NUMERO: 2420A

ANNO: 2019

A P P U N T I

STUDENTE: Ricci Francesco

MATERIA: Operating Systems - Prof. Maurizio Rebaudengo

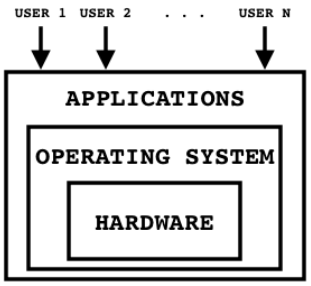
Introduction

Computer: it is a sophisticated electronic calculating machine that:

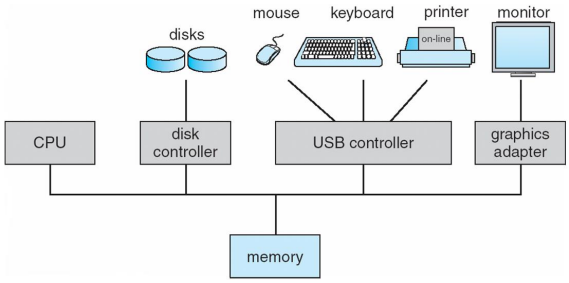
1. accepts input information
2. process the given information (according to a internally stored list of instructions)
3. produce an output information

A computer is composed by 4 components:

1. **Hardware:** provide the basic computing resources
2. **Operating system:** control and coordinates use of hardware among various applications and users. It must protect the hardware from non authorized/malicious actions, while virtualizing it: an application don't care so much about the precise configuration of the current (hardware) system.
3. **Application programs:** define the ways in which the system resources are used to solve the computing problems of the user (word processors, compilers, databases, web browsers)
4. **Users:** could be people, machines or other computers.



Computer-system organization: the today trend is on proving CPU with multiple cores, in order to better parallelize tasks and a shared memory. This memory is connected (and accessed) to device controller(s) by means of a common bus.
Every device can require CPU computation or memory access at any time, for this reason a concurrent execution must be handled.



Basic anatomy of a CPU:

1. **Program counter (PC):** holds memory address of next instruction
2. **Instruction register (IR):** holds instruction currently being executed
3. **Registers (from 1 to n):** hold variables and temporary results
4. **Arithmetic and Logic Unit (ALU):** performs arithmetic and logic operations.
5. **Memory Address Register (MAR):** contains address of memory to be read/written
6. **Memory Data Register (MDR):** contains memory data read or to be written
7. **Stack Pointer (SP):** holds memory address of a stack with a frame for each procedure's local variables & parameters
8. **Processor Status Word (PSW):** contains the mode bit and various control bits

DECODE: the requested data is acquired and stored in the MDR, and the instruction is placed in the IR
EXECUTE: handled by ALU and internal registers

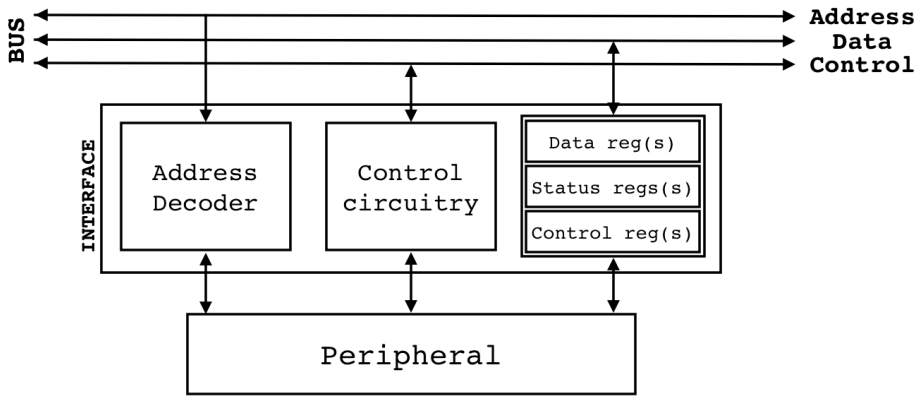
To improve processor core's performance, the pipeline architecture was introduced. A superscalar architecture follow the same basic principle, but is able to issue more than one instruction at every clock cycle by means of redundant functional units inside the processor core.

Peripheral Management and Control: the goal of a general purpose computer is its ability to interact with the environment by means of external I/O devices (keyboard, display, disk-drives, network, etc.). The majority of devices require a fast response from the CPU when various events occur, even when the CPU is busy running a program. For this reason it's needed a mechanism for a device to "gain CPU's attention".

Accessing I/O Devices: Computer system components communicate through an interconnection network. How to address the I/O devices?

1. Memory-mapped I/O: allows I/O registers to be accessed as memory locations.
   - ADV: these registers can be accessed using simply (and only) Load and Store instructions.
   - DIS: the total number of addresses reserved to the memory is reduced. As a result, the memory address space is reduced.
2. Isolated I/O: the addressing spaces for memory and I/O are separated and activated by special signals.
   - ADV: the memory addressing space is untouched.
   - DIS: higher complexity (new signals). The programmer should use specific instructions to access I/O ports (bigger instruction set).

I/O Device Interface: it is a circuit between a device and the interconnection network. It provides the means for data transfer and exchange of status and control information. It includes **data, status, and control registers** accessible with Load and Store instructions (memory-mapped I/O).



Context Switching: An ISR may modify the contents of registers already in use by the main (non-interrupt) code. To prevent corruption of the main code by the ISR a ‘context switch’ may be performed.
A context switch is a section of code that saves and restores important processor information (registers and flags values).

3. **I/O operations:** a running program may require I/O, which may involve a file or an I/O device
4. **File-system manipulation:** the file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
5. **Communications:** processes may exchange information, on the same computer or between computers over a network. Communications may be via shared memory or through message passing (packets moved by the OS)
6. **Error detection:** the OS needs to be constantly aware of possible errors
   - May occur in the CPU and memory hardware, in I/O devices, in user program
   - For each type of error, OS should take the appropriate action to ensure correct and consistent computing.

Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

1. **Resource allocation:** when multiple users or multiple jobs running concurrently, resources must be allocated to each of them. There are many types of resources
   - Special allocation code: for example CPU cycles, main memory, and file storage may have special allocation code
   - General request and release code: for example I/O devices may have general request and release code (2 phase lock mechanism?)

2. **Accounting:** to keep track of which users use how much and what kinds of computer resources

3. **Protection and security:** the owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
   - Protection involves ensuring that all access to system resources is controlled
   - Security of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.

Operating systems goals:

1. **Multiplexing:** A key requirement for an operating system is to support several activities at once: the operating system must time-share the resources of the computer among the programs.
2. **Isolation:** the OS must also arrange for isolation between the processes: if one process has a bug or fails, it shouldn’t affect other programs.
3. **Interaction:** Complete isolation is too strong, since it should be possible for process to interact.

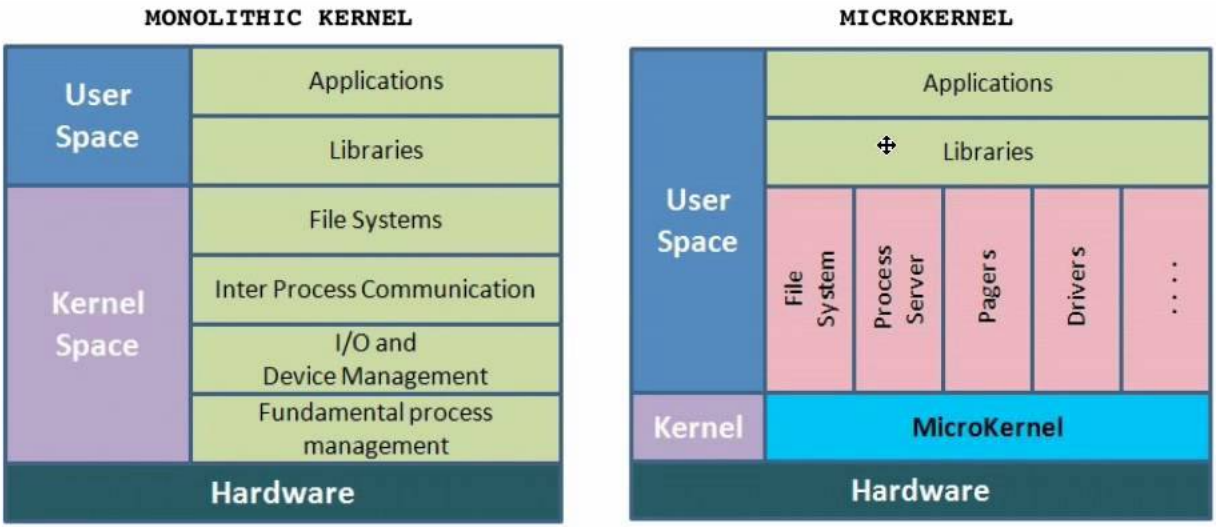
System Calls:

1. Typically written in a high-level language (C or C++)
2. A request to the operating system to perform some activity
3. System calls are expensive: the system needs to perform many things before executing a system call:
   - The computer (hardware) saves its state.
   - The OS code takes control of the CPU, privileges are updated.
   - The OS examines the call parameters.
   - The OS performs the requested function.
   - The OS saves its state (and call results).
   - The OS returns control of the CPU to the caller.
4. System Calls are accessed by programs via a high-level Application Program Interface (API) rather than direct system call use.

Kernel:

Portion of operating system that is in main memory. It contains the most frequently used functions. A kernel can be:

1. **Monolithic kernel** systems provide services like process and memory management, interrupt handling and I/O communication, file system, etc. in kernel space. As the name suggest **it is a single static binary file:** all kernel services exist and execute in the kernel address space.
2. **Microkernel:** the kernel is broken down into separate processes, known as servers. Some of the servers run in kernel space and some run in user-space. All servers are kept separate and run in different address spaces. Servers invoke "services" from each other by sending messages via IPC (Interprocess Communication). This separation has the advantage that if one server fails, other servers can still work efficiently.
   - Modern OS use this approach or an hybrid approach.
   - Examples of servers are:
     a. device drivers
     b. file server
     c. windowing system
     d. security services.



Process Control Block (PCB):

It is the most important data structure in an OS. Contains important process's informations like:

1. **Process identifier:** unique numeric identifier of the process
2. **Process state:** new, ready, running, waiting, terminated
3. **Register Context:**
   - **Program counter** (PC): the address of the next instruction to be executed
   - **CPU registers.**
4. **CPU scheduling information:** needed by the OS to perform its scheduling function.
   - process priority
   - scheduling-related information (e.g., the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running)
   - pointers to scheduling queues
5. **Memory-management information:** mapping between virtual and physical memory locations.
6. **Accounting information:** used for billing purposes and performance measurements:
   - the amount of CPU and real time used
   - main memory storage occupancy
7. **I/O status information:** resources allocated to the process
   - HW units: the list of I/O devices allocated to the process
   - the list of open files.

