The *operating system*:

Scegli un'alternativa:

- a. Coincides exactly with the kernel
- b. Must obey to the CPU scheduling policies as any other user process does
- c. Is stored in main memory (RAM) even after the machine is shut down
- d. Represents the interface between the physical machine (i.e., hardware) and user applications

SELF ESPLANATORY

The transition from *user* to *kernel mode* occurs when:

Scegli un'alternativa:

- a. The first instruction of a program is executed
- b. A program makes a function call
- c. The system is starting up (bootstrap)
- d. The time slice assigned to the currently executing process elapses

KERNEL MODE ALLOWS TO EXECUTE SERVICES THAT ONLY THE OS CAN EXECUTE.
UPON A SYSTEM CALL, THE MODE SWITCHES FROM USER MODE TO KERNEL MODE.
IT IS WHEN THE OPERATING SYSTEM TAKES CONTROL OF THE COMPUTER.

WHENEVER A TRAP OR INTERRUPT OCCURS, THE SYSTEM ALWAYS SWITCHES TO KERNEL MODE.

SO WHEN THE TIMER FOR THE CONTEXT SWITCH EXPIRES, IT THROWS SW INTERRUPT AND THE SYSTEM GOES IN KERNEL MODE.

The *system calls*:

Scegli un'alternativa:

- a. Must be implemented in kernel space
- b. Cause the termination of the currently executing process and the starting of a new one
- c. Are always blocking (i.e., they always suspend the process that makes them)
- d. Must be implemented in user space

BEHIND THE SCENES, THE API PROVIDES FUNCTIONS THAT INVOKE THE ACTUAL SYSTEM CALLS ON THE BEHALF OF THE USER APPLICATION.
(EVEN THO WE COULD AVOID THE API BUT IT WOULD BE MORE DIFFICULT)

REAL TIME ENVIRONMENT

RTE = SOFTWARE NEEDED TO EXECUTE SW KIND OF APPLICATION.

IT PROVIDES A SYSTEM-CALL INTERFACE, THAT INTERCEPTS FUNCTION CALLS IN THE API AND INVOKES ACTUAL SYSTEM CALLS.

THE RTE MAINTAINS A TABLE IN WHICH EACH

SYSTEM CALLS IS INDEED AND READY TO RUN.

A running process on the CPU moves to the *waiting* status when:

Scegli un'alternativa:

- a. Its time slice elapses
- b. It asks for opening a network connection (e.g., a TCP socket)
- c. It makes a function call
- d. It gets an interrupt signal from an I/O device

TO OPEN A TCP CONNECTION IS MUST SEND AN INTERRUPT SIGNAL TO USE THE NETWORK ADAPTER

A *preemptive* scheduler takes over when:

Scegli un'alternativa:

- a. A process moves from *running* to *waiting*
- b. A process moves from *running* to *ready*
- c. A process moves from *waiting* to *ready*
- d. All the previous answers are correct

CPU-SCHEDULING DECISION MAY HAPPEN WHEN:
RUNNING → WAITING : MUST EXECUTE CONTEXT SWITCH
RUNNING → READY : POSSIBLY INTERRUPT OCCURRED
WAITING → READY : AT I/O COMPLETION
AND PROCESS TERMINATION

Compute the *average waiting time* of the following processes, assuming a *round robin* CPU scheduling policy with *time slice* = 3, no I/O requests and negligible time needed for context switch:

Job	$T_{arrival}$	T_{burst}
A	0	5
B	2	8
C	7	4
D	8	3

Scegli un'alternativa:

- a. 6.75
- b. 5.85
- c. 6.5
- d. 7.15

(TIME OF COMPLETION)

