exclusion lock before proceeding

- one process waiting to reacquire the mutual exclusion lock after being awakened from a `wait` (process A and step 4a); or,
- one process waiting to enter the monitor from the outside (process C and step 4b).

The most important side effect of this approach from the practical standpoint is that, when process A waits for a condition and then process B signals that the condition has been fulfilled, process A cannot be 100% sure that the condition it has been waiting for will still be true when it will eventually resume executing in the monitor. It is quite possible, in fact, that another process C was able to enter the monitor in the meantime and, by altering the monitor's shared data, make the condition false again.

To conclude the description, Fig. 29 shows how the producers–consumers problem can be solved by means of the simplest kind of monitor presented so far. Unlike the previous examples, this one is written in "pseudo C" because the C programming language, by itself, does not support monitors. The fake keyword `monitor` introduces a monitor. The monitor's shared data and methods are syntactically grouped together by means of a pair of braces. Within the monitor, the keyword `condition` defines a condition variable.

The main differences with respect to the semaphore-based solution are:

```
#define N 8
monitor ProducersConsumers
{
    int buf[N];
    int in = 0, out = 0;
    condition full, empty;
    int count = 0;

    void produce(int v)
    {
        if(count == N)  wait(empty);
        buf[in] = v;
        in = (in + 1) % N;
        count = count + 1;
        if(count == 1)  signal(full);
    }

    int consume(void)
    {
        int v;
        if(count == 0)  wait(full);
        v = buf[out];
        out = (out + 1) % N;
        count = count - 1;
        if(count == N-1)  signal(empty);
        return v;
    }
}
```

Fig. 29: Producers–consumers problem solved by means of a monitor

- The mutual exclusion semaphore `mutex` is no longer needed because the monitor construct already guarantees mutual exclusion among monitor methods.
- The two synchronization semaphores `empty` and `full` have been replaced by two condition variables with the same name. Indeed, their role is still the same: to make producers wait when the buffer is completely full, and make consumers wait when the buffer is completely empty.
- In the semaphore-based solution, the value of the synchronization semaphores represented the number of empty and full elements in the buffer. Since condition variables have no memory, and thus have no value at all, the monitor-based solution keeps that count in the shared variable `count`.
- Unlike in the previous solution, all `wait` primitives are executed conditionally, that is, only when the invoking process must certainly wait. For example, the producer's `wait` is preceded by an `if` statement checking whether `count` is equal to N or not, so that `wait` is executed only when the buffer is completely full. The same is also true for `signal`, and is due to the semantics differences between the semaphore and the condition variable primitives.

4 Interprocess Communication Based on Message Passing

4.1 Message Passing Synchronization

It represents an alternative way to shared memory solutions. It is a **mandatory solution when actors reside on different machines not sharing memory**.
In its simplest, and most abstract, form a message-passing mechanism involves two basic primitives:

- a **send** primitive, which sends a certain amount of information, called a message, to another process;
- a **receive** primitive, which allows a process to **block waiting** for a message to be sent by another process, and then retrieve its contents.

Even if this definition still lacks many important details it is already clear that the most apparent effect of message passing primitives is to transfer a certain amount of information from the sending process to the receiving one.
At the same time, **the arrival of a message to a process also represents a synchronization signal because it allows the process to proceed after a blocking receive**.

The last important requirement of a satisfactory interprocess communication mechanism, **mutual exclusion, is not a concern here because messages are never shared among processes, and their ownership is passed from the sender to the receiver when the message is transferred**. In other words, the mechanism works as if the **messages were instantaneously copied from the sender to the receiver** (even if real-world message passing systems do their best to avoid actually copying a message for performance reasons).

There are a few main design choices to be made when implementing a message passing mechanism:

1. For a sender, how to identify the intended recipient of a message. Symmetrically, for a receiver, how to specify from which other processes it is interested in receiving messages. In more abstract terms, **how a process naming scheme must be defined**.
   a. how the send and receive primitives are associated to each other;
   b. their symmetry (or asymmetry).
2. The **synchronization model**, that is, under what circumstances communicating processes shall be blocked, and for how long, when they are engaged in message passing.
3. How to make a **message content consistent** (endianity an pointers)

4.2 Direct vs indirect naming

The most straightforward approach is **for the sending process to name the receiver directly**. For example, in an IP communication the sender shall **specify the Internet Protocol (IP) address and port of the intended receiver**.

Considering the receiver, direct naming is not so straightforward:

- Normally the receiver provided a service that may be accessed by different clients
- After a connection has been established sender and receiver exchange messages and can therefore swap their roles
- Both must adhere to a **communication protocol** in order to achieve a consistent communication

Indirect naming means that **the sender does not address the receiver explicitly**. For example in a multicast communication, a set of receivers may receive the message sent by the server, that may also be unaware of the identity of the recipients.

Publish-Subscribe is a common pattern in this context:

- A **publisher** offers some kind of service
- A **subscriber** can subscribe for a given service and shall receive messages issued by the publisher
- More than one subscriber may be registered and the publisher may be unaware of this, especially when association between publishers and subscribers is implemented by a separate **broker**.

4.3 Message Synchronization

The receiver normally needs to synchronize to the message receipt: asynchronous receive shall define a callback to be called asynchronously when a message has been received (Fig. 30).
Normally `send()` call returns soon, unless internal message queuing reaches a given limit.

In rendezvous schema, the sender synchronizes with the message reception (Fig. 31).
Normally rendezvous schema are implemented with a **acknowledge message sent back by the receiver**. The communication protocol specifies the messages to be sent and received in order to achieve the required synchronization.

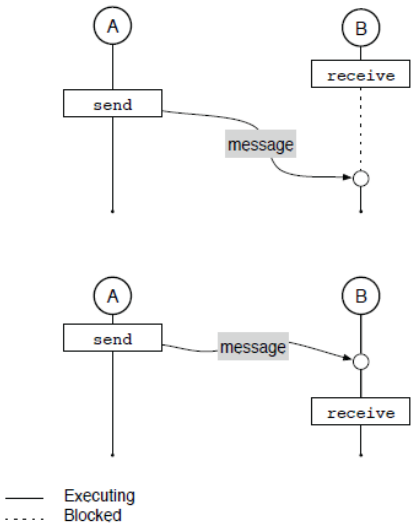

Fig. 30: Asynchronous message transfer. The sender is never blocked by send even if the receiver is not ready for reception

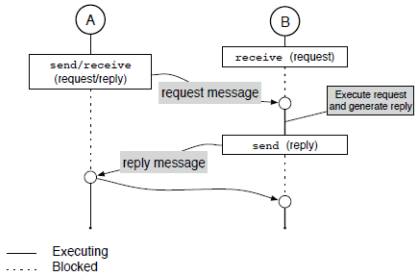

Fig. 31: Remote invocation message transfer, or extended rendezvous. The sender is blocked until it gets a reply from the receiver. Symmetrically, the receiver is blocked until the reply has successfully reached the original sender

4.4 Message Buffering

Very often the `send()` routine will return before the message has been received. If communication occurs within the same computer, this means **temporarily storing exchanged packets in memory** (managed by the OS).
If communication involves different computers, **packets may be stored in any point of the path between the sender and the receiver**.

Considering network communication, messages are often buffered in routers before being dispatched at the right port , based on the message header and the routing information (store and forward).
Other routers start dispatching the message as soon as the header has been received and the destination can be computed (cut-through).

Message buffering is normally transparent to the sender and the receiver and therefore **no assumption can be done about message transfer time**.
If messages are used **for synchronization, the use of Acknowledge return message is required**.

Having a **large buffer between the sender and the receiver decouples the two processes and, on average, makes them less sensitive to any variation in execution and message passing speed**. Therefore **it increases the likelihood of executing them concurrently without waiting for one another**.
On the other hand, **the interposition of a buffer increases the message transfer delay and makes it less predictable**.

For some synchronization models, **the amount of buffer space required** at any given time to fulfill the model **may depend on the processes' behavior and can be very difficult to predict**. For the **purely asynchronous model**, the maximum amount of buffer space to be provided by the system **may even be unbounded in some extreme cases**. This happens, for instance, when **the sender is faster than the receiver so that it systematically produces more messages than the receiver is able to consume**.

4.5 (Un-)Reliable message transfer vs real-time communication

The transmission of a message is subject to several possible **communication errors**.
Effects of communication errors can be:

- Message **corruption**
- Messages **loss**
- Messages **duplicated**

Reasons for communication errors can be:

- **Transmission errors** (e.g. bit flips, cosmic rays, etc. . .)
- **Messages discarded in internal router buffers when traffic is too high**

Reliability can be granted or not depending on the communication protocol adopted:

- **TCP/IP guarantees reliable delivery and non-corrupted messages**
- **UDP guarantees only that received messages are not corrupted** (checksum handleld in the protocol) **but no guarantee is given about message delivery**, i.e. messages can be lost. **For Local Area Networks (LANs), the reason for the loss of a message is exclusively due to internal router buffer overruns**.

Real-time communication requires that a message shall be successfully received within a given amount of time (deadline).
Reliability in communication requires detecting whether messages are lost, normally assigning an increasing timestamp value to each message so that missed messages can be detected checking the timestamp **and handling the message lost**, normally by requesting the missed message

However, this added reliability layer introduces an **unbounded delay even if the transmission time is bounded**, therefore breaking real-time requirements.

On the other side, loss of messages cannot often be avoided for sure; this has to be taken into account in communication for real-time distributed systems.
Sometimes a (limited) loss of messages can be tolerated, e.g. when then system is performing closed loop control: missing messages can be considered an added noise in sensor readout or actuator command and may be tolerated provided the overall controller introduces a stability margin large enough.

Moreover, **for LAN communication, by insulating the system from external (unpredictable) traffic** it is in general possible to make sure that **no internal buffer overrun occurs in the involved routers**.
For this reason, **UDP is often adopted as communication protocol in real-time systems**.

4.6 Message content consistency

Not always the bytes forming a message bring **consistent information**, especially when the receiver resides on a different machine.
A common error is due to the **endianity** that may be different between the machines involved in communication:

- char arrays always work, but **multibyte numbers can be interpreted differently (big/little endian issue)**
- **the IP socket layer defines a default endianity (big endian) and it is up to the sender/receiver properly swap bytes**.

Another issue is related to the **usage of pointers in data structures** because:

- **This affects also communication on the same machine because different processes have a different address space** (virtual addresses)
- **Pointers can be safely exchanged** when communication **among threads of the same process** occurs

A common practice **when sending complex data structure is to serialized them in a sequence of bytes**: if the structure contains **internal pointers** (e.g. a linked queue) the pointers **are normally replaced by an offset from the first byte of the buffer in serialization and replaced by pointers in deserialization**.

4.7 Producer-Consumer implemented with messages

A solution to the producer-consumer problem can be implemented using messages is shown in Fig. 32.

When the producer P wants to send a certain data item `d`, it calls the function `prod` with `d` as argument to perform the following operations:

- Convert the data item to be sent, `d`, from the host representation to a neutral representation that both the sender and the receiver understand. This operation is represented in the code as a call to the abstract function `host_to_neutral()`.
- Send the message to the consumer C. A direct, symmetric naming scheme has been adopted in the example, and hence the send primitive names the intended receiver directly with its first argument. The next two arguments are the memory address of the message to be sent and its size.