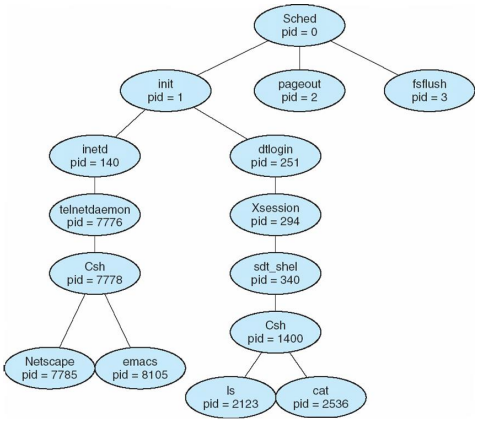
Process table:

Each entry of the process table represent a process and contains the PCB.

Context Switch:

1. When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
2. Context of a process is represented in the PCB
3. Context-switch time is overhead; the system does no useful work while switching
4. Time dependent on hardware support:
   - it depends on the memory speed, the number of registers that must be copied, and the existence of special instructions (e.g., a single instruction to load or store all registers)
   - typical speeds are a few milliseconds.

- Generally, process is identified and managed via a unique process identifier (pid), which is an integer number (a variable of type pid_t).



- Process creation: fork system call creates new process
- Process modification: exec system call (used after a fork) to replace the process's memory space with a new program.

There are some special processes:

1. PID = 0 is usually the scheduler process and is often known as the swapper: it is part of the kernel and is known as a system process.
2. PID = 1 is usually the init process and is invoked by the kernel at the end of the bootstrap procedure.

2. **Process termination:**
   - Process executes the last statement and asks the operating system to delete itself (via the exit() system call or the return instruction executed in the main() function)
   - Parent may terminate the execution of children processes (via a signal kill)
   - Process resources are deallocated by the operating system.

Shared resources: The parent and the child process may:

1. share all the resources (same user level context memo1ry space) or
2. partly share the resources or
3. not share at all the resources (separated user level context memory space).

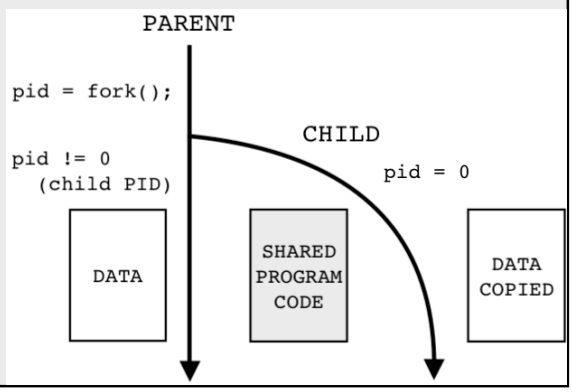
int fork(void)

If the return code is zero for the new (child) process.
If the return code is child PID for the parent process.
If the return code < 0, an error occurred in the process creation.

The two processes (parent and child) share:

- the same program code
- the file descriptors (stdin, stdout and all the open files)
- the user ID, the group ID, the root directory, the working directory

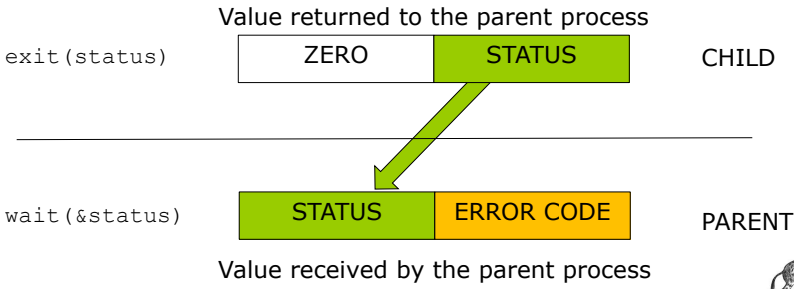
The new process consists of a copy of the address space of the original process.
Both processes share the same value of the Program Counter register: for this reason both processes continue the execution at the instruction after the fork().



NOTE:
The argument to wait()is the address on an integer variable or the NULL pointer.

If it's not NULL, the system writes 16 bits of status information about the terminated child.

Among these 16 bits, the higher 8 bits contain the lower 8 bits of the argument the child passed to exit(), while the lower 8 bits are all zero if the process exited correctly, and contain error information if not.



pid = waitpid(waiting_for_pid, &status, options);

- wait() waits for any child;
- waitpid() waits for a specific child (the one with pid = waiting_for_pid)
- Options: usually 0 (NULL)... but there are many more! (See man).

Read the exit value:

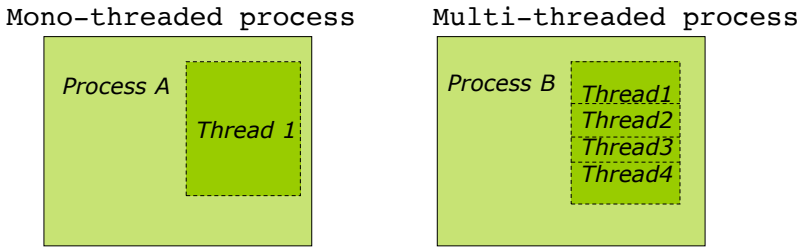
The status value received by the parent from the child via exit() or wait() can be easily read through macros available in <sys/wait.h> (e.g., WIFEXITED, WEXITSTATUS, etc.).

Zombie processes:

A process that terminates cannot leave the system until its parent accepts its return code.
Note: the parent accepts the child's return code either via a wait() or if it terminates
If its parent process is already dead, it'll already have been adopted by the "init" process, which always accepts its children's return codes.
However, if a process's parent is alive but never terminates, the process's return code will never be accepted and the process will remain a zombie.

Multithreaded programming

Single thread vs multithreads: in traditional operating systems, each process has an address space and a single thread of control. There are frequently solutions in which it is desirable to have multiple threads of control in the same address space running concurrently (concurrency: execute different operations in parallel. Those instructions are independent, and in general are found by partitioning the program in a top down approach), as though they are (almost) separate tasks. Programming with threads is a natural way to handle concurrency.



Example of multithreads (I):
Consider how a Web server can improve performance and interactivity by using threads.

1. When a Web server receives requests for images and pages from many clients, it serves each request (from each client) with a different thread.
2. The process that receives all the requests, creates a new separate thread for each request received.
3. This new thread sends the required information to the remote client.
4. While this thread is doing its task, the original thread is free to accept more requests.
5. Web servers are multiprocessor systems that allow for concurrent completion of several requests, thus improving throughput and response time.

Create a number of threads, and for each thread do:

1. get network message from client
2. get URL data from disk
3. send data over network

The advantage of this particular approach is the time saved: the longest wait is due to disk latency, so even if the customer is served in the same amount of time, more customers are served in less time. It means that response time is the same, lead time is improved.

Example of multithreads (II):
Consider a word processor composed of the following threads:

1. a thread interacts with the user
2. a thread handles reformatting the document in the background and performing spell checking
3. a thread handles the disk backups without interfering with the others in order to automatically save the entire file every few minutes to protect the user against losing the work in the event of program crash.

Performance Benefits:

1. It takes far less time to create a new thread in an existing process than to create a new process (100x times faster), because it requires less resources/data structures .
2. It takes less time to terminate a thread than a process
3. It takes less time to switch between 2 threads within the same process than to switch between processes
4. Threads between the same process share memory and files: they can communicate with each other without invoking the kernel.
5. Useful if some threads are I/O-bound => overlap computation and I/O

As a result:

1. If there is an application (or function) that should be implemented as a set of related units of execution, it is far more efficient to do so as a collection of threads rather than a collection of separate processes.
2. Parallel execution: the same process is executed in parallel on a multicore CPU.

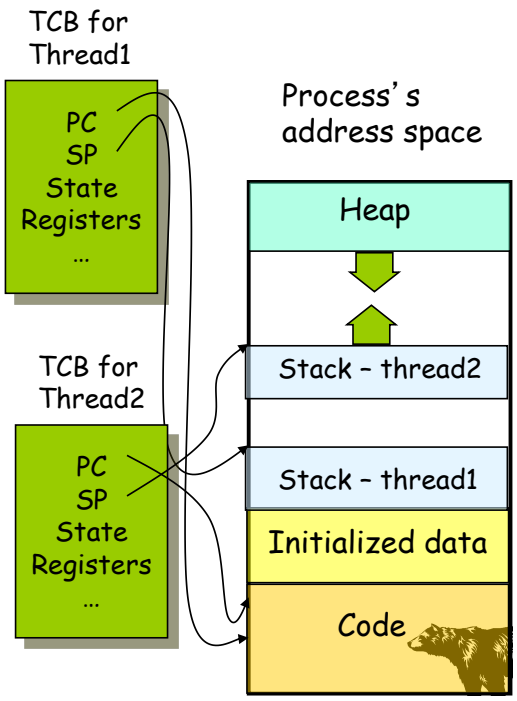
PER PROCESS ITEMS (items shared by all threads in a process)	**PER THREAD ITEMS (items private to each thread)**
Address space	Program counter
Global variables	Registers
Open files	**Stack**
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

NB: signals and alarms are related to a process, not a thread!
NB2: each thread has it's own stack.

Thread Control Block (TCB)
In a multicore architecture, there is the possibility that every processor handle a thread, but also that a single core handles multiple threads (not at the same time: run a bit the first one, than switch to the second one, ecc.). In order to keep track of all threads, we need to store some informations in a particular data structure, called Thread control block. In general, each thread has:

1. an identifier
2. a set of registers (including the program counter and stack pointer)
3. a set of attributes (including the state, the stack size, scheduling parameters, etc).

Implementing threads:



- Processes define an address space
- Threads share the address space

- Process Control Block (PCB) contains process-specific information
  1. Owner
  2. PID
  3. heap pointer
  4. priority
  5. active thread
  6. pointers to thread information

- Thread Control Block (TCB) contains thread-specific information
  1. Stack pointer
  2. PC
  3. thread state (running, …)
  4. register values
  5. a pointer to PCB
  6. ...

Thread Libraries
Thread libraries provide programmer with APIs (Application Programming Interface) for creating and managing threads. There are two primary ways of implementing a thread:

1. **User-Level Threads (ULT)**: library entirely in user space (invoking a function in the library results in a local function call in user space and not a system call). ALL DATA ARE INSIDE USER LEVEL.
2. **Kernel-Level Threads (KLT):** Kernel-level library supported by the OS (system call).

Implemented as a thread library, which contains the code for thread creation, termination, scheduling and switching
Kernel sees one process and it is unaware of its thread activity

- this procedure checks if the thread must be put into blocked state. If so, it saves its context, calls the thread scheduler to pick another thread to run
- the thread scheduler looks in the thread table for a ready thread to run and restores its context

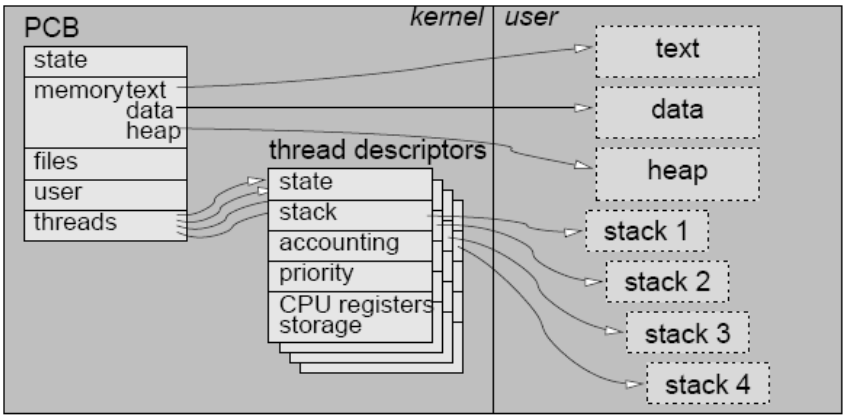
9. When a thread requires the intervention of the kernel (through a system call), the process changes its state and all the threads are blocked.