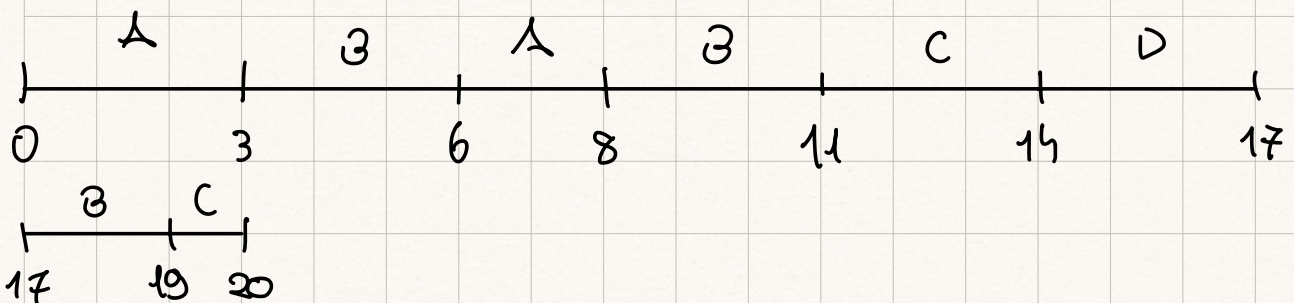
WAITING TIME = TURN AROUND TIME − BURST TIME

(TIME IN THE QUEUE DOING NOTHING)

↓

EXIT TIME − ARRIVAL TIME

ROUND ROBIN VA FA UNO PER VOLTA IN ORDINE DI ARRIVO.



TURN AROUND	WAITING TIME
A = 8	A = 8 − 5 = 3
B = 17	B = 17 − 8 = 9
C = 13	C = 13 − 4 = 9
D = 9	D = 9 − 3 = 6

$$\frac{3+9+9+6}{4} = 6,75 = \text{AVERAGE WAITING TIME}$$

In the *many-to-one* thread mapping:

Scegli un'alternativa:

- a. Several kernel threads are mapped on a single user thread
- b. A blocking call from a user thread *does not* block the other threads (of the same process)
- c. Several user threads can be distributed across more CPUs (if possible)
- d. Several user threads are mapped on a single kernel thread

SELF EXPLANATORY

To use a *lock* synchronisation primitive, we must ensure that:

Scegli un'alternativa:

- a. The lock is initially free
- b. The lock is acquired *before* entering the critical section (of code)
- c. The lock is released *after* exiting the critical section (of code)
- d. All the above conditions must be verified

DOESN'T MAKE SENSE TO ME BUT OK.
RIGUARDA SLIDE

The acquisition of a *lock*:

Scegli un'alternativa:

- a. Must be done atomically so as to avoid the CPU scheduler takes over in the middle
- b. Can only be implemented by atomic hardware instructions
- c. Requires always that the operating system disables interrupts
- d. None of the above answers is correct

SELF EXPLANATORY

Suppose a process whose size is 2,097 bytes and a free block of main memory has size 2,104 bytes. Assuming contiguous memory allocation, the most convenient choice is to:

Scegli un'alternativa:

- a. Allocate the whole block to the process, thereby wasting 7 bytes (*internal* fragmentation)
- b. Allocate only the portion of the block that is necessary to the process and add to the list of free blocks the remaining 7 bytes (*external* fragmentation)
- c. Wait for a smaller block that best fits the process
- d. Wait for a larger block whose size is multiple of that of the process

I GUESS IT'S BECAUSE IF WE MAKE A BLOCK OUT OF THE 7 REMAINING BYTES WE ADD OVERHEAD TO THE USE OF FREE BLOCKS

Suppose you have a memory M whose size is 8 KiB, namely 8,192 bytes. If the smallest addressable unit corresponds to a *word size* equals to a single byte and M is divided into pages, each one with size S = 128 bytes, what is the dimension (i.e., the *number of entries*) of the corresponding *page table* T?