On the other side, the consumer C invokes the function `cons()` whenever it is ready to retrieve a message:

- The function waits until a message arrives, by invoking the `recv` message-passing primitive. Since the naming scheme is direct and symmetric, the first argument of `recv` identifies the intended sender of the message, that is, P. The next two arguments locate a memory buffer in which `recv` is expected to store the received message and its size.
- Then, the data item found in the message just received is converted to the host representation by means of the function `neutral_to_host()`. The result `d` is returned to the caller.

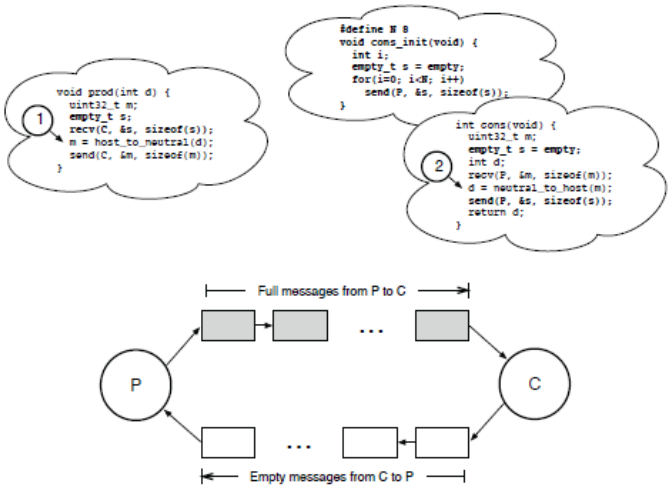

Fig. 32: A solution to the producer-consumer problem based on asynchronous message passing. In this case, the synchronization condition for the producer P is provided explicitly rather than implicitly

If only asynchronous message passing is available, assuming that the message-passing mechanism can successfully buffer at least N messages, a second flow of empty messages that goes from C to P is used to carry synchronization information (the data type `empty_t` represents an empty message).
In particular:

- The consumer C sends an empty message to P after retrieving a message from P itself.
- The producer P waits for an empty message from the consumer C before sending its own message to it
- By means of the initialization function `cons_init()`, the consumer injects N empty messages into the system at startup.

At startup, there are therefore N empty messages. As the system evolves, the total number of empty plus full messages is constant and equal to N because one empty (full) message is sent whenever a full (empty) message is retrieved. The only transient exception happens when the producer or the consumer are executing at locations 1 and 2, respectively. In that case, the total number of messages can be N-1 or N-2 because one or two messages may have been received by P and/or C and have not been sent back yet.

In this way, C still waits if there is no full message from P at the moment, as before. In addition, P also waits if there is no empty message from C. The total number of messages being constant, this also means that P already sent N full messages that have not yet been handled by C.

4.8 The programming model of TCP/IP

In the programming model of TCP/IP, **a socket must be established between the client and the server**. Once the socket has been established, client and servers can communicate, i.e. send and receive messages, that are ensued to be correctly delivered.
Both client and server must adhere to a given protocol for meaningful communication: e.g. if both issue a `receive()` call, they+ stop forever.

In order to establish a connection (socket), **the client must specify the Internet Protocol (IP) address and the port number of the target server**.
The server instead is not aware of potential client until a connection has been established. The server therefore issues an `accept()` command that shall return when a client requested the establishment of a communication. At this point both client and server will receive a socket to be used for further `read()` and `write()` operations.

In order to be able to serve more than one client concurrently, the server shall start a new thread/process (the latter being more safe as explained in 2.4) every time the `accept()` call returns, passing the returned socket to the task.

4.9 The programming model of UDP

Unlike TCP/IP, **the programming model is message oriented, i.e. an UDP datagram** (normally limited to 64 KB) **can be sent and received**.

Still in this case **a socket must be created and used for sending and receiving datagrams**, however in this case **the successful socket creation does not imply the establishment of a connection** (in this case the socket is in 'unconnected' state).

Messages are sent via `sendto()` routine whose arguments shall **specify the IP address and port number of the receiver**.
The task that wants to receive a datagram, it will call `recv()` or `recvfrom()`, after binding the socket to a port number:

- `recv()` shall return when a message has been received by any sender addressing this IP and port.
- `recvfrom()` shall return when a message from the specified sender has been received.

UDP allows also the implementation of **publish-subscribe pattern using UDP multicast**: in this case it is possible to register to a UDP multicast address, so that every message sent to this multicast address shall be received by all registered listeners.

5 Deadlocks

5.1 A Simple Example

Let's consider the example in Fig. 33, which differs from the producer-consumer example previously seen in 3.8 (Fig. 25) because the two semaphore operations in red have been exchanged.

Suppose that `empty` semaphore is zero (no free slots) and that a producer tries to produce a new item while no consumer is active.
When the producer is blocked waiting for `P(empty)` no consumer can consume data item since they should first the mutex, which is blocked by the waiting producer and therefore cannot issue `V(empty)`.

The system hangs forever → **Deadlock**.
Moreover this may never happen during testing, e.g. if the buffer is never full.

```
void prod(int d) {
    P(mutex);
    P(empty);

    buf[in] = d;
    in = (in+1) % N;

    V(mutex);
    V(full);
}
```

```
#define N 8
int buf[N];
int in=0, out=0;
sem_t mutex=1;
sem_t empty=N;
sem_t full=0;
```

```
int cons(void) {
    int c;

    P(full);
    P(mutex);

    c = buf[out];
    out = (out+1) % N;

    V(mutex);
    V(empty);

    return c;
}
```

Fig. 33: Deadlock example in the producer-consumer problem solved with semaphores

5.2 Formal Definition

A deadlock can be defined formally as a situation in which **a set of processes passively waits for an event that can be triggered only by another process in the same set**.
More specifically, when dealing with resources, there is a deadlock **when all processes in a set are waiting for some resources previously allocated to other processes in the same set**.
A deadlock has therefore two kinds of adverse consequences:

- The processes involved in the deadlock will **no longer make any progress in their execution**, that is, they will wait forever.
- **Any resource allocated to them will never be available to other processes in the system again**.

The following conditions are **required** for a deadlock to occur:

- **Mutual exclusion**: **each resource can be assigned**, and used by, **at most one process at a time**. If any process requests a resource currently assigned to another process, it must wait.
- **Hold and wait**: for a deadlock to occur **the process involved in the deadlock must have successfully obtained at least one resource in the past and have not released it yet**, so that they hold those resources and then wait for additional resources.
- **Non-preemption**: **any resource involved in a deadlock cannot be taken away from the process it has been assigned to without its consent**, that is, **unless the process voluntarily releases it**.
- **Circular wait**: **the processes and resources involved in a deadlock can be arranged in a circular chain**, so that the first process waits for a resource assigned to the second one, the second process waits for a resource assigned to the third one, and so on up to the last process, which is waiting for a resource assigned to the first one.

5.3 The Resource Allocation Graph

A resource allocation graph (Fig. 34) is a directed graph with two kind of nodes and two kind of arcs.

The two kinds of nodes represent the processes and resources of interest, respectively. In the most common notation:

- **Processes are shown as circles**.
- **Resources are shown as squares**.

The two kinds of arcs express the request and ownership relations between processes and resources, in particular:

- **An arc directed from a process to a resource indicates that the process is waiting for the resource**.
- **An arc directed from a resource to a process denotes that the resource is currently assigned to, or owned by, the process**.

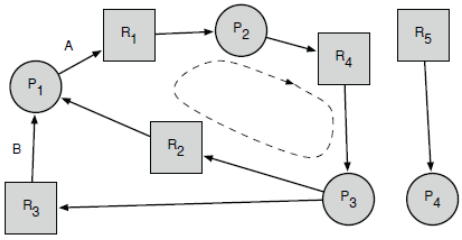

Fig. 34: A simple resource allocation graph, indicating a deadlock

If a cycle is found, then there is a deadlock in the system, and the deadlock involves precisely the set of processes and resources belonging to the cycle.

5.4 Deadlock Prevention

The general idea behind all deadlock prevention techniques is to prevent deadlocks from occurring by **making sure that (at least) one of the individually necessary conditions** presented in 5.2 **can never be satisfied in the system**.

The **mutual exclusion** condition can be attacked by **allowing multiple processes to use the same resource concurrently without waiting when the resource has been assigned to another process**.

- This is **not possible for** several resources such as **a lock protecting a critical section**.
- Other resources such as printers, can be made shareable letting another task (a spooler) collect and enqueue requests for that resource.

The **hold and wait** condition is actually made of two parts that can be considered separately:

- **The wait part can be invalidated by making sure that no processes will ever wait for a resource**: one simple way to achieve this is to **force processes to request all the resources they may possibly need during execution all together and right at the beginning of their execution** (all or nothing).
- **The hold part may be invalidated by constraining processes to release all the resources they own before requesting new ones**: the new set of resources being requested can include, of course, some of the old ones if they are still needed, but **the process must accept a temporary loss of resource ownership** anyway. If a stateful resource, like for instance a printer, is lost and then reacquired, the resource state will be lost too.

The **non-preemption** condition can be attacked by **making provision for a resource preemption mechanism, that is a way to forcibly take away a resource from a process against its will**.
Like for the mutual exclusion condition, the difficulty of doing this heavily depends on the kind of resource to be handled:

- Preempting a resource such as a processor is quite easy, and is usually done (by means of a context switch) by most operating systems.
- Preempting an operation already in progress in favor of another one entails losing a certain amount of work.

The **circular wait** condition can be invalidated by **imposing a total ordering on all resource classes and imposing that, by design, all processes follow that order when they request resources**. In other words, an integer-valued function $f(R_i)$ is defined on all resource classes R_i and it has a unique value for each class. Then, if a process already owns a certain resource R_j , it can request an additional resource belonging to class R_k if, and only if, $f(R_k) > f(R_j)$.

Deadlock prevention implies putting into effect and enforcing appropriate design or implementation rules, or constraints. This may **complicate life to programmers and reduces the flexibility in resource**

utilization (e.g. by imposing an ordering in the resources to be used). Such constraints can be relaxed when relying on **Deadlock Avoidance** schemas, that is, **avoiding a resource allocation in case this may lead to a deadlock condition**.

5.5 Deadlock Avoidance

Unlike deadlock prevention algorithms, **deadlock avoidance algorithms take action later, while the system is running rather that during system design**. As in many other cases, **this choice involves a trade-off**:

- on the one hand, it makes programmers happier and more productive because they are **no longer constrained to obey any deadlock-related design rule**;
- on the other hand, **it entails a certain amount of overhead**.

The general idea of any deadlock avoidance algorthm is to **check resource allocation requests, as they come from processes, and determine whether they are safe or unsafe for what concerns deadlocks**:

- **If a request is deemed to be unsafe, it is postponed, even if the resources being requested are free**.
- **The postponed request will be reconsidered in the future, and eventually granted if and when its safety can be proved**.

Usually, deadlock avoidance algorithms also need a certain amount of **preliminary information about process behavior** to work properly.
The Banker's algorithm can be used to check whether granting requested resources is safe.

5.6 The Banker's Algorithm

First let's describe the used structures in the algorithm:

A (column) vector **t**, representing the **total number of resources of each class** (e.g. mutex) **initially available in the system**.

$$\mathbf{t} = \begin{pmatrix} t_1 \\ \vdots \\ t_m \end{pmatrix}$$

A matrix **C**, with m rows and n columns, that is, a column for each process and a row for each resource class, which holds the **current resource allocation state**.