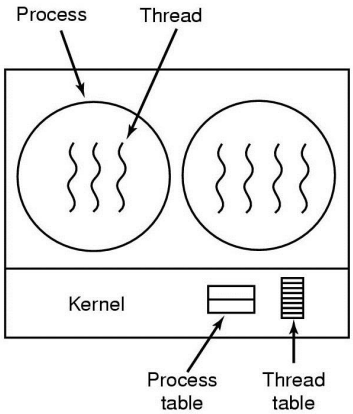
ULT comments:

1. ADV: Fast to create and switch
   - procedures that saves the thread's state and the scheduler are user procedures
   - no system call is needed
   - no context switch is needed
2. DISADV: When a ULT executes a system call, all the threads within the process are blocked E.g., read from file can block all threads
3. DISADV: User-level scheduler can fight with kernel-level scheduler
4. DISADV: A multithread application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time. There are applications that would benefit the ability to execute portions of code simultaneously.

Kernel-Level Threads:

The kernel knows the threads and manages them
There is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system
When a process wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction by updating the kernel thread table
The thread table containing TCBs holds the same information as with the ULT, but now kept in the kernel instead of in user space.





KLT comments:

1. ADV: Kernel-level threads do not block process for a system call
   - if one thread in a process is blocked by a system call (e.g., for a page fault), the kernel can easily check if the process has one thread ready to run
2. ADV: Only one scheduler (and kernel has global view)
3. DISADV: Can be difficult to make efficient (create & switch)

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

Arguments:

1. thread: a unique identifier for the new thread returned by the subroutine (its type is pthread_t): for this reason you pass an address!
2. attr: it specifies a thread attributes object, or NULL for the default values (see later, an example of attribute)
3. start_routine: the C routine that the thread will execute once it is created. It is a function... after that it dies.
4. arg: a single argument that may be passed to start_routine. It must be passed by reference as a pointer cast of type void. Cast back into the start_routine. NULL may be used if no argument is to be passed.

Return value: If successful, it returns 0. Otherwise, it returns a nonzero error code.

Example:

```
#include <stdio.h>
#include <pthread.h>

main() {
     pthread_t f2_thread, f1_thread, f3_thread; // unique identifiers
     int i1=1,i2=2;                             // arguments
     void *f2(), *f1(),*f3();          // prototypes of start_routines
     pthread_create(&f1_thread,NULL,f1,&i1);
     pthread_create(&f2_thread,NULL,f2,&i2);
     pthread_create(&f3_thread,NULL,f3,NULL);   // no argument passed
     ...
}

void *f1(void *i){          // start routine
     int a;
     a = * ((int *) i);
}

void *f2(void *i){
     ...
}
void *f3() {
}
```

Multiple arguments pass in pthread_create()

We can **pass only one parameter to the thread using pthread_create()** system call: to overcome this limitation, we can build a data structure (struct) and pass the pointer to that data structure in the pthread_create().

Thread / Process termination

A whole process (with all its threads) terminates if:
1. one of its threads executes an exit() => If one thread is executing an exit() [remember: exit() means termination of a process!] the entire process terminates.
2. one of its threads receives a signal to terminate
3. the main thread executes a return()

A single thread terminates if:
1. It executes a pthread_exit()
2. It executes a return() in its initial function
3. It receives a pthread_cancel() from another thread.

void pthread_exit(void *ret_val)

1. When a thread has finished its work, it can exit by calling the pthread_exit() library procedure.
2. The pthread_exit() function terminates the calling thread and returns a value via ret_val that (if the thread is joinable) is available to another thread in the same process that calls
3. The thread then vanishes and is no longer schedulable and the stack is released.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h> /* POSIX threads */

void *my_thread(void *);
int sharedVar ;       /* global variable */

int main() {
     pthread_t tid;

     sharedVar = 1234;
     printf("Main: sharedVar= %d\n", sharedVar);
     pthread_create(&tid, NULL, my_thread, NULL);
     sleep(1); /* yield to another thread */

     printf("Main: sharedVar= %d\n", sharedVar);

     sharedVar = 999;
     printf("Main: sharedVar= %d\n", sharedVar);
     sleep(1); /* yield to another thread */

     printf("Main: sharedVar= %d\n", sharedVar);
     printf("DONE\n");
     exit(0);
}
```

```
“BIG” EXAMPLE:

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello (void *threadid) {
      long * tid;
      tid = (long *) threadid;
      printf("Hello World! It's me, thread #%ld!\n", * tid);
      pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
      pthread_t threads[NUM_THREADS];
      int rc;
      long t;

      for(t=0; t<NUM_THREADS; t++) {
            printf("In main: creating thread %ld\n", t);
            rc = pthread_create(&threads[t],NULL,PrintHello,(void *)&t);

            if (rc) {
                  printf("ERROR; return code from pthread_create() is
                                                                  %d\n", rc);
                  exit(-1);
            }
      }
      pthread_exit(NULL);
}
```

This BIG example gives as output the following:

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #1!
In main: creating thread 2
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #4!
Hello World! It's me, thread #5!
```

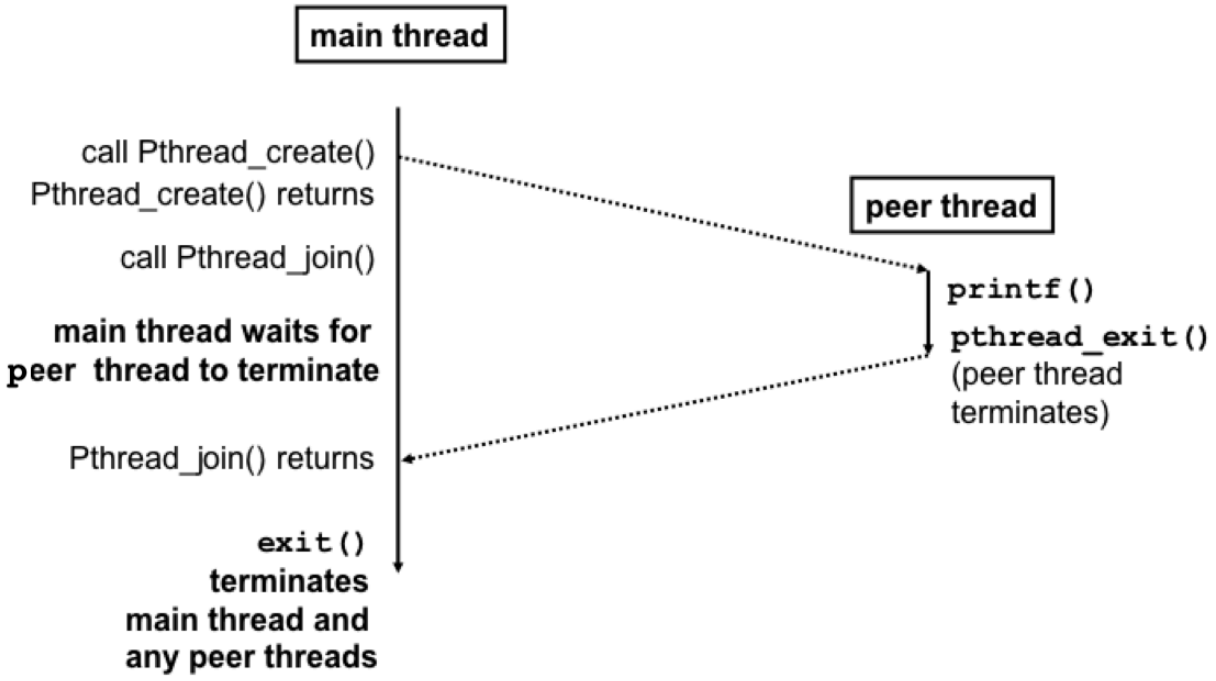
Which is the error? After the main create thread 0, the thread should print on the screen an ‘hello world’ message. However, the thread number isn’t the correct one (I expect from 0 to 4, instead the program displays from 1 to 5).

Why this error occurs? Look the pthread_create(): t is a pointer! During the creation of the thread, the value of t changes! In other words, the for loop is faster than the thread creation. A possible way to fix the program is to implement a local array of numbers (the army contains 0,1,2,3,…). The pthread_create() receive a certain array element, that will remain the same forever, even if the for loop continues to increment the value of t.

This was an example of bad synchronization.

Example: idea is to crate a thread and wait until the end of the thread

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *howdy(void *vargp);

int main() {
     pthread_t tid;

     pthread_create(&tid, NULL, howdy, NULL);
     pthread_join(tid, NULL);    // wait until the created
                                 // thread terminates
     exit(0);
}

void *howdy(void *vargp) {        // start_routine
  printf("Hello, world!\n");
  pthread_exit(NULL);
}
```



Example: implement the summation function **∑x from 1 to argv[1]**, supposing argv[1] >= 0, running it inside a separate thread.

```
#include <stdio.h>
#include <pthread.h>

int sum;                          /* data shared by the thread(s) */
void *runner(void *param);        /* thread's start_routine prototype */

int main(int argc, char *argv[]) {

     pthread_t tid;        // thread ID

     if (argc != 2) {
          fprintf(stderr,"usage: a.out <integer value>\n");
          return -1;
     }

     if (atoi(argv[1]) < 0) {
          fprintf(stderr,"Argument %d must be non-negative\n",
                                                    atoi(argv[1]));
          return -1;
     }

     /* create the thread */
     pthread_create(&tid, NULL, runner, argv[1]);