Scegli un'alternativa:

- a. There is no enough information to answer this question
- b. 13
- c. 64
- d. 7

8192 / 128 = 64

The *virtual memory* allows to:

Scegli un'alternativa:

- a. Increment the efficiency of I/O operations
- b. Lower the degree of multiprogramming
- c. Execute a process directly from secondary storage (e.g., disk)
- d. Keep in main memory only some of the pages of the whole virtual address space of a process

SELF EXPLANATORY

Suppose the access time to physical memory is t_{MA} = 50 nsec. and the time for handling a page fault is t_{FAULT} = 15 msec. Assuming the probability of page fault is p = 0.0002, what is the total time to access physical memory?

Scegli un'alternativa:

- a. ~3.05 microsec
- b. ~305 nsec
- c. ~30.5 microsec
- d. ~30.5 nsec

EFFECTIVE ACCESS TIME:

$$\text{HIT RATE} \cdot \underset{\text{TIME}}{\text{LOOKUP}} + \text{MISS RATE} \cdot \underset{\text{COST}}{\text{PAGE FAULT}}$$

ES.

$$0.80 \cdot 10 + 0.20 \cdot 20 = 12 \text{ NANOSECONDS}$$

HERE:

$$0,9998 \cdot 50 + 0.0002 \cdot 15\,000\,000 = 3050 \text{ NANOSECONDS}$$

WHICH IS EQUAL TO 3.05 MICROSECONDS

10^{-9} 10^{-6} 10^{-3}

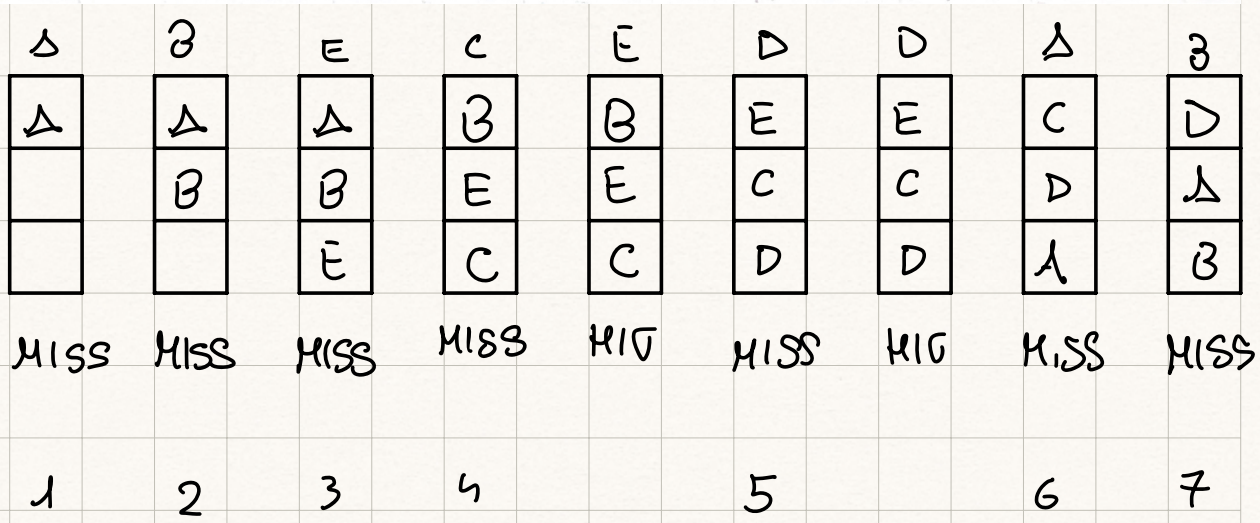
NANOSECONDS → MICROSECONDS → MILLISECONDS

1s = 1 000 000 000 NS 1s = 1 000 000 MCS 1s = 1000 MS

Consider a main memory made of 3 (physical) frames and a process spawning 5 (virtual) pages: A, B, C, D, E. Calculate the number of page faults that occur upon the following sequence of page requests issued by the process: A, B, E, C, E, D, D, A, B. We assume that, initially, *no* page is loaded in main memory and the system uses a FIFO page replacement algorithm.