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix}$$

A matrix **X**, also with m rows and n columns, containing information about the **maximum number of resources that each process may possibly require, for each resource class, during its whole life**. Note that this information must be provided by the programmer (no implementation in Linux), e.g. a supervisor class.

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix}$$

An auxiliary matrix **N**, representing the **worst-case future resource needs of the processes**: $\mathbf{N} = \mathbf{X} - \mathbf{C}$

A vector **r**, representing the **resources remaining in the system at any given time**.

$$\mathbf{r} = \begin{pmatrix} r_1 \\ \vdots \\ r_m \end{pmatrix}$$

A vector $\mathbf{q_j}$, representing the **request coming from the j-th process** (e.g.: $q_{1j} = 1, q_{mj} = 0$, means that process j is requesting resource class 1 and not m).

$$\mathbf{q_j} = \begin{pmatrix} q_{1j} \\ \vdots \\ q_{mj} \end{pmatrix}$$

Let's now describe the steps of the algorithm:

1. It checks whether the request could **in principle be granted immediately or not**, depending on current resource availability. Since **r** represents the resources currently available in the system, the test can be written as $\mathbf{q_j} \leq \mathbf{r}$.
   a. **If the test is satisfied, there are enough resources in the system to grant the request and the Banker's algorithm proceeds to the next step.**
   b. **Otherwise, the request cannot be granted immediately, due to the lack of resources, and the requesting process has to wait**

2. The Banker's algorithm **simulates the allocation and generates a new state that reflects the effect of granting the request** on resource allocation (c_j), future needs (n_j), and availability (r), as follows:

$$\begin{cases} \mathbf{c_j}' &= \mathbf{c_j} + \mathbf{q_j} \\ \mathbf{n_j}' &= \mathbf{n_j} - \mathbf{q_j} \\ \mathbf{r}' &= \mathbf{r} - \mathbf{q_j} \end{cases}$$

3. Then, **the new state is analyzed to determine whether it is safe or not for what concerns deadlock. If the new state is safe, then the request is granted and the simulated state becomes the new, actual state of the system. Otherwise, the simulated state is discarded, the request is not granted immediately even if enough resources are available, and the requesting process has to wait**.

To assess the safety of a resource allocation state, the banker uses a **conservative approach**. It tries to compute at least one sequence of processes, called a **safe sequence**, **comprising all the n processes in the system** and that, when followed, allows each process in turn to attain the **worst-case** resource need it declared, and thus successfully conclude its work.

The safety assessment algorithm uses two auxiliary data structures:

- A (column) vector $\mathbf{w}$ that is initially set to the currently available resources (i.e., $\mathbf{w} = \mathbf{r}'$ initially) and tracks the evolution of the available resources as the safe sequence is being constructed.
- A (row) vector $\mathbf{f}$ of n elements. The j-th element of the vector, $\mathbf{f_j}$, corresponds to the j-th process: $f_j = 0$ if the j-th process has not yet been inserted into the safe sequence, $f_j = 1$ otherwise. The initial value of f is zero, because the safe sequence is initially empty.

The algorithm tries to find a new, suitable candidate to be appended to the safe sequence being constructed. In order to be a suitable candidate, a certain process P_j must not already be part of the sequence, and it must be able to reach its worst-case resource need, given the current resource availability state, i.e. $\mathbf{f_j} = \mathbf{0}$ (P_j is not in the safe sequence yet) $\wedge\, \mathbf{n_j} \leq \mathbf{w}$ (there are enough resources to satisfy n_j).

- If no suitable candidates can be found, the algorithm ends.
- After discovering a candidate, it must be appended to the safe sequence: $\mathbf{f_j} \coloneqq \mathbf{1}$ (P_j belongs to the safe sequence now) and $\mathbf{w} \coloneqq \mathbf{w} + \mathbf{c_j}$ (it releases its resources upon termination).

Then, the algorithm goes back to step 1, to extend the sequence with additional processes as much as possible. If $f_j = 1\, \forall j$, then **all processes belong to the safe sequence and the simulated state is certainly safe for what concerns deadlock**.

5.7 Deadlock Detection

The deadlock prevention approach poses significant restrictions on system design, whereas the Banker's algorithm presented requires information that could not be readily available and has a **significant run-time overhead**.

To address these issues, a **third family of methods acts even later than deadlock avoidance algorithms**. That is, these methods **allow the system to enter a deadlock condition but are able to detect this fact and react accordingly with an appropriate recovery action**. For this reason, they are collectively known as **deadlock detection and recovery algorithms**.

If there is **only one resource for each resource class in the system**, a straightforward way to detect a deadlock condition is to **maintain a resource allocation graph**, updating it whenever a resource is requested, allocated, and eventually released. Since this maintenance only involves adding and removing arcs from the graph, it is not computationally expensive and, with a good supporting data structure, can be performed in constant time.
The **resource allocation graph is examined at regular intervals**, looking for cycles. **The presence of a cycle is a necessary and sufficient indication that there is a deadlock in the system**.

If there are multiple resource instances belonging to the same resource class, this method cannot be applied. On its place we can, for instance, another algorithm, due to Shoshani and Coffman,similar to the Banker's algorithm. It can be executed at regular times and does not require knowing in advance the maximum number of resources that can be allocated by every process.

5.8 Starvation

Deadlock is only one aspect of a more **general group of phenomenons, known as indefinite wait, indefinite postponement, or starvation**.

In the Banker's algorithm, when more than one request can be granted safely but not all of them, a crucial point is how to pick the "right" request, so that no process is forced to wait indefinitely in favor of others.

More in general, **when several tasks cyclically compete for the same resource, it may happen that one 'unlucky' task, for any reason, will never acquire the resource because every time it tries accessing, there is another competing task that wins**.
In other words, it is not possible to define **an upper bound in the wait time for that resource**.

Several algorithms take into account starvation and guarantee that any requesting task will eventually have its request satisfied: for example the Peterson Mutual Exclusion algorithm (3.6) ensures that two competing tasks will gain alternatively access to the critical section.

6 Real-Time Systems

6.1 Real-Time Systems overview

In any concurrent program, the exact order in which processes execute is not completely specified. The interprocess communication and synchronization primitives are used to enforce **a set of constraints that limit the possible interleavings in order to ensure that the result of a concurrent program is correct in all cases**.

Nevertheless, the program will still exhibit a significant amount of **nondeterminism** because its processes may interleave in different ways without violating any of those constraints. The concurrent program output will of course be the same in all cases, but **its timings may still vary considerably from one execution to another**.

If one of the processes in a concurrent program has a **tight deadline on its completion time, only some of the interleavings that are acceptable from the concurrent programming perspective will also be adequate from the real-time execution point of view**.

As a consequence, **a real-time system must further restrict the nondeterminism found in a concurrent system because some interleavings that are acceptable with respect to the results of the computation may be unacceptable for what concerns timings**.

6.2 Real-Time Operating Systems

Most **general-purpose scheduling algorithms** emphasizes aspects such as:

- **Fairness**: in a general-purpose system, it is important to grant to each process a fair share of the available processor time and not to systematically put any process at a disadvantage with respect to the others.
- **Efficiency**: the scheduling algorithm is invoked very often in an operating system, and applications perceive this as an overhead. For this reason, the complexity of most general-purpose scheduling algorithms is forced to be $\mathcal{O}(1)$, i.e. it must not depend on how many processes there are in the system.
- **Throughput**: especially for batch systems, this is another important parameter to optimize because it represents the average number of jobs completed.

The main goal of a scheduling model is to ensure that a concurrent program does not only produce the expected output in all cases but is also **correct with respect to timings**. In order to do this, a scheduling model must comprise two main elements:

1. A **scheduling algorithm**, consisting of **a set of rules for ordering the use of system resources, in particular the processors**.
2. An analytical means of analyzing the system and predicting its **worst-case behaviour with respect to timing when that scheduling algorithm is applied**.

In a hard real-time scenario, the worst-case behavior is compared against the timing constraints the system must fulfill, to check whether it is acceptable or not.

6.3 The Basic Process Model

It turns out that the **analysis of an arbitrarily complex concurrent program, in order to predict its worst case timing behavior, is very difficult**. It is therefore necessary to introduce a **simplified process model that imposes some restrictions on the structure of real-time concurrent programs to be considered for analysis**.

The simplest model, also known as the **basic process model**, has the following characteristics:

1. The concurrent program consists of a **fixed number of processes, and that number is known in advance**.
2. **Processes are periodic, with known periods**. Moreover, **process periods do not change with time**. For this reason, processes can be seen as an infinite sequence of instances. **Process instances becomes ready for execution at regular time intervals at the beginning of each period**.
3. **Processes are completely independent of each other**
4. **Timing constrains are expressed by means of deadlines**. For a given process, **a deadline represents the upper bound on the completion time of a process instance**. All processes have **hard deadlines**, that is, **they must obey their temporal constraints all the time, and the deadline of each process is equal to its period**.
5. All processes have a **fixed worst-case execution time that can be computed offline**.
6. **All system's overheads**, for example, context switch times, **are negligible**.

The basic process model has some limitations, such as:

- Process independence must be understood in a very broad sense. It means that **there are no synchronization constraints among processes at all, so no process must ever wait for another → this rules out**, for instance, **mutual exclusion and synchronization semaphores**.
- **The deadline of a process is not always related to its period, and is often shorter that it**.
- Some processes are **sporadic rather than periodic**. In other words, **they are executed "on demand" when an external event, for example an alarm, occurs**.
- For some applications and hardware architectures, **scheduling and context switch times may not be negligible**.
- **The behavior of some nondeterministic hardware components, for example, caches, must sometimes be taken into account**, and this makes it **difficult to determine a reasonably tight upper bound on the process execution time**.

In Tab. 1 the notation used for real-time scheduling algorithms and analysis methods is summarized, while in Fig. 35 a clarifying example is reported.

Symbol	Meaning
τ_i	The i-th task
τ_{ij}	The j-th instance of the i-th task
T_i	The period of task τ_i
D_i	The relative deadline of task τ_i
C_i	The worst-case execution time of task τ_i
R_i	The worst-case response time of task τ_i
r_{ij}	The release time of τ_{ij}
f_{ij}	The response time of τ_{ij}
d_{ij}	The absolute deadline of τ_{ij}

Tab. 1: Notation for real-time scheduling algorithms and analysis methods

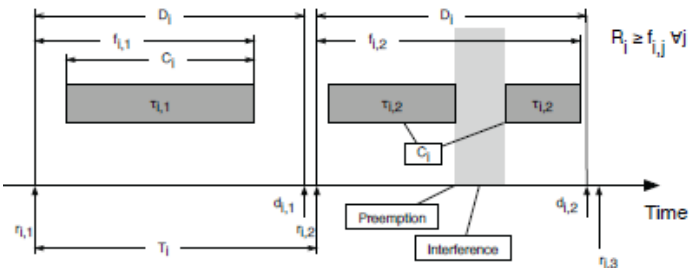

Fig. 35: A notation clarifying example

The example shows the execution of two instances, $\tau_{i,1}$ and $\tau_{i,2}$ of task τ_i. It is important to distinguish among the worst-case execution time of task τ_i, denoted by C_i, the response time of its j-th instance f_{ij}, and its worst-case response time, denoted by R_i. The worst-case execution time is the time required to complete the task without any interference from other activities, that is, if the task being considered were alone in the system.