     /* now wait for the thread to exit */
     pthread_join(tid, NULL);

     printf("sum = %d\n",sum);
}

void *runner(void *param) {
     int i, upper = atoi(param);
     sum = 0;

     for (i = 1; i <= upper; i++) {
          sum += i;
     }

     pthread_exit(0);
}
```

NOTES:

1. **atoi** : make the transaction form string to number (int).
2. argv[1] : parameter that I pass to the program before the run phase (if any).
3. argv[0] contains the name of the program.

EXECUTION: in the main function, an attribute is passed. If it is >= 0, a thread is created, and that number is passed as an argument. The main will wait until the thread terminates (pthread_join). The tread perform the summation function (for loop) and then exit. The total is stored in a global variable, so both main thread and secondary thread can access it. Finally the main thread prints the result on the screen.

int pthread_cancel (pthread_t tid)

It terminates the specified thread (it has a pthread_exit embedded), identified by the thread identifier tid. It returns zero in case of success, otherwise an error code, in case of failure.

int pthread_detach (pthread_t tid)

The pthread_detach() function marks the thread identified by tid as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.

A detached thread cannot be joined: all the future calls pthread_join to thread tid will fail with an error code.

Parameter: tid is a thread identifier
Return value: 0, in case of success; an error code, in case of failure.

Process Synchronization

Concurrency: The system must support concurrent execution of threads

1. Scheduling: it deals with execution of "unrelated" threads
2. Concurrency: it deals with execution of "related" threads

Why is it necessary?

1. **Cooperation:** One thread may need to wait for the result of some operation done by another thread (e.g. "Calculate Average" must wait until all "data reads" are completed)
   - Each process is aware of the other
   - Processes Synchronization
     Exchange information with one another (Shared memory, Message passing)

2. **Competition:** Several threads may compete for exclusive use of resources (e.g. two threads trying to increment the value in a memory location)
   - Processes compete for resources
   - Each process could exist without the other

Resources: An object, necessary to a task in order to be executed.

1. Hardware resources:
   - Co-processor
   - I/O system (e.g., printer, disks)
   - memory
   - network
2. Software resources:
   - buffer memory space
   - portion of code source.

Mutual Exclusion: this is a fundamental concept. It requires some definitions first:

1. **Critical resource:** a resource not shareable for which sharing by the threads must be controlled by the system.
   (e.g. "Calculate Average": a shared variable 'cnt' could be shared among threads. Is important however that only one thread at a time can access/manipulate that resource. If a thread has not finished the manipulation of the resource, and another thread reads it, it will cause unexpected behavior and hard-to-replicate bugs.

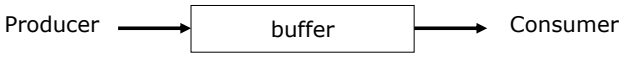
2. **Critical section (or critical region) of a program:** a part of a program (= portion of code) where access to a critical resource occurs.

3. Mutual exclusion: If one thread is going to use a shared (= critical) resource (like a file a variable, printer, register, etc) the other threads must be excluded from using the same resource. It means that **when a process is inside a critical section that contains the critical resource 'a', all the other processes that share the critical resource 'a' must not enter a critical section.**

NB: a child inherit the same open files of the parent.

Example: Producer-Consumer problem

In this example 2 processes are involved: a producer process produces information that is consumed by a consumer process later. Since the producer and the consumer can work at different speeds, a buffer is needed where the producer can temporarily store data that can be retrieved by the consumer at a more appropriate speed.



The buyer can store a certain number of informations, and an integer 'count' keeps track of the number of full positions of the buffer. Initially, count is set to 0. It is incremented by the producer after it produces a new information that store into the buffer. It is decremented by the consumer after it consumes an information stored into the buffer. The buffer is treated as a circular array: two integers (in and out) points respectively to the next free position and the fist full position.

If count reaches the size of the buffer (buffer is full of info) the producer has to wait; if the buffer is empty (count = 0) the consumer has to wait. The following is a section of example's code:

Bounded buffer

```
#define BUFFER_SIZE 10
typedef struct {
      ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer

```
while(1) {
      /* produce an item and put in nextProduced */
      while (count == BUFFER_SIZE)
            ; // do nothing
            buffer [in] = nextProduced;
            in = (in + 1) % BUFFER_SIZE;
            count++;
}
```

Consumer

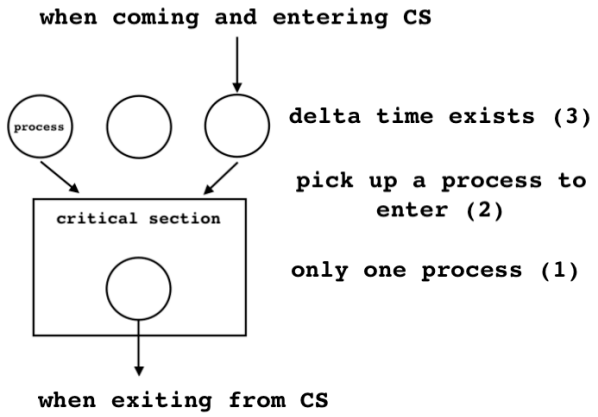
```
while (true) {
      while (count == 0)
            ; // do nothing
            nextConsumed = buffer[out];
            out = (out + 1) % BUFFER_SIZE;
            count--;

            /* consume the item in nextConsumed
}
```

3. **Exit Section:** the end of the critical section, releasing or allowing others in.
4. **Remainder Section:** rest of the code after the critical section.

Solution to critical-section problem: the critical section must enforce all 3 of the following rules:

1. Mutual Exclusion: if process P_i is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress: if no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely. (deadlock represent a violation of progress).
3. Bounded Waiting: a bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. (starvation represent a violation of bounded waiting).

   Speed and Number of CPUs: No assumption may be made about speeds or number of CPUs.



Concurrency requirements:

1. Among all threads that have CSs for the same resource, only one thread at a time is allowed into its CS
2. It must not be possible for a thread requiring access to a CS to be delayed indefinitely => no deadlock, no starvation
3. When no thread is in a CS, any thread requesting entry to the CS must be granted permission without delay
4. No assumptions are made about the relative thread speeds or number of processors.
5. A thread remains inside its CS for a finite time only.

Other ways to report the rules: requirements for CS problem:

1. **Safety** (aka mutual exclusion): guarantee eta are correct, by allowing only one process/thread at a time inside CS
2. **Liveness** (aka progress): if nobody has access and somebody wants to get in, somebody gets in
3. **No starvation** (aka bounded waiting): if you want to get in, you will eventually get in

SOFTWARE SOLUTIONS:

ALGORITHM 0: use of shared variable 'inside'.

```
char inside = 0; // boolean variable
do {
     while (inside) continue;
     inside = true;
     /* critical section */
     inside = false;
     /* reminder section */
} while (1)
```

MUTUAL EXCLUSION VIOLATION: since a scheduler is present, the order of execution of the instructions of the two processes is not known. Most of the time the two process instructions are interleaved. The code is unsafe, because a situation like this could happen:

```
// inside = false
Thread i: while(inside) continue;     // the thread enter the critical
                                      // section, but has not set
                                      // inside = true yet.

Thread j: while(inside) continue;     // also thread j enters the
                                      // critical section, because
                                      // inside is still false.
```

Both access the critical section at the same time!

PROGRESS: ok
BOUNDED WAITING: for small number of processes is ok

ALGORITHM 1: use of shared variable turn;

```
char turn = i; // Pi can enter CS
do {
     // CS is accessed by j
     while (turn != i);
     /* critical section */
     turn = j;
     /* reminder section */
} while (1);
```

```
do {
     // CS is accessed by i
     while (turn != j);
     /* critical section */
     turn = i;
     /* reminder section */
} while (1);
```

MUTUAL EXCLUSION: ok
PROGRESS VIOLATION (STARVATION): there is a strict order of execution of the processes: in fact the scheme is P0-P1-P0-P1.. (in case f 2 processes). This scheme can cause STARVATION! In fact there is no guarantee that such a certain scheme is always followed. For example: process P0 enters the CS rarely, instead P1 wants to access CS frequently. P1 has to wait until P0 re-enter the CS, so waste a lot of time! (The resource is available, so is not deadlock, but starvation).

HARDWARE SYNCHRONIZATION:
Any solution to the CS problem requires a simple tool: **a lock.**
Race conditions are prevented by requiring that critical regions be protected by locks.

```
do {
      acquire lock
      /* critical section */
      release lock
      /* remainder section */
} while (TRUE);
```

We want to define an hardware solution that guarantees that the lock is used by one single process at a time.

Lock: prevents someone from doing something
1. Lock before entering critical section and before accessing shared data
2. Unlock when leaving, after accessing shared data
3. Wait if locked: all synchronization involves waiting.

Hardware Synchronization:

1. **Disabling interrupts:**
   - It disable context switching too (switching from a process to another require a specific interrupt).
   - It cannot be done for long (or some interrupts will be lost)
   - User processes are not allowed to do that
   - I/O may be needed while in a critical section (and it will never be completed with interrupts disabled)
   - It disables even time interrupts, thus not allowing preemption
   - It does not work in a multi-processor system, anyway! (Mutual ex. not preserved!)
   - Mutual exclusion is guaranteed
   - Degrades efficiency because processor cannot interleave threads when interrupts are disabled
   - Simplest solution, but is not desirable to give a thread the power of controlling interrupts.
   - Generally not an accepted solution
2. **Test and set:**
   - the hardware lock is done with a binary flag (which indicates if the resource is free or not).
   - Each task will test the flag before using the resource: if it is already used by another task, it will do a busy waiting, otherwise the task will set the flag 'busy' before entering the CS and will set the flag 'free' after the exit from the CS.
   - Is important the usage of a sort of atomic operation that can test the flag and set the 'busy' value. (Otherwise two interleaved processes can read the 'free' lock and enter CS).
   - Modern processors usually have a "TEST and SET" instruction allowing to do the first access to the memory (read the flag value) and the second access to the memory (write 'busy' in the flag) in a concatenated way.

3. **Compare & swap:** see slides to learn more about it. We not studied this argument in depth this year.
4. **Special atomic hardware instructions:** by definition an atomic operation is an operation that always runs to completion (without

OS SOLUTIONS:

Semaphores

Hardware solutions are too complex for the application programmer; a new synchronization tool, called semaphore, has been proposed (by professor Edsger Dijkstra, 1965). A semaphore is a data structure (typically a integer variable, always protected by the OS).

Semaphores do not require busy waiting! There are two standard operations modify a semaphore S:

- wait() (originally called: P())
- signal() (originally called: V())

The OS must guarantee that those two operations are atomic. It means that the assembler operations required to execute a wait() or signal() must never be interleaved with something else. [wait and signal are CS and should be protected].

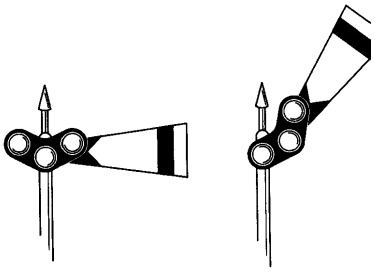
There are **two types of semaphores: counting** (the int value of semaphore can range over unrestricted domain. The more is higher, more processes are waiting) **and binary** (semaphore int value can only assume 0 or 1).