Scegli un'alternativa:

- a. 4
- b. 8
- c. 7
- d. 6



FIFO = FIRST IN FIRST OUT

THIS ALGORITHM ASSOCIATES EACH ENTRY WITH THE TIME THEY ARE BROUGHT INTO MEMORY.

EVERY TIME THERE IS A PAGE FAULT, THE OLDEST PAGE IS CHOSEN TO BE KILLED.

In a magnetic (rotational) disk, the *seek time*:

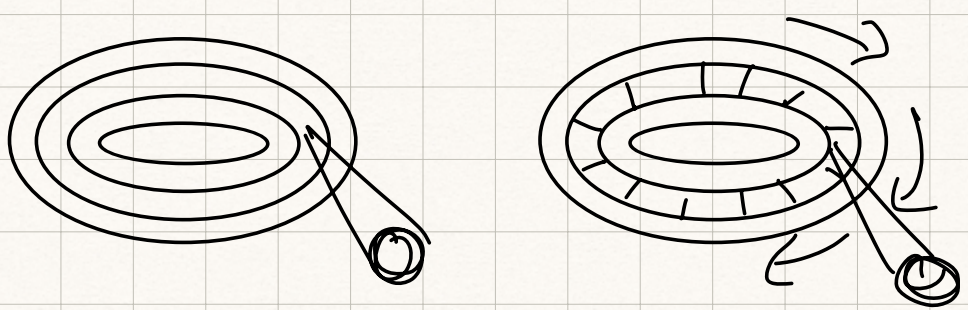
Scegli un'alternativa:

- a. Is negligible with respect to the whole time needed for transferring data (to main memory)
- b. Is the time required to the disk for placing their head over a specific cylinder
- c. Includes the time for transferring data to main memory
- d. Is the time required to the disk for placing their head over a specific sector

POSITIONING TIME CONSISTS OF TWO PARTS:

SEEK TIME
TIME FOR ARM TO GET THE SPECIFIC CYLINDER

ROTATIONAL LATENCY
TIME FOR THE DISK TO ROTATE AND GET TO SPECIFIC SECTOR



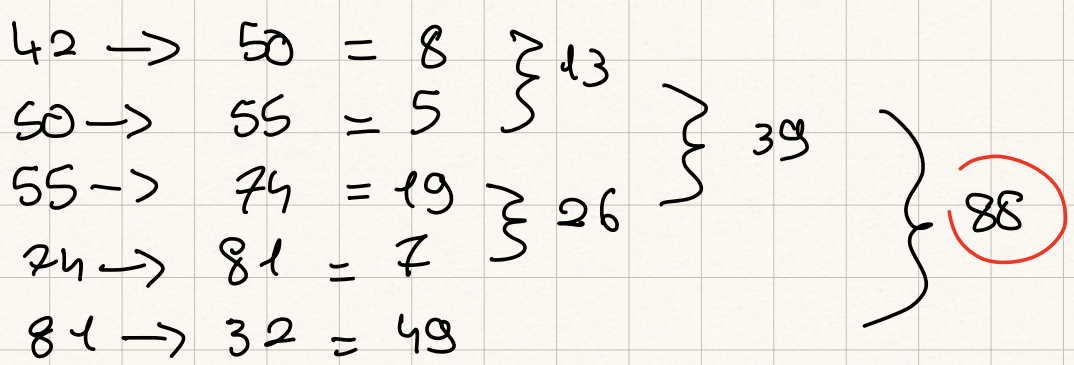
(GET CYLINDER) (GET SECTOR)

POSITIONING TIME = SEEK TIME + ROTATIONAL LATENCY

Let us consider a magnetic (rotational) disk composed of 128 cylinders/tracks, numbered from 0 to 127 (where 0 is the index of the outermost cylinder/track), whose head is initially over cylinder 42. Calculate the total number of cylinders/tracks traversed by the head, assuming the following sequence of requests (74, 50, 32, 55, 81) is managed by an SSTF (Shortest Seek Time First) disk scheduling algorithm, without considering the rotational delay.