The response time may (and usually will) be longer due to the effect of other tasks. A higher-priority task becoming ready during the execution of τ_i will lead to a preemption for most scheduling algorithms, so the execution of τ_i will be postponed and its completion delayed. Moreover, the execution of any tasks does not necessarily start as soon as they are released, that is, as soon as they become ready for execution.

It is also important to clearly distinguish between relative and absolute deadlines. The relative deadline D_i is defined for task τ_i as a whole and is the same for all instances. It indicates, for each instance, the distance between its release time and the deadline expiration. On the other hand, there is one distinct absolute deadline d_{ij} for each task instance τ_{ij}. Each of them denotes the instant in which the deadline expires for that particular instance.

6.4 The Cyclic Executive

The basic idea is to lay out **offline a completely static schedule such that its repeated execution causes all tasks to run at their correct rate and finish within their deadline**.
For what concerns its implementation, the schedule can essentially be thought of as a table of procedure calls, where each call represents (part of) the code of a task.

During execution, a very simple software component, the **cyclic executive, loops through the table and invokes the procedures it contains in sequence**.
To keep the executive in sync with the real elapsed time, the table also contains **synchronization points in which the cyclic executive aligns the execution with a time reference usually generated by a hardware component**.

The complete table is also known as the major cycle and is typically split into a number of slices called minor cycles, of equal and fixed duration.
Minor cycle boundaries are also synchronization points: during execution, the cyclic executive switches from one minor cycle to the next after waiting for a periodic clock interrupt. As a consequence, the activation of the tasks at the beginning of each minor cycle is synchronized with the real elapsed time, whereas all the tasks belonging to the same minor cycle are simply activated.

In Tab. 2 an example of a simple task set to be executed by a cyclic executive is reported, while in Fig. 36 an example of how a cyclic executive (with the relative code shown in Fig. 37) can successfully schedule it.

Task τ_i	**Period** T_i (ms)	**Execution time** C_i (ms)
τ_1	20	9
τ_2	40	8
τ_3	40	8
τ_4	80	2

Tab. 2: A simple task set to be executed by a cyclic executive

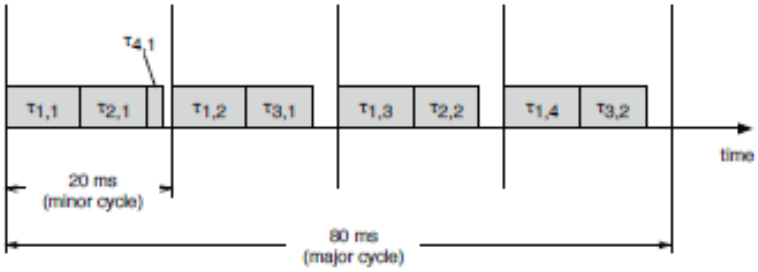

Fig. 36: An example of how a cyclic executive can successfully schedule the task set of Tab. 2

```
int main(int argc, void *argv[])
{
  ...
  timer_setup(20);

  while(1)
  {
    wait_for_interrupt();
      task_1();
      task_2();
      task_4();

    wait_for_interrupt();
      task_1();
      task_3();

    wait_for_interrupt();
      task_1();
      task_2();

    wait_for_interrupt();
      task_1();
      task_3();
  }
}
```

Fig. 37: Sample implementation of the cyclic executive

All task periods must be an integer multiple of the minor cycle period. Otherwise, it would be impossible to execute them at their proper rate.
On the other hand, it is also desirable to **keep the minor cycle length as large as possible**. This is useful not only to **reduce synchronization overheads** but also to **make it easier to accommodate tasks with a large execution time**.

To satisfy both constraints **the minor cycle length is set to be equal to the Greatest Common Divisor (GCD) of the periods of the tasks to be scheduled**.
The cyclic executive repeats the same schedule over and over at each major cycle.
Therefore, **the major cycle must be big enough to be an integer multiple of all task periods, but no larger than that to avoid making the scheduling table larger than necessary for no reason**.
A sensible choice is to let the **major cycle length be the Least Common Multiple (LCM) of the task periods**. Sometimes this is also called the hyperperiod of the task set.

There are cases in which it may be necessary to split tasks with large execution times into pieces (with an increase in complexity regarding the management of shared resources): this is the case for the task set in Tab. 3 as shown in Fig. 38

Task τ_i	**Period** T_i (ms)	**Execution time** C_i (ms)
τ_1	25	10
τ_2	50	8
τ_3	100	20

Tab. 3: Large execution times, of τ_3 in this case, may lead to problems when designing a cyclic executive

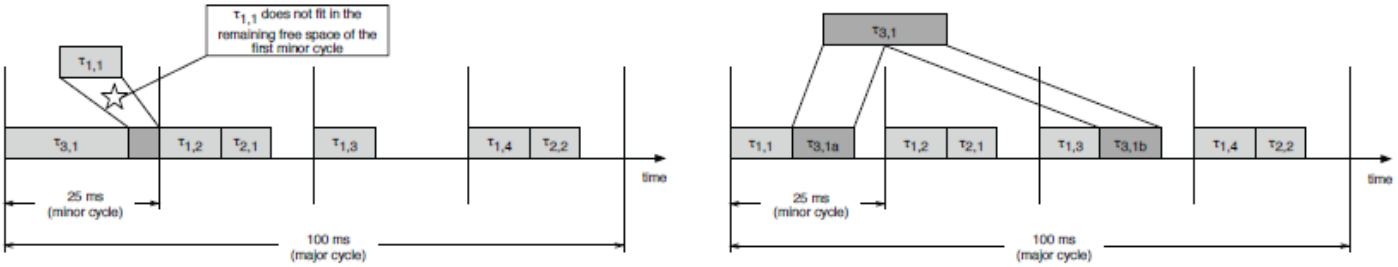

Fig. 38: In some cases, such as for the task set of Tab. 3, it is necessary to split one or more tasks with a large execution time into pieces to fit them into a cyclic executive.

7 Task Based Scheduling

7.1 Fixed and variable priorities in Real-Time scheduling

Variable (dynamic) priority assigned to tasks is normally performed in non real-time systems in order to make the system more "fluid": based on the concept that a task that is going to use the processor for the shortest time should take it first → this maximizes the number of tasks executed per unit of time.

Fixed priority assignment favors "important" tasks that must be executed first.
In Linux and Windows tasks above a given priority are not subject to dynamic priority management.

Dynamic priorities are also used in real-time systems: while the implementation is more complex, the processor utilization is better exploited.

7.2 Critical instant

The **critical instant** is the **worst situation that may occur when a set of periodic tasks with given periods and computation times is scheduled**.
Task jobs can in fact be **released at arbitrary instants within their period**, and **the time between period occurrence and job release is called the phase of the task**.

We shall see that **the worst situation will occur when all the jobs are initially released at the same time** (i.e., when the phase of all the tasks is zero).
Proving that the system under consideration is schedulable in such a bad situation means proving that it will be schedulable for every task phase.

The relative deadline of a task is equal to its period, that is $D_i = T_i \, \forall i$.
Hence, for each task instance, **the absolute deadline is the time of its next release**, that is $d_{i,j} = r_{i,j} + 1$.
We shall say that there is an **overflow** at time t if t is the deadline for a job that misses the deadline.

A scheduling algorithm is feasible for a given set of tasks if they are scheduled so that no overflows ever occurs.
A critical instant for a task is an instant at which the release of the task will produce the largest response time.
A critical time zone for a task is the interval between a critical instant and the end of the task response.

Theorem. *A critical instant for any task occurs whenever it is released simultaneously with the release of all higher-priority tasks.*

Proof. Using Fig. 39 as reference, if τ_m is released at t_1, between t_1 and $t_1 + T_m$, the time of the next release of τ_m, other task with a higher priority will possibly be releases and interfere with the execution of τ_m because of preemption.

It can easily be observed that if we anticipate the release of τ_m the interference due to higher priority tasks will never lower.
As a consequence, the critical instant corresponds to phase zero, i.e. the task is released at the same time in respect of higher priority tasks. ■

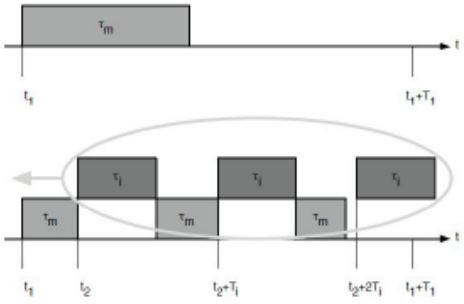

Fig. 39: Interference to τ_m due to higher-priority tasks τ_i

7.3 Rate Monotonic (RM) priority assignment

Rate monotonic is a policy for fixed-priority assignment in periodic tasks, which assigns a priority that is inversely proportional to the task period: the shorter the task period, the higher its priority.

Consider, for example, the three tasks in Fig. 40: task τ_1, which has the smallest period, will have the highest priority, followed by task τ_3 and τ_2. Task τ_2 is preempted by task τ_1 and τ_3, which have higher priority, and its execution is split into two and three parts.
If the simulation is successful for the LCM of the periods, then it is also successful for any other period.

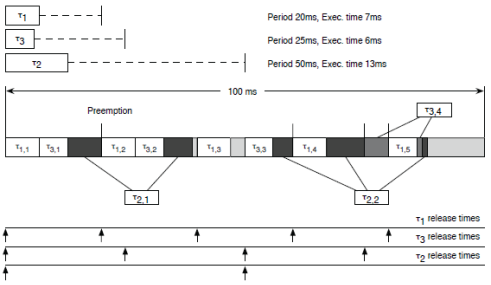

Fig. 40: Scheduling sequence for tasks τ_1, τ_2, τ_3 under RM

Given the following assumptions:

- Every task τ_i is periodic with period T_i
- The relative deadline D_i for every tasks τ_i is equal to its period T_i
- Tasks are scheduled preemptively and according to their priority
- There is only one processor

The following theorem holds true:

Theorem. *RM is the optimal priority assignment policy, i.e. every system that is schedulable using fixed priorities is schedulable using RM policy.*

In other words, if a system is not schedulable under RM policy, then it will remain not schedulable under any other fixed priority assignment.

Proof. Let's consider two tasks, τ_1 and τ_2, with $T_1 < T_2$. If their priorities are not assigned according to RM, then τ_2 will have a priority higher then τ_1. At a critical instant, their situation is that shown in Fig. 41.

The schedule is feasible if (and only if) the following inequality is satisfied: $C_1 + C_2 \leq T_1$. In fact, if the sum of the computation time of τ_1 and τ_2 is greater than the period of τ_1, it is not possible that τ_1 can finish its computation within its deadline.

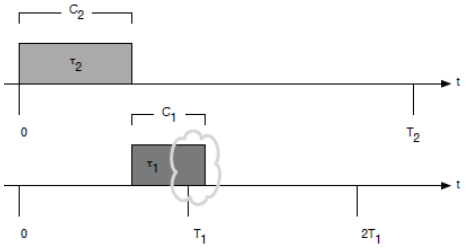

Fig. 41: Tasks τ_1 and τ_2 not scheduled under RM

If priorities are assigned according to RM, then task τ_1 will have a priority higher than τ_2. If we let F be the number of periods of τ_1 **entirely** contained within T_2, that is, $F = \lfloor \frac{T_2}{T_1} \rfloor$, then in order to determine the feasibility conditions, we must consider two cases (which cover all possible situations):

1. The execution time C_1 is "short enough" so that all the instances of τ_1 within the critical zone of τ_2 are completed before the next release of τ_2.
2. The execution of the last instance of τ_1 that starts within the critical zone of τ_2 overlaps the next release of τ_2.