Binary semaphore:

1. As said before, the integer value can range only between 0 and 1
2. Binary semaphores are known also as **mutex** locks as they are locks that provide **mut**ual **ex**clusion.
3. We can use binary semaphore to deal with the CS problem for multiple processes.
4. The n processes share a semaphore, mutex, initialized to 1

THREAD A THREAD B

semaphore mutex = 1 -- unlocked



```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

semaphore mutex = 0 -- locked

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```

```
1 repeat
2    wait(mutex);
3    critical section
4    signal(mutex);
5    remainder section
6 until FALSE
```


© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 49 di 108

Semaphore implementation:

1. The main disadvantage of Test&Set is the busy waiting. The simplest semaphore implementation (called **spinlock**) has this issue.
2. To overcome this limitation, the concept of block() and wakeup() are introduced:

```
Typedef struct {
     int value;
     struct process *list // pointer to the waiting list
} semaphore
```

- block() : place the process into the appropriate waiting list (each semaphore has its own one).
- wakeup() : remove one process from the waiting list and place it in there ready queue
- Block and wakeup are used into the wait() and signal() functions

Implementation of wait()

```
wait (semaphore *S) {
     S->value--;
     if (S->value < 0) {
         add this proc to S->list;
         block();
     }
}
```

Implementation of signal()

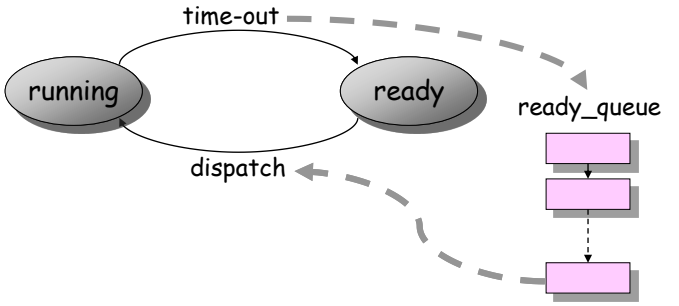
```
signal (semaphore *S) {
     S->value++;
     if (S->value <= 0) {
       remove a proc from S->list;
       wakeup(P);
     }
}
```

1. Note that the semaphore value may be negative.
2. However, in order to enforce mutual exclusion, the value should be less or equal than 1. It can assume any negative number (see point 3).
3. Its magnitude is the number of processes waiting on that semaphore.
4. The list of waiting processes can be easily implemented by a link field in each process control block (PCB).

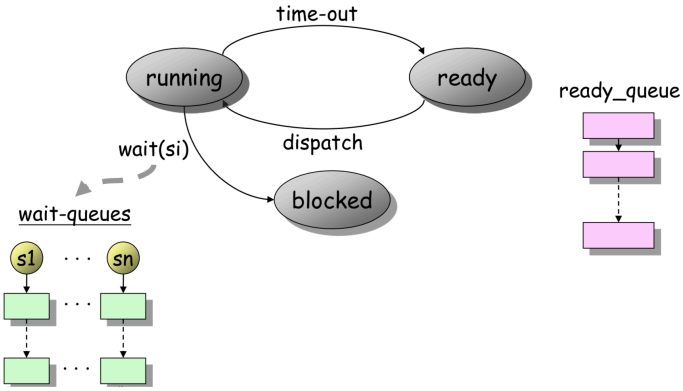
Advantages:

1. OS guarantees that Wait and Signal are atomic
2. Programmer does not need to worry about interleaving within the entry and exit sections.
3. No spin locks
4. Semaphores are machine-independent
5. They are simple but very general
6. They work with any number of processes
7. We can have as many critical regions as we want by assigning a different semaphore to each critical region

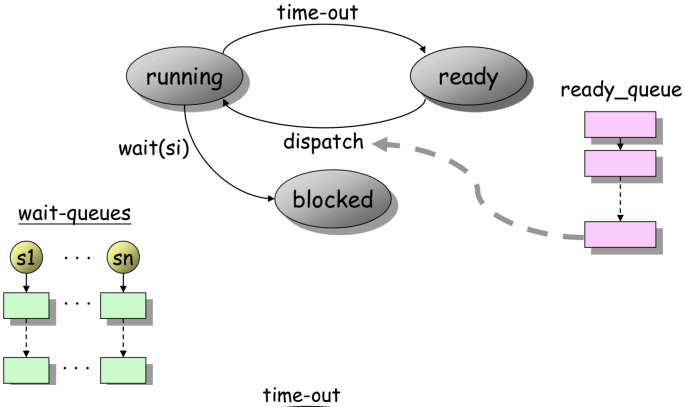
OS designer must use the features of the hardware to provide semaphores. Key point: it means that hardware strategies are used, and they may vary from machine to machine, but the programmer still use the same wait() and signal(). It is convenient to the programmer, but is not a "solution".



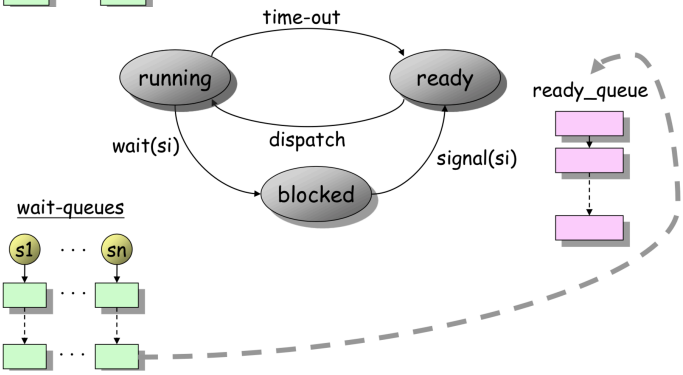
A process is in running state. Because there are many other processes, after a certain time-out, that process is moved into a waiting state (fairness). After a while, it will return into the running state (dispatch).



A running process wants to access a CS. It looks at the semaphore: if it is already locked, the process is moved to a 'blocked' state. In this state, the process does not continue to waste cpu time. (Using wait(s) system call, there is not busy waiting).



Of course, when a process complete its execution, it expires, and a new process is dispatched from the ready_queue into the running state.



The signal(si) moves a **single** process form a waiting state (waiting_queue) to the ready_queue. Sooner or later the scheduler will dispatch it, moving it to the running state.

Implementation of semaphores:

1. Any software schemes, such as Peterson's algorithm, can be used to implement semaphores. However, the busy-waiting in each of them imposes a large overhead.
   - Recall that the Peterson's algorithm discussed above involves only two processes. Generalizing this algorithm for n processes to implement general semaphores has a **large overhead.**
2. Hardware implementation based on
   - test and set instruction
     a. the busy-waiting in the V() and P() operations are relatively short
   - disabling interrupts
     a. there is no wait loop, but this approach works only on a single processor system.

Implementing a binary semaphore by Test and Set:

```
Typedef struct {
     int value;
     struct process *list; // pointer to waiting_list
} semaphore
```

Initial values:
- S.value = 1
- S.flag = 0
- S.L = NULL

```
wait(S):

repeat until test-and-set(S.flag)
if (S.value == 0) {
     add this process to S.L;
     sleep & S.flag=0;
} else {
     S.value = 0;
     S.flag=0;
}
```

```
signal(S):

repeat until test-and-set(S.flag)
if (queue is not empty) {
     wakeup(); // move proc from
               // blocked 2 ready
} else {
     S.value = 1;
     S.flag=0
}
```

Implementing a counting semaphore by Test and Set:

```
Typedef struct {
     int value;
     struct process *list; // pointer to waiting_list
} semaphore
```

Initial values:
- S.value = 1
- S.flag = 0
- S.L = NULL

```
wait(S):

repeat until test-and-set(S.flag)
S.value--;
if (S.value < 0) {
     add this process to S.L;
     sleep & S.flag=0;
} else {
     S.flag=0;
}
```

```
signal(S):

repeat until test-and-set(S.flag)
S.value++;
if (S.value <= 0) {

     wakeup(); // move proc from
               // blocked 2 ready
}
S.flag = 0;
```

NB: flag=0 means the lock is unlocked (in test&set works opposite way).

Condition variables:

A condition variable is a **queue** that processes can put themselves on when some state of execution (i.e. some **condition**) is not reached (by **waiting** on the condition)

They support two operations:

1. **cond.wait():** suspends this process until signaled (it sleeps on the waiting queue associated to the condition variable)
2. **cond.signal():** wakes up one process waiting on the condition variable
   - If no process is waiting, signal has no effect (e.g., signals on condition variables aren't "saved up").

Condition variables, with respect to semaphores, have no initial state (S.value in not initialized).
The process executing a cond.signal() will actually wake up another process (if any is waiting, otherwise any effect).

Typically the condition variable works within a critical section; we need to protect the condition variable with a mutex semaphore...
The operations work in that way:

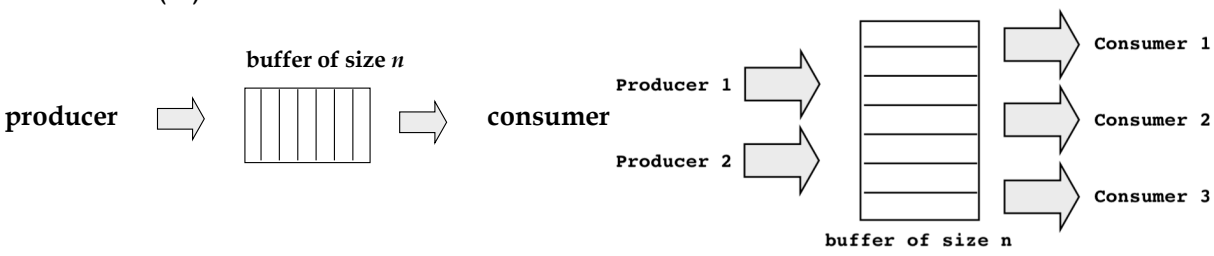
1. **cond.wait(condition, mutex):**
   - the process is suspended (sleeping on the waiting queue associated to the condition variable)
   - It releases the critical section lock (mutex)
2. **cond.signal(condition):** wakes up one process waiting on the condition variable
   - when that condition variable is signaled, a sleeping process will become ready again; it will attempt to reacquire that critical section lock and only then it will be able to proceed.

Consumer/Producer with Bounded Buffer:

We already tacked this problem before. Now ew are interested on a more general version of it. In fact, the problem could be generalized in the case of more than one producer, and more the one consumer. Is important the synchronization between all producers and between all consumers.

Example: if consumers are not synchronized, 2 process could consume the same data, causing an unexposed program behavior.

Recall: Two processes (at least) use a shared buffer located in memory. The buffer has a fixed non infinite size (in other words, is bounded). The producer(s) writes on the buffer and the consumer(s) reads from it. A full buffer stops the producer(s), an empty buffer stops the consumers(s).



Correctness Constraints:

1. Consumer must wait for producer to fill buffers, if empty (scheduling constraint)
2. Producer must wait for consumer to empty buffers, if full (scheduling constraint)
3. **Only one thread can manipulate buffer queue at a time (mutual exclusion)**

The solution adopt 3 Semaphores:

1. mutex (initially mutex = 1, means unlocked)
2. full (counting semaphore) (initially full = 0) # of full cells
3. empty (counting semaphore) (initially empty = n) # of empty cells

Like in the previous chapter implementation, two pointers are used:

1. in : points to the first possible empty space in the buffer
2. out : points to the oldest data in the buffer still not consumed

Lets introduce two functions: append and take are CS because they use the shared buffer. We protect them with the three previously described semaphores.

```
void append(int v) {
     b[in] = v; // write new data in buffer
     in = (in+1)%N; // updates pointer 'in' value
}
```

```
int take(void) {
     w = b[out]; // read data from buffer
     out = (out+1)%N; // updates pointer 'out' value
     return w;
}
```

NB: the data type of v could be different from int.
Just be consistent in the two functions.

Readers-Writers problem:

A data set is shared among a number of concurrent processes. They can be divided into two groups:

1. Readers : only read the data set; they do not perform any updates. Many readers may access a database without fear of data corruption (interference)
2. Writers : can both read and write.

Correctness constraints:

1. Many readers can read at the same time: any writer can be involved.
2. Only one writer can write on the shared resource at the same time: any writers and reader can be involved. Each writer has exclusive access.

Possibile implementations:

1. Readers wait only if a writer has already obtained access permission **(no reader will be kept waiting if there are writers waiting)**
2. Writers have priority, start right away, temporarily blocking readers. Once a writer is ready, that writer has to perform its write as soon as possible, after old readers (or writer) are completed. Thus, **if a writer is waiting to access the object, no new readers may start reading.**

A solution to either problem may result in starvation.

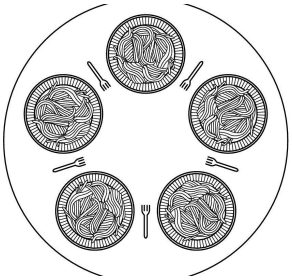
1. with reader precedence: Writers
2. with writer precedence: Readers.

Possible solution (implementation 1):

Shared Data:

1. data set (can be a file, a struct or just a variable)
2. int readcount (initially readcount= 0) (current number of readers)
3. semaphore mutex (initially mutex = 1) (protect readcount updates)
4. semaphore wrt (initially wrt = 1) (protect exclusion of writers)

Dining Philosophers Problem: model allocating several resources among several processes.



Plot: Five philosophers are seated around a circular table. On the table there are five plates of spaghetti, and five forks, one between each pair of plates.
Philosophers spend most of the time thinking and sometimes they eat. When they want to eat, they need two forks. They pick up one and then the other one, not both of them at the same time.
Each processes is modeled with a thread.

Wrong implementation (deadlock):

Shared data: semaphore fork[5] (initially {1,1,1,1,1} => all unlocked)

```
do{
      wait(fork[i]);         // get left fork
      wait(fork[(i+1)%5]); // get right fork

      /* eat */

      signal(fork[i]);        // return left fork
      signal(fork[(i+1)%5]); // return right fork

      /* think */

} while(true);
```

Deadlock:
Due of scheduler instruction interleaving, if all five philosophers take their left fork simultaneously, no one will be able to take the right fork and the program will remain blocked forever.

Possible solutions:

0) Teach philosophers to eat spaghetti with 1 single fork (not feasible)
1) Allow at most 4 philosophers at the table. Still will be 5 plates and 5 forks
   - At least one process will have access to both forks, it will leave the table, giving possibility to another process to take two forks...
   - Implemented with a counting semaphore.
2) Use asymmetry – odd philosophers pick up left first, while even philosophers pick up right first
3) Check to see if both forks are available, then pick them up
   - Make the two consecutive take_fork() a single atomic operation take_forks(). Same for put_forks()
   - Add an additional state: THINKING, **HUNGRY**, EATING. Hungry means the philosopher want to eat, but forks are not available yet.

Semaphore Operations:

1. Semaphore variable is of type *sem_t*
2. Atomic operations to initialize, increment and decrement value
3. Semaphore functions start with *sem_*
4. All semaphore functions: if successful, return 0, otherwise they return -1 and sets errno to indicate the error.