Let's consider case 1 first, corresponding to Fig. 42. It occurs when:

$$C_1 < T_2 - FT_1$$

The task set is schedulable if and only if:

$$(F+1)C_1 + C_2 \leq T_2$$

Now consider case 2, corresponding to Fig. 43. It occurs when:

$$C_1 \geq T_2 - FT_1$$

The task set is schedulable if and only if:

$$FC_1 + C_2 \leq FT_1$$

In summary, given a set of two tasks, τ_1 and τ_2, with $T_1 < T_2$ we have the following two conditions:

1. when priorities are assigned according to RM, the set is schedulable if and only if:
   - $(F+1)C_1 + C_2 \leq T_2$, when $C_1 < T_2 - FT_1$
   - $FC_1 + C_2 \leq FT_1$, when $C_1 \geq T_2 - FT_1$
2. when priorities are not assigned according to RM, the set is schedulable if and only if $C_1 + C_2 \leq T_1$

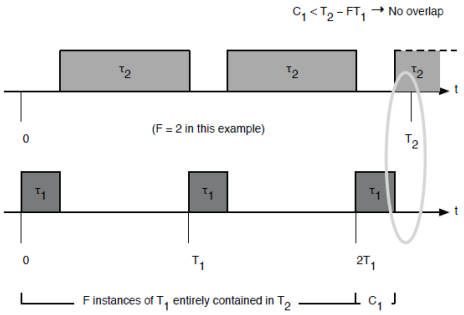

Fig. 42: Situation in which all the instances of τ_1 are completed before the next release of τ_2

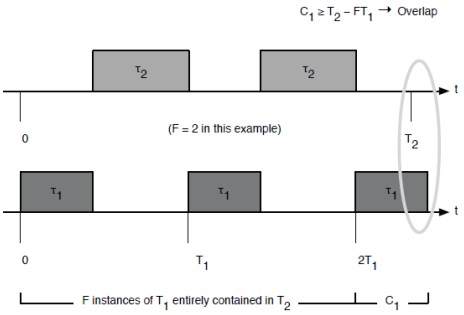

Fig. 43: Situation in which the last instance of τ_1 that starts within the critical zone of τ_2 overlaps the next release of τ_2

To prove the optimality of RM with two tasks, we must show that the following two implications hold:

1. If $C_1 < T_2 - FT_1$, then $C_1 + C_2 \leq T_1 \implies (F+1)C_1 + C_2 \leq T_2$
2. If $C_1 \geq T_2 - FT_1$, then $C_1 + C_2 \leq T_1 \implies FC_1 + C_2 \leq FT_1$

Considering the first implication: if we multiply both member of $C_1 + C_2 \leq T_1$ by F and then add C_1, we obtain:

$$(F+1)C_1 + FC_2 \leq FT_1 + C_1$$

We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), and hence,

$$FC_2 \geq C_2$$

Moreover, from the hypothesis we have:

$$FT_1 + C_1 < T_2$$

As a consequence, we have:

$$(F+1)C_1 + C_2 \leq (F+1)C_1 + FC_2 \leq FT_1 + C_1 \leq T_2$$

which proves the first implication.
Consider now the second implication: if we multiply both member of $C_1 + C_2 \leq T_1$ by F, we obtain:

$$FC_1 + FC_2 \leq FT_1$$

We know that $F \geq 1$ (otherwise, it would not be $T_1 < T_2$), and hence,

$$FC_2 \geq C_2$$

As a consequence, we have:

$$FC_1 + C_2 \leq FC_1 + FC_2 \leq FT_1$$

which concludes the proof of the optimality of RM when considering two tasks. ■

The optimality of RM is then extended to an arbitrary set of tasks thanks to the following theorem:

Theorem. *If the task set $\tau_1, \ldots, \tau_n$ (n tasks) is schedulable by any arbitrary, but fixed, priority assignment, then it is schedulable by RM as well.*

Proof. As a direct consequence of the previous considerations: let τ_i and τ_j be two tasks of adjacent priorities, τ_i being the higher-priority one, and suppose that $T_i > T_j$. Having adjacent priorities, both τ_i and τ_j are affected in the same way by the interferences coming from the higher-priority tasks (and not at all by the lower-priority ones).

Hence, we can apply the result just obtained and state that if we interchange the priorities of τ_i and τ_j, the set is still schedulable.
Finally, we notice that the RM priority assignment can be obtained from any other priority assignment by a sequence of pairwise priority reorderings as above, thus ending the proof. ■

7.4 Earliest Deadline First (EDF)

The Earliest Deadline First (abbreviated as EDF) algorithm selects tasks according to their absolute deadlines. That is, **at each instant, tasks with earlier deadlines will receive higher priorities**. Recall that the absolute deadline d_{ij} of the j-th instance (job) of the task τ_i is formally $d_{i,j} = \phi_i + jT_i + D_i$ where ϕ_i is the phase of task τ_i, that is, the release time of its instance (for which $j = 0$), and T_i and D_i are the period and relative deadlines of task τ_i, respectively.

The priority of each task is assigned **dynamically**, because **it depends on the current deadlines of the active task instances**.
Task priorities may be updated only when a new task instance is released (task instances are released at every task period). Afterwards, when time passes, **the relative order due to the proximity in time of the next deadline remains unchanged among active tasks, and therefore, priorities are not changed**.

As for RM, EDF is an intuitive choice as it makes sense to increase the priority of more "urgent" tasks, that is, for which deadline is approaching.

It can be proved that **EDF is the optimal scheduling algorithm**, that is, **if any task set is schedulable by any (fixed, or dynamic) scheduling algorithm, then it is also schedulable by EDF**.
This holds under the following assumptions:

- Task are scheduled preemptively
- There is only one processor

In practice EDF is more difficult to implement and in this case **the scheduler requires additional information about task deadlines that is normally not known** (if dealing with fixed priority, use RM instead).

8 Schedulabilty Analysis based on Utilization, Response Time Analysis

8.1 Schedulability analysis

We have analyzed two priority assignment policies: fixed priority and variable priority.

- For fixed-priority scheduling, it has been shown that Rate Monotonic (RM) Scheduling is optimal
- For variable-priority assignment, the optimality of Earliest Deadline First (EDF)

Despite the elegance and importance of these two results, their practical impact for the moment is rather limited. In fact, what we are interested in practice is to **know whether a given task assignment is schedulable**, before knowing what scheduling algorithm to use.

A **sufficient condition for schedulability** will be presented, which, **when satisfied, ensures that the given set of tasks is definitely schedulable**. The schedulability check will be **very simple**, being **based on an upper limit in the processor utilization**.

This simplicity is, however, paid for by the fact that this condition is only a **sufficient** one: as a consequence, **if the utilization check fails, we cannot state that the given set of tasks is not schedulable**.

8.2 Processor Utilization

In the following,it is assumed that the **basic process model** is being used and, in particular, we shall consider **single-processor systems**.

Given a set of N periodic task $\Gamma = \{\tau_1, \ldots, \tau_N\}$, the **processor utilization factor U is the fraction of processor time spent in the execution of the task set**, that is

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \quad \text{where } C_i/T_i \text{ is the fraction of processor time spent executing task } \tau_i$$

The processor utilization factor is therefore **a measure of the computational load imposed on the processor by a given task set** and can be increased by increasing the execution times C_i of the tasks.

A task set Γ is said to **fully utilize the processor with a given scheduling algorithm A if it is schedulable by A, but any increase in the computational load** C_i **of any of its task will make it no longer schedulable**. The corresponding **upper bound of the utilization factor** is denoted as $\mathbf{U}_{ub}(\mathbf{\Gamma}, \mathbf{A})$.

If we consider now all the possible task sets Γ, it is interesting to ask **how large the utilization factor can be in order to guarantee the schedulability of any task set** Γ **by a given scheduling algorithm A**. In order to do this, we must determine **the minimum value of** $U_{ub}(\Gamma, A)$ **over all task sets** Γ **that fully utilize the processor with the scheduling algorithm A**.
This new value, called **least upper bound** and denoted as $\mathbf{U}_{lub}(\mathbf{A})$ will **only depend on the scheduling algorithm A** and it is defined as

$$U_{lub}(A) = \min_{\Gamma} U_{ub}(\Gamma, A) \quad \text{where } \Gamma \text{ represents the set of all task sets that fully utilize the processor}$$

A pictorial representation of the definitions of $U_{ub}(\Gamma, A)$ and $U_{lub}(A)$ is given in Fig. 44.

For every possible task set Γ_i, the maximum utilization depends on both A and Γ_i.
The actual utilization for task set Γ_i, will depend on the computational load of the tasks but will never exceed $U_{ub}(\Gamma_i, A)$.

Since $U_{lub}(A)$ is the minimum upper bound over all possible task sets, **any task set whose utilization factor is below** $U_{lub}(A)$ **will be schedulable by A**.

Observe that for a given task set Γ with scheduling algorithm A its **utilization may exceed** $U_{lub}(A)$ **and be schedulable nevertheless, but this does not hold in general**.
On the other side, **if the utilization factor** U for a given task set Γ with scheduling algorithm A **does not exceed** $U_{lub}(A)$ **we can for sure state that is schedulable, i.e. no task will ever miss its deadline**.
This represents therefore a **sufficient condition for schedulability**.

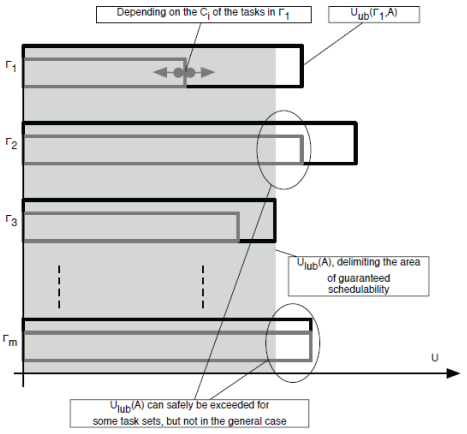

Fig. 44: Upper Bounds and Least Upper Bound for scheduling algorithm A

Regardless of the adopted scheduling algorithm, **there is an upper limit in processor utilization that can never be exceeded**, as stated in the following theorem:

Theorem. *If the processor utilization factor* U *of a task set* Γ *is greater than one (that is, if* $U > 1$*), then the task set is not schedulable, regardless of the scheduling algorithm.*

This result is intuitive, a set of tasks cannot require more than 100% cpu time in order to be executed, regardless of the chosen scheduling algorithm: this represents therefore a **necessary condition for schedulability** (Fig. 45).

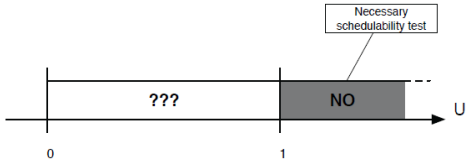

Fig. 45: Necessary schedulability condition

8.3 A sufficient condition fo Rate Monotonic

For RM it is possible to compute U_{lub} as stated by the following theorem (of which we will omit the proof):

Theorem. *For a set of N periodic tasks scheduled by the Rate Monotonic algorithm, the least upper bound of the processor utilization factor U_{lub} is*

$$U_{lub} = N(2^{1/N} - 1)$$

Recalling that a sufficient schedulability condition for a given set of tasks with processor utilization U and scheduling algorithm A is that U is not greater than $U_{lub}(A)$, this result provides a **sufficient schedulability condition for RM**.