```
#include <semaphore.h>
      int sem_init     (sem_t *sem, int pshared, unsigned int value);
      int sem_destroy  (sem_t *sem);
      int sem_wait     (sem_t *sem);
      int sem_trywait  (sem_t *sem);
      int sem_post     (sem_t *sem);
      int sem_getvalue (sem_t *sem, int *sval);
```

int sem_init (sem_t *sem, int pshared, unsigned int value)

Initializes semaphore to value parameter.
pshared is the type of semaphore:
- if 0, then semaphore is local to the current process (used by threads of the same process).
- else the semaphore may be shared between processes (for example process and child). The semaphore should be located in a region of shared memory. Any process that can access the shared memory region can operate on the semaphore using sem_post, sem_wait, etc.

int sem_getvalue (sem_t *sem, int *sval)

Allows the user to examine the value of a semaphore. It sets the integer referenced by sval to the value of the examined semaphore.

int sem_destroy (sem_t *sem);

Destroys a previously initialized semaphore.
If sem_destroy attempts to destroy a semaphore that is being used by another process, it may return –1.

int sem_wait (sem_t *sem)

Classical semaphore wait operation. If the semaphore value is 0, sem_wait blocks until it can successfully decrement value.

int sem_trywait (sem_t *sem)

Similar to *sem_wait* except instead of blocking on 0, it returns –1 and sets errno to EAGAIN.

CPU scheduling

1. The objective of **multiprogramming** is to have some process running at all times, to maximize the CPU utilization. In general, one of the OS's goals is to manage resources (one of which is the CPU).
2. The objective of **time sharing** is to switch the CPU among processes so frequently that users can interact with each program while it is running.
3. To meet these objectives, the **process scheduler** selects an available process for program execution on the CPU. The scheduling is important both in multi-cores and single-core systems.
4. **Virtual memory:** it is a very common technique. The goal is to have multiple processes run together, so is important to have them inside the main memory. If a process is very long (= code size is high), it may saturate the main memory, denying other processes to be executed. Virtual memory size is grater than physical main memory, because it includes also a section of secondary memory. This approach gives two advantages:
    - I don't need a physical main memory large as the program I want to run. Without this technique a long process cannot be executed, because it doesn't fit in memory.
    - Temporal and spacial locality: is not necessary to store all the code and resources of the single process/program always in the RAM; a large part is stored on the secondary memory. Because of temporal and spacial locality, the RAM misses remains low.

Process Scheduling Queues:

1. Job queue: set of all processes in the system
2. Ready queue: set of all processes residing in main memory, ready and waiting to execute
3. Device queues: set of processes waiting for an I/O device. Each device has its own device queue.
4. Processes migrate among the various queues.
5. Queues are generally stored as linked lists

Schedulers

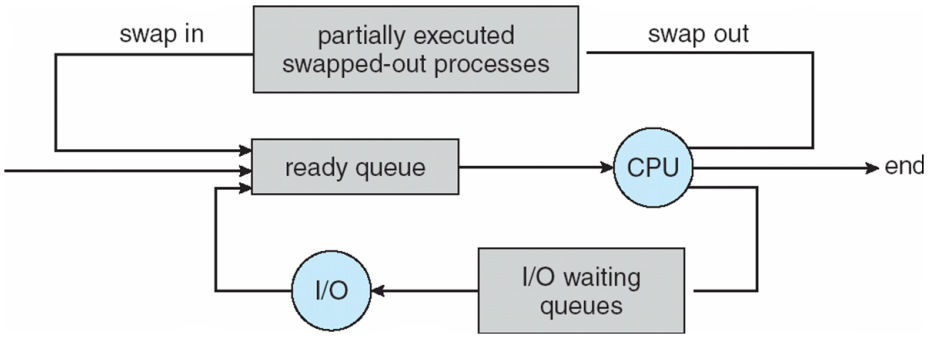
1. **Long-term scheduler** (or job scheduler): selects which processes should be brought into the ready queue.
    - In a batch system active processes are **spooled** to a mass-storage device, where they are kept for later execution. The long-term scheduler selects processes from this pool and loads them into memory for execution.
    - It is invoked not very frequently (seconds, minutes). Because of the longer interval between executions, the long-term scheduler can afford to be slower (w.r.t. short-term scheduler), and can apply more complex policies to gain some performance advantages.

Medium Term Scheduler:

Some operating systems may introduce an additional, intermediate level of scheduling.

Swapping scheme: It could be advantageous to remove processes from memory and thus reduce the level of multiprogramming. Later, the process could be reintroduced in memory, and its execution can be continued where it left off.

Swapping scheme is convenient because **performance does not depends on the total number of programs but by their distribution** (remember that best performance requires balance between I/O and CPU bound program). If the balance changes too much, the medium scheduler can re-establish an equilibrium by moving one or more processes in another queue (swapping operation).



To summarize:

1. **Long-term scheduler:**
   - Determines which programs are admitted to the system for processing (and so controls the degree of multiprogramming).
   - Attempts to keep a balanced mix of processor-bound and I/O-bound processes (optimize system performance and CPU usage).
2. **Medium-term scheduler:**
   - it makes swapping decisions based on the current degree of multiprogramming, in order tore-establish an equilibrium between CPU-bound and I/O bound processes.
3. **Short-term scheduler:**
   - selects from the ready queue in memory which process to execute next.
   - It is invoked on events that may lead to choose another process for execution: clock and I/O interrupts, operating system calls and traps, signals.

Preemptive vs. nonpreemptive scheduling:

Many CPU scheduling algorithms have both preemptive and nonpreemptive versions:

1. **Preemptive:**
   - schedule a new process even when the current process does not intend to give up the CPU
   - Currently running process may be interrupted and moved to the Ready state by the OS
   - Prevents one process from monopolizing the processor
2. **Non-preemptive:**
   - Once a process is in the running state, it will continue until it terminates or blocks for an I/O
   - only schedule a new process when the current one does not want CPU any more.
3. CPU scheduling decisions may take place when a process:
   - Switches from running to waiting state (I/O request)
   - Switches from running to ready state (CPU receives an interrupt)
   - Switches from waiting to ready (completion of an I/O operation)
   - Terminates

   Scheduling under 1 and 4 is nonpreemptive;

   Scheduling under 2 and 3 is preemptive.

Scheduling Policies:

The dispatcher implement a scheduling policy. The policy depend on the requirements; here are some of the most used policies:

1. FCFS (first come, first served)
2. SJF (shortest job first)
3. PriorityScheduling
4. Round robin
5. Multilevel queues
6. Multilevel feedback queues

First Come First Served (FCFS):

1. FIFO: assigns a process to the CPU based on the order of arrival
2. Non-preemptive: A process keeps running on a CPU until it is blocked or terminated.
3. Short jobs can get stuck behind long jobs (convoy effect): when a process is moved to the read list, must wait until all previously arrived jobs are completed.
4. Very simple to implement
5. Turnaround time is not ideal

Shortest Job First (SJF):

1. SJF runs whatever job puts the least demand on the CPU, also known as STCF (shortest time to completion first).
2. Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
3. Advantages:
   - Provably optimal in terms of turnaround time
   - Gives minimum average waiting time for a given set of processes
   - Great for short jobs
   - Small degradation for long jobs
4. Disadvantages:
   - The difficulty is knowing the length of the next CPU request: can be estimated using the length of previous CPU bursts, using exponential averaging.
   - Non-preemptive

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n$$

t: actual length of burst
tau: estimated length

Shortest remaining time first (SRTF):

It is a preemptive version of SJF. If a new job arrives in the ready queue with a shorter time to completion than the current executing process, SRTF preempts the CPU for the new job.

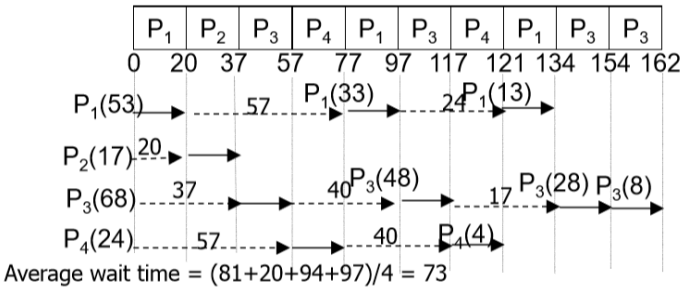
© Proprietà riservata dell'autore - Digitalizzazione e distribuzione a cura del CENTRO APPUNTI - Corso Luigi Einaudi, 55 - Torino / Pagina 73 di 108

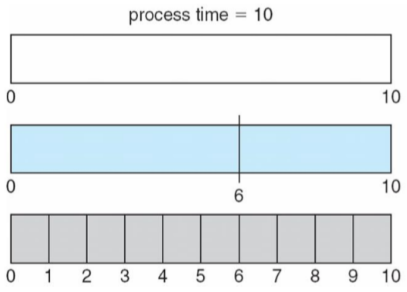
Round Robin (RR):

1. **Goal is an equal distribution of processing time:** n processes running, each gets 1/n of the CPU time. RR execute the processes in order.
2. Round Robin periodically releases the CPU from long-running jobs
   - Based on timer interrupts
   - Preemptive: a process can be forced to leave its running state and replaced by another running process
   - Time slice (or time quantum): interval between timer interrupts (usually 10-100 milliseconds)
   - Most widely used scheduling algorithm
3. After the time slice has elapsed, the process is preempted and added to the end of the ready queue. In this way, the ready queue is treated as a circular queue.
4. If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units at once (if the process is shorter than the time quantum, the switch is anticipated). No process waits more than (n-1)q time units. If the list of ready processes is very long, every single process will wait a lot of time.
5. If time slice is too long: scheduling degrades to FCFS. Long processes not released until termination.
6. If time slice is too short: context switching cost dominates (overhead)
7. Time quantum set to ~100 milliseconds (usually)
8. Context switches typically cost < 1 millisecond. Context switch is usually negligible (< 1% per time slice).

Example: suppose time quantum Q = 20 and suppose ready queue contains processes P1, P2, P3, P4. If process have length reported in the table:

Process	Burst time	Wait time
P1	53	57 +24 = 81
P2	17	20
P3	68	37 + 40 + 17= 94
P4	24	57 + 40 = 97





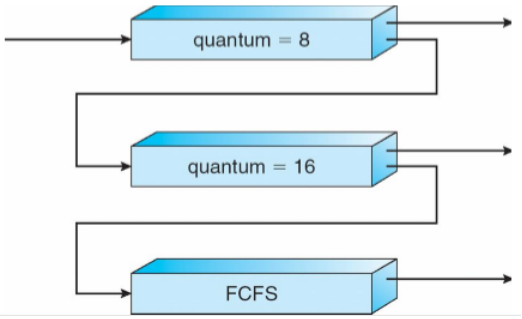
quantum	context switches
12	0
6	1
1	9

NB: by decreasing the time quantum size, the number of context switch increase. This approach can benefit short processes, but also introduce an overhead. On the right, there is a graphical representation of quantum...

Multilevel feedback queue:

1. Multilevel feedback queues algorithm uses multiple queues with different priorities
   - Round robin at each priority level
   - Run highest priority jobs first
   - Once those finish, run next highest priority, etc
   - Jobs start in the highest priority queue
   - If time slice expires, drop the job by one level
   - If process waits for I/O, push the job up by one level
2. We can move processes between queues!
3. Approximates SRTF
   - A CPU-bound (with long CPU bursts) job drops like a rock (not very effective)
   - I/O-bound (with short CPU bursts) jobs stay near the top
   - Unfair for long running jobs: a counter-measure is aging. Aging technique keep track of the last time a process was served and increase the priority (of long running jobs) accordingly.

Example: pag 69



queue on the top has the highest priority, but also the shortest quantum. A long job will fall to the lower priority queue... in this way processes travel from one level to the other…

Is easy to see an important difference between multilevel feedback queue and multilevel queues: the number of entries is different! (feedback has just 1 entry, 'standard' has 2 entries).

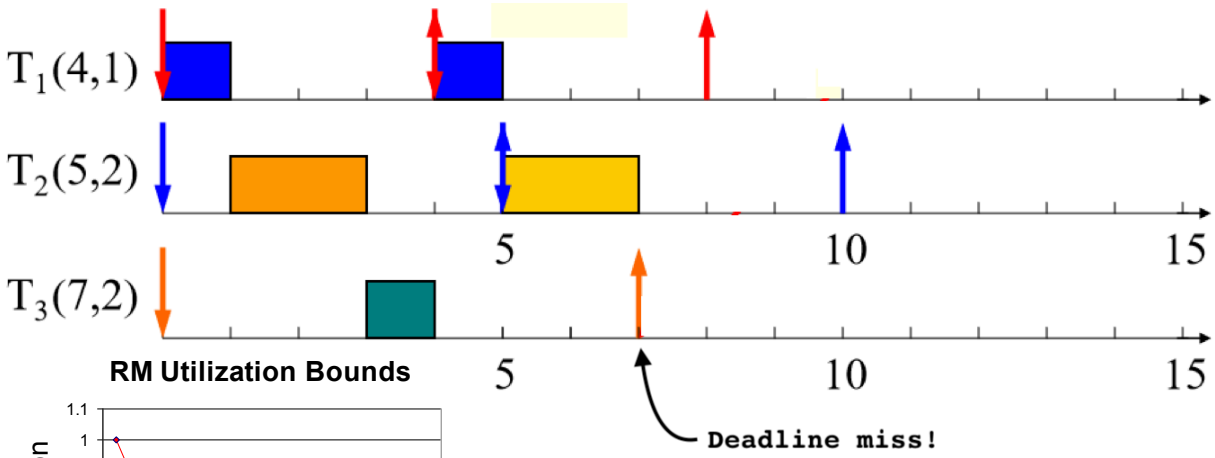
Side note: we are talking about scheduling algorithms like a black box near the CPU that gives the new process to be executed. But remember that this black box is an algorithm executed by the CPU. The CPU schedule itself the processes. Now, how much time is required to choose the next process to be executed? It depends on the algorithm used:
o(1): in RR the processes have a certain order. CPU just pick the next in the list.
o(N): in SRTF, the CPU should look at all the processes, one by one, understanding how much time they require to terminate. Then choose the shorter one. => overhead!

Rate monotonic (RM):

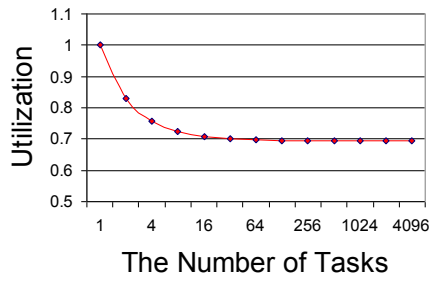
RM is an optimal static-priority scheduling. It assigns priority according to period (a task with a shorter period has higher priority).

Utilization bound: It always executes first the job with shorter period, and this can cause a deadline miss. For this reason, a real-time system is schedulable under RM if

$$\sum U_i \leq n(2^{1/n} - 1)$$

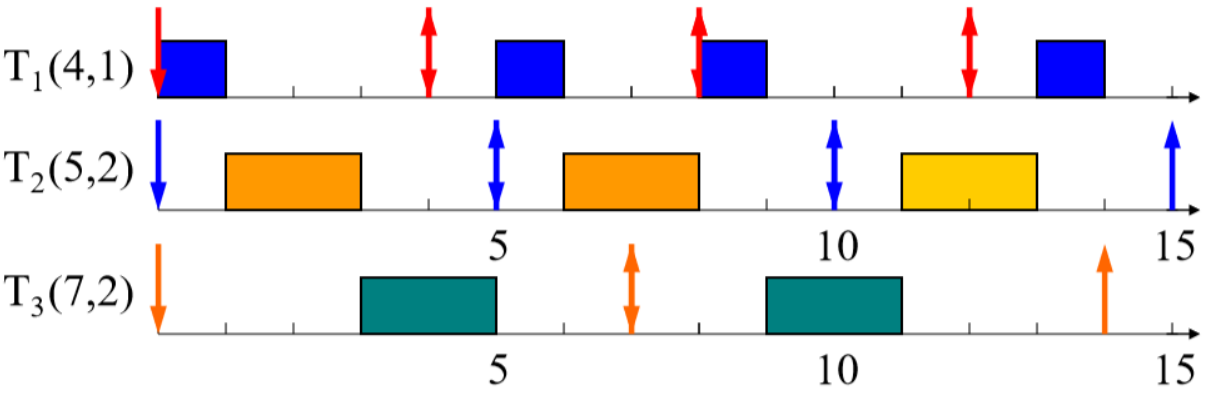


Instead of finishing T3, scheduler is executing second instance of T1 and T2 because they have an higer priority.



Earliest Deadline First (EDF):

EDF is an optimal dynamic priority scheduling. A task with a shorter deadline has a higher priority, it means it executes a job with the earliest deadline.



Utilization Bound: real-time system is schedulable under EDF if

$$\sum U_i \leq 1.$$

However during overload conditions, there may be a domino effect: even if a deadline is not met, the system will prioritize the execution of that job, causing other deadline misses.

Single Queue Multiprocessor Scheduling (SQMS):

1. Most basic design: all processes go into a single queue. CPUs pull tasks from the queue as needed.
2. Advantages: good for load balancing (CPUs pull processes on demand).
3. Disadvantages:
   - The process queue is a shared data structure: it necessitates locking, or careful lock-free design
   - SQMS does not respect cache affinity! In the worst case scenario every CPU have a lot of different processes. In numa architecture it means a communication on the shared bus, that is slow!

Multiple Queue Multiprocessor Scheduling (MQMS):

1. SQMS can be modified to preserve affinity
2. Each CPU maintains it's own queue of processes, and it means that every CPU schedule its processes independently.
3. Advantages:
   - Very little shared data, because queues are (mostly) independent
   - Respects cache affinity
4. Disadvantages:
   - MQMS is prone to load **imbalance** due to:
     a. different number of processes per CPU
     b. variable behavior across processes (short or long, I/O bound or CPU bound).
   - Must be dealt with through **process migration!**
     a. **Push migration:** a processor can ask some other processor to **take** one or more processes. It is implemented as follow: a specific task periodically checks the load on each processor and, if it finds an imbalance, evenly distributes the load by moving processes from overloaded to idle or less-busy processors

     b. **Pull migration:** a processor can ask some other processor to **give** one or more processes. It is implemented as follow: an idle processor pulls a waiting task from a busy processor

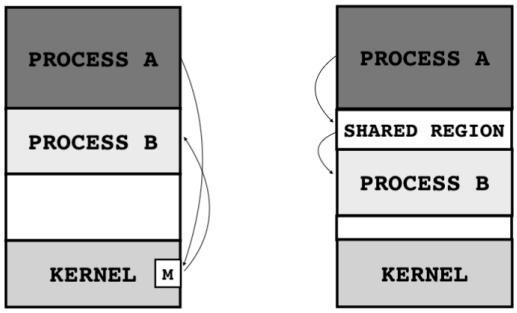
     c. Migration can cause conflicts with processor affinity (move data from a workspace to another).
     d. Often a combination of the pull and push migration approaches is implemented. Example: Linux implements both techniques: it runs its load balancing algorithm every 200 ms (push migration) or whenever the run_queue for a processor is empty (pull migration)

Interprocess communication (IPC)

A process is **independent** if it cannot be affected by the other processes executing in the system.
A process is **cooperating** if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process. Cooperation is important and bring two main advantages: **modularity** and **speed-up.**

The cooperation (or exchange of data and information) is implemented by means of interprocess communication (IPC) mechanism. Two models of IPC:

1. **Shared memory:** a region of memory that is shared by cooperating processes. They can exchange information by reading and writing data to the shared region.
2. **Message passing:** communication takes place by means of messages exchanged between the cooperating processes.



Shared Memory:

1. Shared memory allows 2 or more processes to access the same memory as if they all called malloc() and were returned pointers to the same actual memory. When one process changes the memory, all the other processes see the modification.
   - In general, one process creates or allocates the shared memory segment
   - The size and access permissions for the segment are set when it is created
   - The process then attaches the shared segment, causing it to be mapped into its current data space
   - If needed, the creating process then initializes the shared memory.
   - Once created, and if permissions permit, other processes can gain access to the shared memory segment and map it into their data space
   - For each process involved, the mapped memory appears to be no different from any other of its memory addresses
   - Each process accesses the shared memory relative to its attachment address

2. Shared memory is the fastest, but not necessarily the easiest, way for processes to communicate with another one.

3. FAST: Access to a shared memory is as fast as accessing a process's non shared memory, and it does not require a system call or entry to the kernel

4. DIFFICULT: Because the kernel does not synchronize accesses to shared memory, the programmer must provide his own synchronization: is critical that two processes don't write to the same memory location at the same time.

key_t ftok(const char *path, int id)

1. type key_t is actually a long, you can use any number you want.
2. ftok() generate a unique key, from two arguments:
   - path is the file that this process can read,
   - id is usually just set to some arbitrary char, like 'A'.
3. The ftok() function uses information about the named file (in particular its *inode number*) and the id to generate a probably-unique key for the shared memory segment.

void *shmat(int shmid, const void *shmaddr, int shmflg)

Attach a shared memory segment: shmat is used to attach the referenced shared memory segment into the calling process's data segment.
Parameters:
1. shmid is a valid shared memory identifier
2. shmaddr allows the calling process some flexibility in assigning the location of the shared memory segment
   - If a nonzero value is given, shmat uses this as the attachment address for the shared memory segment
   - If shmaddr is 0, the system will choose an available address
3. shmflg is used to specify the access permissions for the shared memory segment and to request special attachment conditions, such as a read-only segment

Return value: if successful, it returns the address of the actual attachment; if fails, it returns -1.

int shmdt (void * shmaddr)

Detach a process from a shared memory segment: usually when a process finished with a shared memory segment, the segment should be detached using shmdt().
To detach a segment, it is needed to pass the address returned by shmat().

Parameter: void ***shmaddr:** a reference to an attached memory segment (the shared memory pointer).
Return value: if successful, it returns 0, otherwise -1 (failure).

int shmctl(int shmid, int cmd, struct shmid_ds *buf)

It call returns info about a shared memory segment, and can modify it.
Parameters:
1. **shmid** corresponds to the shared memory identifier (i.e., the address returned by shmat())
2. **cmd:** perform a specific operation coded into a command. Examples:
   - to obtain information: IPC_STAT (see man pages)
   - to remove a segment: pass IPC_RMID as the second parameter and NULL as the third parameter.
     a. The segment is removed when the last process that has attached it, finally detaches it.
3. **buf** is a pointer to a structure. Depending on the command, can perform different tasks or be totally unnecessary (NULL).

NOTES:
- the process that creates a new mailbox is the initial owner of the mailbox that can receive messages through this mailbox.
- The ownership and receiving privilege may be passed to other processes through appropriate system calls. This provision could result in multiple receivers for each mailbox.

Synchronization:

Message passing may be either blocking or non-blocking

1. **Blocking** is considered **synchronous**
   - Blocking send: the sender blocks until the message is received
   - Blocking receive: the receiver blocks until a message is available

2. **Non-blocking** is considered **asynchronous**
   - Non-blocking send: the sender sends the message and continues
   - Non-blocking receive: the receiver receives a valid message or null

Possible combinations are

1. blocking send and blocking receive (rendez-vous)
2. non blocking send and blocking receive
3. non blocking send and non blocking receive.

Buffering:

Mailboxes (aka message queues) can be implemented in 3 ways:

1. **Zero capacity** (0 messages can be stored). The sender must wait for receiver (rendez-vous).
2. **Bounded capacity** (finite length of n messages can be stored inside mailbox). The sender must wait if mailbox is full.
3. **Unbounded capacity** (infinite mailbox's length). Sender never waits.

POSIX Message Queue:

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier.
A message is composed of message type and message data. It can be either private or public:

- **private:** it can be accessed only by its creating process or child processes of that creator.
- **public:** it can be accessed by any process that knows the queue's key.

int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtype, int msgflg)

A program can remove a message from the message queue by calling msgrcv().
Parameters:

1. **msqid** corresponds to the message queue identifier (the value returned by msgget())
2. **msgp** argument points to a user-defined buffer for holding the message to be retrieved (the format is the same data structure type+data adopted by the sender)
3. **msgsz** specifies the actual size of the message text
4. **msgtype** can be used by the receiver for message selection
   - = 0 : first message available in a FIFO queue
   - > 0 : first message on queue whose type equals msgtype
   - < 0 : first message on queue whose type is the lowest value less than or equal to the absolute value of msgtype
5. **msflg:** flag argument is a bit mask (see the man pages):

Return value: if successful, returns the number of bytes in the text of the message, otherwise it returns -1.

int msgctl(int msqid, int cmd, struct msqid_ds *buf)

It is used to perform control operations on a message queue.
Parameters:

1. **msqid** corresponds to the message queue identifier (i.e., the value returned by msgget())
2. **cmd:** perform a specific operation coded into a command. Examples:
   - to obtain information: IPC_STAT (see man pages)
   - to remove a message queue: IPC_RMID as the second parameter and NULL as the third parameter.
     a. The message queue (and any data still on the queue) is removed.
     b. The removal is immediate and any other process still using the message queue will get an error on its next attempted operation on the queue.
3. **buf** is a pointer to a structure. Depending on the command, can perform different tasks or be totally unnecessary.