Since U_{lub} is monotonically decreasing with respect to N, for large values of N, it asymptotically approaches the value $\ln 2 \approx 0.693$.
From this observation **a simpler, but more pessimistic, sufficient test can be stated: regardless of N, any task set with a combined utilization factor of less than ln 2 will always be schedulable by the Rate Monotonic algorithm**.

In Fig. 46 the schedulability conditions for Rate Monotonic are shown.

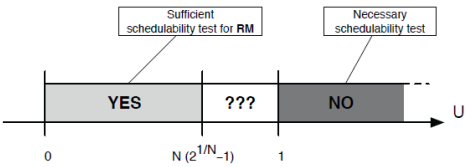

Fig. 46: Schedulability conditions for Rate Monotonic

Let's analyze some examples:

- The processor utilization for the task set in Tab. 4 is $0.4 + 0.1 + 0.125 = 0.625 < \ln 2 \approx 0.693$. We can therefore definitely state that this set of tasks is schedulable by RM scheduling.
- The processor utilization for the task set in Tab. 5 is $0.2+0.2+0.5 = 0.9 > \ln 2 \approx 0.693$. The schedulability test is not conclusive: in this specific case, the task set is not schedulable by RM scheduling since task τ_1 misses its deadline when all tasks are released at their critical instant, as shown in Fig. 47.
- The processor utilization for the task set in Tab. 6 is $0.5 + 0.250 + 0.250 = 1 > \ln 2 \approx 0.693$. The schedulability test is not conclusive: in this specific case, the task set is schedulable when all tasks are released at their critical instant, as shown in Fig. 48.

Task τ_i	Period T_i	Computation Time C_i	Priority	Utilization
τ_1	50	20	Low	0.400
τ_2	40	4	Medium	0.100
τ_3	16	2	High	0.125

Tab. 4: A task set definitely schedulable by RM.

Task τ_i	Period T_i	Computation Time C_i	Priority	Utilization
τ_1	50	10	Low	0.200
τ_2	30	6	Medium	0.200
τ_3	20	10	High	0.500

Tab. 5: A task set for which the sufficient RM scheduling condition does not hold.

Task τ_i	Period T_i	Computation Time C_i	Priority	Utilization
τ_1	80	40	Low	0.500
τ_2	40	10	Medium	0.250
τ_3	20	5	High	0.250

Tab. 6: A task set for which the sufficient RM scheduling condition does not hold.

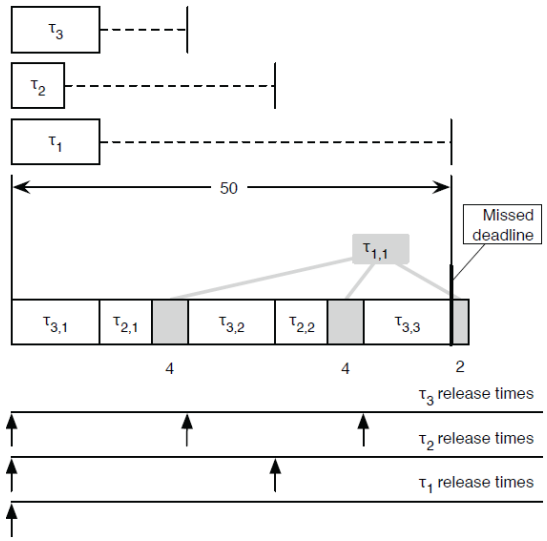

Fig. 47: RM scheduling for a set of tasks with $U = 0.900$

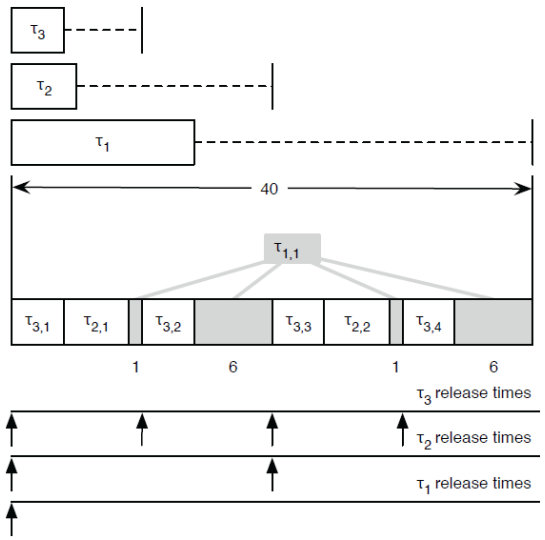

Fig. 48: RM scheduling for a set of tasks with $U = 1$

8.4 Schedulability condition for EDF

Theorem. *A set of N periodic tasks is schedulable with the Earliest Deadline First algorithm if and only if its processor utilization is not greater than 1, i.e.*

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

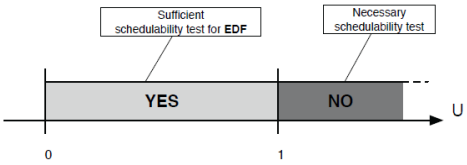

Fig. 49: Schedulability conditions for EDF

8.5 Response Time Analysis

Response Time Analysis (RTA) is an exact (necessary and sufficient) schedulability test for any fixed-priority assignment scheme on single-processor systems. It allows prediction of the worst-case response time of each task, which depends on the interference due to the execution of higher-priority tasks. The worst-case response times are then compared with the corresponding task deadlines to assess whether all tasks meet their deadline or not.

In this analysis the condition $D_i = T_i$ assumed before is now relaxed into condition $D_i \leq T_i$.
During execution, the preemption mechanism grabs the processor from a task whenever a higher-priority task is released. For this reason, **all tasks (except the highest-priority one) suffer a certain amount of interference from higher-priority tasks during their execution**.

Therefore, **the worst-case response time R_i of task τ_i is computed as the sum of its computation time C_i and the worst-case interference I_i it experiences, that is $R_i = C_i + I_I$.**
Observe that the interference must be considered over any possible interval $[t, t + R_i]$, that is, for any t, to determine the worst case.
We already know, however, that **the worst case occurs when all the higher-priority tasks are released at the same time as task τ_i.** In this case, t becomes a **critical instant** and, without loss of generality, it can be assumed that all tasks are released simultaneously at the critical instant $t = 0$.

The contribution of each higher-priority task to the overall worst-case interference will now be analyzed individually by considering the interference due to any single task τ_j of higher priority than τ_i.
Within the interval $[0, R_i]$, τ_j will be released one (at $t = 0$) or more times. The **exact number of releases** can be computed by means of a ceiling function, as $\left\lceil \frac{R_i}{T_j} \right\rceil$.

Since each release of τ_j will impose on τ_i an interference of C_j, **the worst case interference imposed on τ_i by τ_j is** $\left\lceil \frac{R_i}{T_j} \right\rceil C_j$.
This is because if task τ_j is released at any time $t < R_i$, then its execution must have finished before R_i, as τ_j has a larger priority, and therefore, that instance of τ_j must have terminated before τ_i can resume.

Let $hp(i)$ denote the **set of task indexes with a priority higher than τ_i**. These are the tasks from which τ_i will suffer interference. Hence, the **total interference endured by τ_i** is

$$I_i = \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Recalling that $R_i = C_i + I_i$, we get the following **recursive relation for the worst case response time R_i of task τ_i**:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

No simple solution exists for this equation since R_i appears on both sides.
The equation may have more than one solution: the smallest solution is the actual worst-case response time. The simplest way of solving the equation is to form a **recurrence relationship** of the form

$$w_i^{(k+1)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^{(k)}}{T_j} \right\rceil C_j$$

where $w_i^{(k)}$ is the k-th estimate of R_i and $w_i^{(k+1)}$ the (k+1)-th estimate from the k-th in the above relationship.
The initial approximation $w_i^{(0)}$ is chosen by letting $w_i^{(0)} = C_i$ (the smallest possible value of R_i).
The succession $w_i^{(0)}, w_i^{(1)}, \ldots, w_i^{(k)}, \ldots$ is monotonically non-decreasing.

Two cases are possible for the succession $w_i^{(0)}, w_i^{(1)}, \ldots, w_i^{(k)}, \ldots$:

- **If the equation has no solution, the succession does not converge, and it will be $w_i^{(k)} > D_i$ for some k.** In this case, τ_i **clearly does not meet its deadline**.
- Otherwise, **the succession converges to R_i, and it will be $w_i^{(k)} = w_i^{(k-1)} = R_i$ for some k.** In this case, τ_i **meets its deadline if and only if $R_i \leq D_i$.**

As an example, let's consider the task set in Tab. 7, with $D_i = T_i$. The priority assignment is Rate Monotonic and the CPU utilization factor U is

$$U = \sum_{i=1}^{3} \frac{C_i}{T_i} = \frac{3}{8} + \frac{4}{14} + \frac{5}{22} \simeq 0.89$$

Task τ_i	**Period** T_i	**Computation Time** C_i	**Priority**
τ_1	8	3	High
τ_2	14	4	Medium
τ_3	22	5	Low

Tab. 7: A task set for which the sufficient RM scheduling condition does not hold.

The necessary schedulability test ($U \leq 1$) does not deny schedulability, but the sufficient test for RM is of no help in this case because $U > 1/3(2^{1/3} - 1) \simeq 0.78$.

The highest-priority task τ_1 does not endure interference from any other task. Hence, it will have a response time equal to its computation time, that is, $R_1 = C_1$. In fact, $hp(1) = \emptyset$ and, given $w_1^{(0)} = C_1$, we trivially have $w_1^{(1)} = C_1$. In this case, $C_1 = 3$, hence $R_1 = 3$ as well. Since $R_1 = 3$ and $D_1 = 8$, then $R_1 \leq D_1$ and τ_1 meets its deadline.

For τ_2, $hp(2) = \{1\}$ and $w_2^{(0)} = C_2 = 4$. The next approximations of R_2 are

$$w_2^{(1)} = 4 + \left\lceil \frac{4}{8} \right\rceil 3 = 7$$
$$w_2^{(2)} = 4 + \left\lceil \frac{7}{8} \right\rceil 3 = 7$$

Since $w_2^{(2)} = w_2^{(1)} = 7$, then the succession converges, and $R_2 = 7$. In other words, widening the timing window from 4 to 7 time units did not introduce any additional interference. Task τ_2 meets its deadline, too, because $R_2 = 7$, $D_2 = 14$, and thus $R_2 \leq D_2$.

For τ_3, $hp(3) = \{1, 2\}$. It gives rise to the following calculations:

$$w_3^{(0)} = 5$$
$$w_3^{(1)} = 5 + \left\lceil \frac{5}{8} \right\rceil 3 + \left\lceil \frac{5}{14} \right\rceil 4 = 12$$
$$w_3^{(2)} = 5 + \left\lceil \frac{12}{8} \right\rceil 3 + \left\lceil \frac{12}{14} \right\rceil 4 = 15$$
$$w_3^{(3)} = 5 + \left\lceil \frac{15}{8} \right\rceil 3 + \left\lceil \frac{15}{14} \right\rceil 4 = 19$$
$$w_3^{(4)} = 5 + \left\lceil \frac{19}{8} \right\rceil 3 + \left\lceil \frac{19}{14} \right\rceil 4 = 22$$
$$w_3^{(5)} = 5 + \left\lceil \frac{22}{8} \right\rceil 3 + \left\lceil \frac{22}{14} \right\rceil 4 = 22$$

$R_3 = 22$ and $D_3 = 22$, and thus $R_3 \leq D_3$ and τ_3 (just) meets its deadline.
In this case RTA guarantees that all tasks meet their deadline.

9 Sporadic tasks, task interaction and blocking

9.1 Aperiodic and sporadic tasks

Up to now we have considered only periodic tasks, where every task consists of an infinite sequence of identical activities that are regularly released, or activated, at a constant rate.
A **aperiodic task consists of an infinite sequence of identical jobs**. However, unlike periodic tasks, **their release does not take place at a regular rate**.
Typical examples of aperiodic tasks are:

- **User interaction**: Events generated by user interaction (key pressed, mouse clicked) and which require some sort of system response
- **Event reaction**: External events, such as **alarms**, may be generated at unpredictable times whenever some condition either in the system or in the controlled plant (as an example) occurs.

A aperiodic task for which its is possible to determine a **minimum inter-arrival time interval** is called a **sporadic task**.
Sporadic tasks can model many situations that occur in practice.

- For example, a minimum inter-arrival time can be safely assumed for events generated by user interaction, because of the reaction time of the human brain.
- More in general, **systems events can be filtered in advance to ensure that, after the occurrence of a given event, no new instance will be issued until after a given dead time**.

One simple way of expanding the basic process model to include sporadic tasks is to **interpret the period T_i as the minimum inter-arrival time interval**.
For example, a sporadic task τ_i with $T_i = 20\,\text{ms}$ **is guarantee not to arrive more frequently than once in any 20 ms interval**. Actually, it may arrive less frequently, but **a suitable schedulability analysis test will ensure (if passed) that the maximum rate can be sustained**.

For these tasks, **assuming $D_i = T_i$**, that is, **a relative deadline equal to the minimum inter-arrival time, is unreasonable because they usually encapsulate error handlers or respond to alarms**.
The fault model of the system may state that **the error routine will be invoked rarely, but, when it is, it has a very short deadline**.
For many periodic tasks it is useful to define a deadline shorter than the period.

The RTA method just described is adequate for use with the extended process model just introduced, that is, when $D_i \leq T_i$. Observe that **the method works with any fixed-priority ordering**, and not just with the RM assignment, **as long as the set $hp(i)$ of tasks with priority larger than task τ_i is defined appropriately for all i and we use a preemptive scheduler**.

9.2 Deadline Monotonic Priority Order (DMPO)

RM was shown to be an optimal fixed-priority assignment scheme when $D_i = T_i$, this is no longer true for $D_i \leq T_i$.
The following theorem introduces another **fixed-priority assignment no more based on the period of the task but on their relative deadlines**.

Theorem. *The deadline monotonic priority order (DMPO), in which each task has a fixed priority inversely proportional to its deadline, is optimum for a preemptive scheduler under the basic process model extended to let $D_i \leq T_i$.*

The optimality of DMPO means that, if any task set Γ can be scheduled using a preemptive, fixed-priority scheduling algorithm A, then the same task set can also be scheduled using the DMPO.
As before, such a priority assignment sounds to be good choice since it makes sense to give precedence to more "urgent" tasks.

The formal proof involves transforming the priorities of Γ (as assigned by A), until the priority ordering is Deadline Monotonic (DM), showing that each transformation step will preserve schedulability.

9.3 Task Interaction and Priority Inversion

The basic process model, assumed an underlying set of hypotheses to prove several interesting properties of real-time scheduling algorithms. Unfortunately, some aspects of the basic process model are not fully realistic, and make those results hard to apply to real-world problems.

The hypothesis that tasks are completely independent from each other regarding execution **is particularly troublesome because it sharply goes against the basics of all the interprocess communication methods**.
In one form or another, **they all require tasks to interact and coordinate, or synchronize, their execution**. In other words, **tasks will sometimes be forced to block and wait until some other task performs an action in the future**.

For example, tasks may either have to wait at a critical region's boundary to keep a shared data structure consistent, or wait for a message from another task before continuing.
In all cases, **their execution will clearly no longer be independent from what the other tasks are doing at the moment**.

The following discussion will mainly address **task interactions due to mutual exclusion**, a ubiquitous necessity when dealing with shared data.

Informally speaking, **when a high-priority task is waiting for a lower-priority task to complete some required computation, the task priority scheme is**, in some sense, **being hampered** because the high-priority task would take precedence over the lower-priority one in the model.
This happens even in very simple cases, for example, when several tasks access a shared resource by means of a critical region protected by a mutual exclusion semaphore. Once a lower-priority task enters its critical region, the semaphore mechanism will block any higher-priority task wanting to enter its own critical region protected by the same semaphore and force it to wait until the former exits.
We have already seen an example of what has just been described in 3.7.

This phenomenon is called **priority inversion** and, if not adequately addressed, **can have adverse effects on the schedulability of the system**, to the point of **making the response time of some tasks completely unpredictable because the priority inversion region may last for an unbounded amount of time**. In this case the system is said to suffer from **unbounded priority inversion**.

Knowing in advance the amount of time any task will spend in the critical section, at a first glance, **mutual exclusion can be taken into account adding to the computation time the maximum time spent by any task in the critical section**.
Indeed, in the worst case, if a task is accessing N critical sections, it has to wait for each critical section the maximum time T_N, i.e. the maximum time spent by any task in that critical section.

Let's now see an example of unbounded priority inversion: consider the execution of three real-time tasks τ_H (high priority), τ_M (middle priority), and τ_L (low priority), executed under the control of a fixed-priority scheduler on a single-processor system. We will also assume, as shown in Fig. 50, that τ_H and τ_L share some information, stored in a certain shared memory area M.

Both τ_H and τ_L make access to M only within a suitable critical region, protected by a mutual exclusion semaphore `m`. On the contrary, τ_M has nothing to do with τ_H and τ_L, that is, it does not interact with them in any way. The only relationship among τ_M and the other two tasks is that "by chance" (unknowingly to the programmer) it was assigned a priority that happens to be between the priority of τ_H and τ_L.
The following sequence of events may happen:

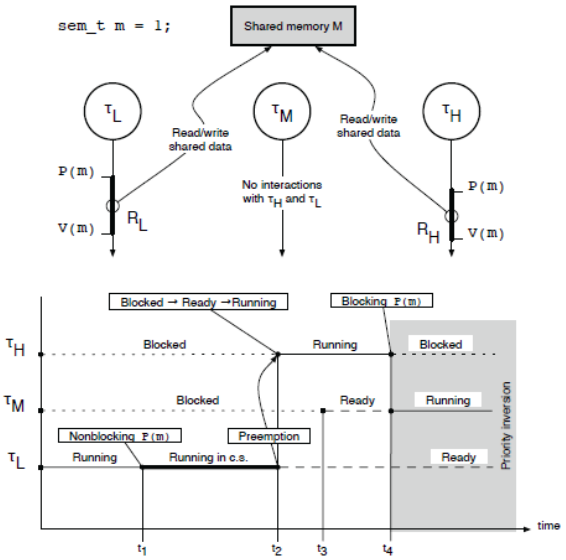

Fig. 50: A simple example of unbounded priority inversion involving three tasks

- Initially, neither τ_H nor τ_M are ready for execution. They may be, for instance, periodic tasks waiting for their next execution instance or they may be waiting for the completion of an I/O operation.
- Conversely, τ_L is ready; the fixed-priority scheduler moves it into the Running state and executes it.

- During its execution, at t_1 in the figure, τ_L enters into its critical region R_L, protected by semaphore `m`. The semaphore primitive `P(m)` at the critical region's boundary is nonblocking because no other tasks are accessing the shared memory M at the moment. Therefore, τ_L is allowed to proceed immediately and keeps running within the critical region.
- If τ_H becomes ready while τ_L still is in the critical region, the fixed-priority scheduler stops executing τ_L, puts it back into the Ready state, moves τ_H into the Running state and executes it. This preemption takes place at t_2.
- At t_3, task τ_M becomes ready for execution and moves into the Ready state, too, but this has no effect on the execution of τ_H, because the priority of τ_M is lower than the priority of τ_H.
- As τ_H proceeds with its execution, it may try to enter its critical region (at t_4). At this point, τ_H is blocked by the semaphore primitive `P(m)` because the value of semaphore `m` is now zero. This behavior is correct since τ_L is within its own critical region R_L, and the semaphore mechanism is just enforcing mutual exclusion between R_L and R_H, the critical region τ_H wants to enter.
- Since τ_H is no longer able to run, the scheduler chooses another task to execute. As τ_M and τ_L are both Ready but the priority of τ_M is greater than the priority of τ_L, the former is brought into the Running state and executed.

Therefore, starting from t_4, a priority inversion region begins: τ_H (the highest priority task in the system) is blocked by τ_L (a lower priority task) and the system executed τ_M (yet another lower priority task).

Although the existence of this priority inversion region cannot be questioned because it entirely depends on how the mutual exclusion mechanism for the shared memory area M has been designed to work, an interesting question is for how long it will last.
Contrary to expectations, the answer is that the duration of the priority inversion region does not depend at all on the two tasks directly involved in it, that is, τ_H and τ_L. It depends instead on how much time τ_M keeps running, over which the programmer has no control at all: the time τ_H has to wait for τ_L to exit its critical region is therefore unbounded.

9.4 The Priority Inheritance Protocol

On a single-processor system, a very simple and drastic solution to the unbounded priority inversion problem is to **forbid preemption completely during the execution of all critical regions**.
This may be obtained by **disabling the operating system scheduler or**, even more drastically, **turning interrupts off within critical regions**.

However, this technique introduces a new kind of blocking, of a different nature. That is, any higher-priority task τ_M that becomes ready while a low priority task τ_L is within a critical region will not get executed and we therefore consider it to be blocked by τ_L until τ_L exits from the critical region, even if τ_M does not interact with τ_L at all.

A better solution is to **dynamically increase the priority of a task as soon as it is blocking some higher-priority tasks**. In particular, **if a task τ_L is blocking a set of n higher-priority tasks $\tau_{H_1}, \ldots, \tau_{H_n}$ at a given instant, it will temporarily inherit the highest priority among them. This prevents any middle-priority task from preempting τ_L and unduly make the blocking experienced by $\tau_{H_1}, \ldots, \tau_{H_n}$ any longer than necessary**.

The **priority inheritance protocol** itself consists of the following set of rules:

1. **When a task τ_H attempts to enter a critical region that is "busy"**, that is, its controlling semaphore has already been taken by another task τ_L, **it blocks, but it also transmits its active priority to the task τ_L that is blocking it if the active priority of τ_L is lower than τ_H's**.
2. As a consequence, τ_L **will execute the rest of its critical region with a priority at least equal to the priority it just inherited. In general, a task inherits the highest active priority among all tasks it is blocking**.
3. **When a task τ_L exits from a critical region and it is no longer blocking any other task, its active priority returns back to the baseline priority**.
4. Otherwise, **if τ_L is still blocking some tasks**, this happens when critical regions are nested into each other, **it inherits the highest active priority among them**.