```
// RECEIVER

#define BUF_SIZE 512

struct my_msg_struct {
      long int my_msg_type;
      char some_text[BUF_SIZE];
};

int main(){
      int running = 1;
      int msgid;
      struct my_msg_struct some_data;
      long int type_rcv = 0;
      int mykey = getuid();

      msgid = msgget((key_t) mykey, 0666 | IPC_CREAT);

      if (msgid == -1) {
            printf("msgget() failed.\n");
            exit(EXIT_FAILURE);
      }

      while(running) {
            if (msgrcv(msgid, &some_data, BUF_SIZE, type_rcv, 0) == -1){
                  printf("msgrcv() failed.\n");
                  exit(EXIT_FAILURE);
            }

            printf("You wrote: %s", some_data.some_text);

            if (strncmp(some_data.some_text,"end", 3) == 0){
                  running = 0;
            }
      }

      if (msgctl(msgid, IPC_RMID, 0) == -1) {
            printf("msgctl() failed.\n");
            exit(EXIT_FAILURE);
      }

      exit(EXIT_SUCCESS);
}
```

int open(char *path, int flags [, int mode])

It makes a request to the OS to use a file.
Parameters:

1. **path** argument specifies the file you would like to use

2. **flags** and **mode** arguments specify how you would like to use it (mode is optional).

Return value:

a. If the operating system approves your request, it will return a file descriptor to you. **This is a non-negative integer.** Any future accesses to this file needs to provide this file descriptor.

b. If it returns -1, then you have been denied access; check the value of global variable "**errno**" to determine why (or use **perror()** to print corresponding error message). Possible error cases:
   - wrong file name set (e.g. already exist).
   - open a read-only file in write mode.

The allowable option_flags as defined in #include <fcntl.h> are:

```
  #define O_RDONLY 0       /* open the file for reading only */
  #define O_WRONLY 1       /* open the file for writing only */
  #define O_RDWR 2         /* open the file for both reading and
                                                  writing*/
  #define O_APPEND 010     /* append (writes guaranteed at the end)*/
  #define O_CREAT 00400    /* open with file create
                                        (uses third open arg) */
  #define O_EXCL 02000     /* error if create and file exists */
```

Multiple values are combined using the | operator (i.e. bitwise OR).

void perror(char * s)

Print a message on stderr (usually correspond to the display). The message explain what kind of error occurred.
Parameters:

1. Pointer s: the programmer can attach a custom message (pointed by s) just before the error message (useful for debugging).
2. Different kind of errors could occur. Each error is a associated with a number; a table links the error string with the number.

Open, close, read and write system calls are really important! They are used by many higher level libraries. Since they are atomic operations, concurrent access is not a problem!

```
#include <fcntl.h>                                    // Example of read()

int main() {
      char *c;
      int fd, num;
      c = (char *) malloc(100 * sizeof(char));  //(char *) is a cast.

      fd = open("file.txt", O_RDONLY);
      if (fd < 0) {
            perror("file.txt");
            exit(1);
      }

      num = read(fd, c, 10);   // num store how may bytes were read
                               // 1 byte = 1 char

      printf("called read(%d, c, 10), which read %d bytes.\n", fd,
num);
      c[num] = '\0';          // print the string, without any
                              // random chars at the end

      printf("Read the string: %s\n", c);
      close(fd);
}
```

NB: Which is the size occupied by a pointer? **The size of a pointer is FIXED and depends on the architecture, and in particular on the memory size** (pointer is an address and should cover all memory positions)!

NB: Which is the type (int, char, ...) of a pointer? Because a pointer has a fixed size, a pointer to a char or int or long int has the same size! You are not able to understand what type of data it is pointing to ... this information is crucial, because tells the OS how many bytes should read at a time. **A cast is necessary!**

NB: the malloc function take as input the number of bytes to reserve. It returns a pointer: it has no type associated with, and a cast is necessary.

Standard Input, Output and Error:

Every process in Unix starts out with three file descriptors predefined:
- File descriptor 0 is standard input.
- File descriptor 1 is standard output.
- File descriptor 2 is standard error.

You can read from standard input, using **read(0, ...)**, and write to standard output using write**(1, ...)** or using two library calls
- printf
- scanf

off_t lseek(int fd, off_t offset, int whence)

All open files have a "file pointer" associated with them to record the current position for the next file operation.
1. When file is opened, file pointer points to the beginning of the file
2. After reading/write m bytes, the file pointer moves m bytes forward

The lseek moves the file pointer explicitly.
Parameters:
1. **whence** argument specifies from where the seek (= search) should be performed
   - SEEK_SET: from the **beginning** of the file
   - SEEK_CUR: from the **current value** of the pointer
   - SEEK_END: from the **end** of the file
2. **offset** represent the offset of the pointer (measured in bytes). Could be negative.
3. **fd** is the file descriptor

Return value: if successful, it returns the offset of the pointer (in bytes) from the beginning of the file. If there was an error moving the pointer, it returns -1.

The lseek is really convenient because allow us to move forward and backward inside the file, without reading it (a file is just a sequence of chars).

Examples:

```
• lseek(f1, 100, SEEK_SET)  // starting from the beginning of the file,
                            // the pointer now points to the 100th byte.
• lseek(f1, 0, SEEK_SET)    // return to the beginning of the file
• lseek(f1, 0, SEEK_END)    // start from the end of the file
• lseek(f1, -100, SEEK_END) // starting form the end of the file, move
                            // the pointer 100 bytes back.
• lseek(f1, 0, SEEK_CUR)    // returns the offset between the current
                            // pointer position and the beginning
                            // of the file.
```