Let us now consider again the example in Fig. 50 and see how the priority inheritance protocol works in this case.
The effects are shown in Fig. 51:

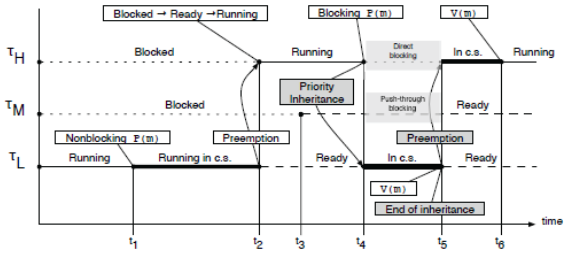

Fig. 51: A simple application of the priority inheritance protocol involving three tasks

- From t_1 to t_4 the system behaves as before. The priority inheritance protocol has not been called into action yet, because no tasks are blocking any other, and all tasks have got their initial, or baseline, priority.
- At t_4, τ_H is blocked by the semaphore primitive `P(m)` because the value of semaphore `m` is zero. At this point, τ_H is blocked by τ_L because τ_L is within a critical region controlled by the same semaphore `m`. Therefore, the priority inheritance protocol makes τ_L inherit the priority of τ_H.
- Regardless of the presence of one (or more) middle-priority tasks like τ_M, at t_4 the scheduler resumes the execution of τ_L because its active priority is now the same as τ_H's priority.
- At t_5, τ_L eventually finishes its work within the critical region R_L and releases the mutual exclusion semaphore with a `V(m)`. This has two distinct effects: the first one pertains to task synchronization, and the second one concerns the priority inheritance protocol:
  1. Task τ_H acquires the mutual exclusion semaphore and returns to the Ready state;
  2. Task τ_L returns to its baseline priority because it is no longer blocking any other task, namely, it is no longer blocking τ_H.

  As a result, the scheduler immediately preempts the processor from τ_L and resumes the execution of τ_H.
- Task τ_H executes within its critical region from t_5 until t_6. Then it exits from the critical region, releasing the mutual exclusion semaphore with `V(m)`, and keeps running past t_6.

It can be proved that **under priority inheritance policy, the time a task spends waiting for a resource** (here represented as a semaphore) **is bounded**.
The following theorem provides **an upper bound for the time spent waiting for resource**:

Theorem. *Let K be the total number of semaphores in the system. If critical regions cannot be nested, the worst-case blocking time experienced by each activation of task τ_i under the priority inheritance protocol is bounded by B_i:*

$$B_i = \sum_{k=1}^{K} \text{usage}(k,i)C(k)$$

where:

- *$usage(k,i)$ is a function that returns 1 if semaphore S_k is used by (at least) one task with a priority less than the priority of τ_i and (at least) one task with a priority higher than or equal to the priority of τ_i. Otherwise, $usage(k,i)$ returns 0.*
- *$C(k)$ is the worst-case execution time among all critical regions corresponding to, or guarded by, semaphore S_k.*

9.5 The Priority Ceiling Protocol

The underlying hypotheses of the original priority ceiling protocol **are the same as those of the priority inheritance protocol**.
The protocol uses additional information for each semaphore: the **ceiling, defined as the maximum initial priority of all tasks that use it**.

As in the priority inheritance protocol, each task has a current (or active) priority that is greater than or equal to its initial (or baseline) priority, depending on whether it is blocking some higher-priority tasks or not. **The priority inheritance rule is exactly the same in both cases**.

A task can immediately acquire a semaphore only if its active priority is higher than the ceiling of any currently locked semaphore, excluding any semaphore that the task has already acquired in the past and not released yet. Otherwise, it will be blocked.
In this situation, **it is therefore possible that a task is blocked even if the requested resource is accessible**.

The priority ceiling protocol has the following properties:

1. **A high-priority task can be blocked at most once during its execution by lower-priority tasks**.
2. **The protocol prevents deadlock**.
3. **Mutual exclusive access to resources is ensured by the protocols themselves**.

In particular, the first property leads to the following theorem:

Theorem. *Let K be the total number of semaphores in the system. The worst-case blocking time experienced by each activation of task τ_i under the priority ceiling protocol is bounded by B_i:*

$$B_i = \max_{k=1}^{K}\{\text{usage}(k,i)C(k)\}$$

where:

- *$usage(k,i)$ is a function that returns 1 if semaphore S_k is used by (at least) one task with a priority less than the priority of τ_i and (at least) one task with a priority higher than or equal to the priority of τ_i. Otherwise, $usage(k,i)$ returns 0.*
- *$C(k)$ is the worst-case execution time among all critical regions corresponding to, or guarded by, semaphore S_k.*

9.6 Response Time Analysis considering locking

The worst-case response time R_i can be redefined to take B_i into account as follows: $\mathbf{R_i = C_i + B_i + I_i}$.

In this way, the worst-case response time R_i of task τ_i is expressed as the sum of three components:

- **the worst-case execution time** C_i,
- **the worst-case interference time** I_i,
- **the worst-case blocking time** B_i.

The corresponding **recurrence relationship** introduced for Response Time Analysis (RTA) becomes:

$$w_i^{(k+1)} = C_i + B_I + \sum_{j \in hp(i)} \left\lceil \frac{w_i^{(k)}}{T_j} \right\rceil C_j$$

The new recurrence relationship still **has the same properties as the original one**. In particular, **if it converges, it still provides the worst-case response time R_i for an appropriate choice of $w_i^{(0)}$. As before, $w_i^{(0)} = C_i$ are good starting points**.

10 Real-Time Linux

10.1 Pre-emptible Kernel

Up to Linux 2.4, **the kernel was not pre-emptible**:

- This means that **a kernel operation had to finish before giving the chance for another task to resume, i.e. before calling** `schedule()`.
- **Interrupts of course occur**, including those the may make a high priority task newly ready, **but the call to** `schedule()` **is deferred until the termination of the pending kernel operation**.

As a consequence, **a high priority task that becomes again eligible for execution may be deferred for an undefined amount of time** since some kernel operation may require hundreds of ms.
Linux 2.4 could therefore not be considered a real-time OS.

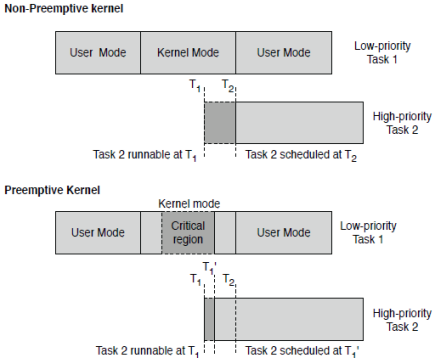

Fig. 52: Latency due to non-preemptible kernel sections

From Linux 2.6 **the kernel becomes interruptible, i.e. when an event occurs that makes a high priority task ready again, control is returned to the latter even**

before terminating the kernel operation that shall be resumed later. In this case **it is possible to avoid in most cases unbounded wait for high priority tasks, making Linux 2.6 a soft real-time system**.
This comes not for free, and it is necessary to **protect critical regions inside the kernel to avoid data structure corruption**.

In Fig. 52 we can see the difference between the two versions of the kernel.

10.2 Kernel Threads

When invoked via **software interrupt** (the only way to move programmatically to kernel mode) **kernel code is running in interrupt context**: as such, **a kernel section cannot be preempted, only suspended by a higher priority interrupt** (i.e. an HW interrupt).

Therefore, it is necessary to let most kernel operations to be carried out by Kernel Threads.
Kernel Threads do not live within the context of any process, but they share the same kernel virtual address.
In this way, using the same scheduler mechanism it is possible to **suspend a kernel thread when another thread/process with higher priority becomes ready, and eventually to resume it**.

Kernel threads are also important in **reducing the amount of code executed at interrupt level in drivers, moving actions that are not essential** (such as writing an immediate answer into the device registers) **to a kernel thread and therefore reducing again the dead time in system responsiveness**. **An interrupt routine can be interrupted only by a higher priority HW interrupt**.

The protection of critical section **cannot rely on the scheduler mechanism because critical sections may be required also by interrupt routines** and therefore **a lower level mechanism is required**.

On a single processor system an immediate way to ensure mutual exclusion is to **disable HW interrupts when the critical section is accessed**.
For multiprocessor (multicore) systems this is no more enough, as **the critical section can be accessed by code running on a different processor**.

In this case **spinlocks** are used, based on shared memory, **looping until the critical section is accessible**: as interrupts are disabled, only a separate processor can be competing for the critical section.
Spinlocks require some sort of atomic Test&Set HW mechanism, that is provided by most, if not all, platforms.

10.3 The PREEMPT_RT Patch

In pre-emptible kernel, **system latency** is mainly due to:

- **critical sections in the kernel code that are protected by spinlocks**; preemption is in fact disabled as long as a single spinlock is active.
- **Interrupt Service Routines (ISRs) running outside any task context**, and therefore potentially **introducing delays in the system reaction to events because the scheduler cannot be invoked until no pending interrupts are present**.

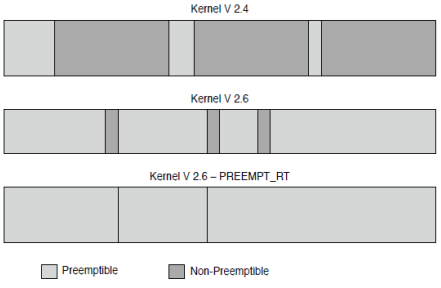

Fig. 53: The evolution of kernel preemption in Linux

The **PREEMPT_RT** Linux patch represents one step further toward **hard real-time** Linux performance.
The PREEMPT_RT patch provides the following features:

- **Preemptible critical sections**
- **Priority inheritance for in-kernel spinlocks and semaphores**
- **Preemptible interrupt handlers**

Real-time performance of the (free) Linux MRG are now very close to those of specialized (expensive) real-time systems such as vxWorks (that has been used in the NASA Mars missions)

The evolution of kernel preemption in Linux is shown in Fig. 53.

In PREEMPT_RT, a **new locking mechanism** is made available. **It is no more implemented as the cyclic atomic test and set but is internally implemented by a semaphore called** `rt-semaphore`.
This semaphore is **implemented in a very efficient way if the underlying architecture supports atomic compare and exchange; otherwise, an internal spinlock mechanism is used**. Therefore, **the task entering a critical section can now be interrupted**.

The integrity of the critical section section is still ensured since, if a new task tries to acquire the same lock, it will be put on wait: **spinlocks are still required for ISR because they are outside the kernel thread context**.

The impact of this different, **semaphore-based mechanism** is not trivial since it **may lead to deadlocks using spinlocks**: in fact, **in the case the task owning a spinlock is put on wait, there would be no chance for other tasks to gain processor usage**.

Priority inversion may lead to potentially unbounded delays in task response even when task priorities (fixed) are accurately planned.
Observe that **for spinlocks priority inversion cannot occur because the task taking ownership disables interrupts and therefore implicitly becomes the highest priority task in the system** (interrupts disabled $\implies$ the task cannot be preempted by higher priority tasks).

For `rt-semaphores` **PREEMPT_RT implements priority inheritance**: the kernel knows the priority of the threads owning the semaphore and of the requesting thread and in case can therefore increment the priority of the current owner, lowering it at its original value when the semaphore is released.

Well-written device driver avoids defining lengthy ISR code. Rather, the ISR code should be made **as short as possible, delegating the rest of the work to kernel threads**.

PREEMPT_RT takes one step further and **forces almost all interrupt handlers to run in task context unless marked SA_NODELAY to cause it to run in interrupt context**.
By default, **only a very limited set of interrupts is marked as SA_NODELAY among them, only the timer interrupt is normally used**.

In this way, **it is possible to define the priority associated with every interrupt source in order to guarantee a faster response to important events**: this would not have been possible if ISR were handled at interrupt level, even when using Interrupt Priority levels (normally limited to 8).