Scegli un'alternativa:

- a. 123
- b. 49
- c. 88 ✗
- d. 86



I GUESS THAT SSTF ALWAYS GOES TO THE CYLINDER THAT COSTS LESS NEXT.

Contiguous allocation of file on disk:

Scegli un'alternativa:

- a. Is optimal both for direct (i.e., random) and sequential access
- b. Suffers from the problem of fragmentation
- c. Needs to keep track of the free blocks using a dedicated data structure
- d. All the above answers are correct ✗

a) BECAUSE OF LOCALITY AND POSSIBLE CACHE
b) JUST TRUE
c) JUST TRUE

A possible example of an application that needs sequential access to a file is:

Scegli un'alternativa:

- a. A compiler
- b. A search engine within a database system
- c. A search engine within a contact list
- d. None of the above answers are correct

I GUESS IT'S BECAUSE IT NEEDS THE ADDRESS SPACE TO BE CONTIGUOUS.

A CPU takes 5 clock cycles to fetch-decode-execute an instruction (i.e., Cycles Per Instruction or CPI = 5). If the CPU clock frequency is 5 MHz, how many instructions it is able to run in one second? (Notice that 1 MHz = 1*10^6 clock cycles per second)

Scegli un'alternativa:

- a. 1*10^6
- b. 25*10^6
- c. 25*10^3
- d. 1*10^3

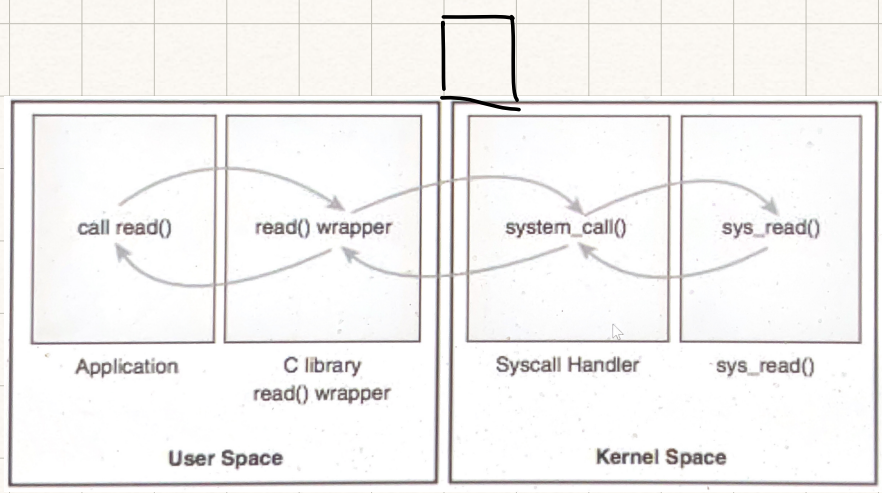
IF THE CPU IS ABLE TO DO $5 \cdot 10^6$ CLOCK CYCLES PER SECOND, THEN IF AN INSTRUCTION TAKES

5 CLOCK CYCLES, $\frac{5 \cdot 10^6}{5} = 1 \cdot 10^6$ INSTRUCTIONS

The code of the generic *system call handler*:

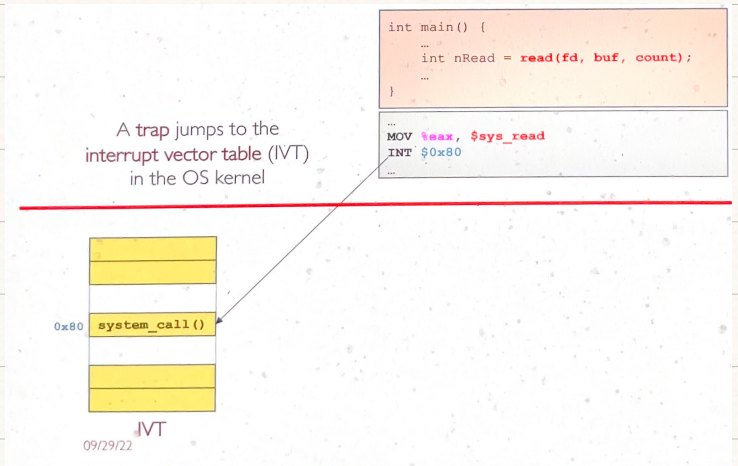
Scegli un'alternativa:

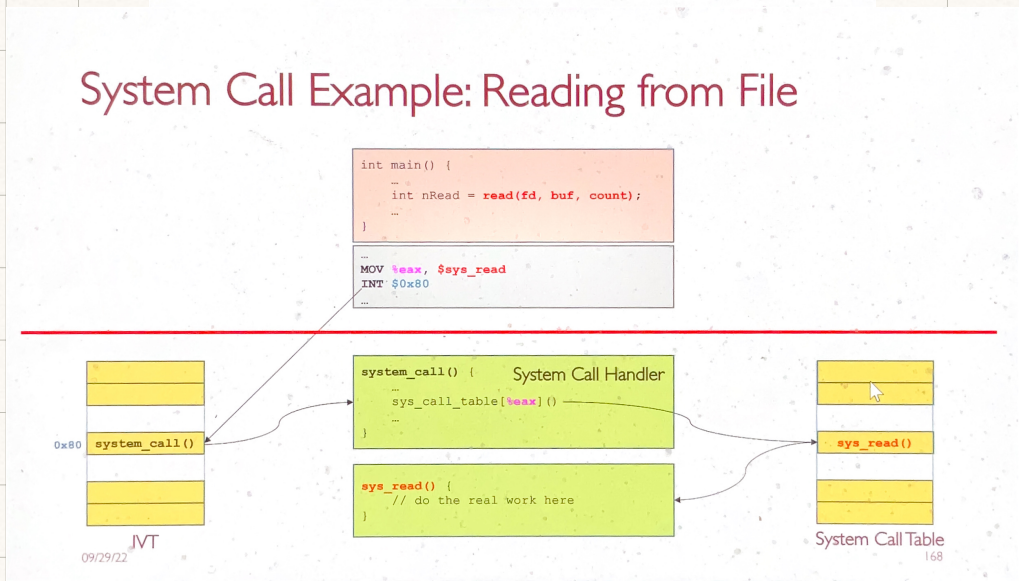
- a. Is executed in *user space*
- b. Is found by a lookup in the *interrupt vector table* (IVT)
- c. Is invoked once the time quantum expires
- d. Is invoked by the CPU scheduler



THE TRAP CAUSED BY SYSTEM CALL INVOCATION MAKES THE CPU SWITCH TO KERNEL MODE.

THE SYSTEM CALL HANDLER FINDS AND JUMPS TO THE CORRECT ROUTINE FOR THAT TRAP.





A process running on the CPU switches to the *waiting* status when:

Scegli un'alternativa:

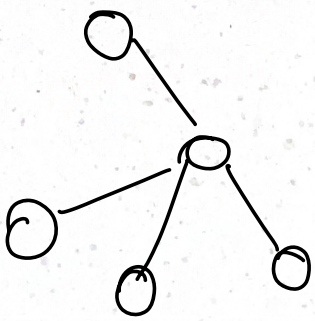
- ✗ a. It asks for input data from the user (e.g., via the keyboard)
- b. It makes a function call
- c. It gets an interrupt signal from an I/O device
- d. Its time quantum expires

THE OTHER ONES DON'T MAKE SENSE AND KEYBOARD IS I/O I GUESS.

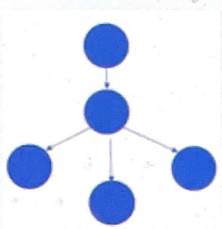
Considering the following code snippet, indicate the corresponding tree of processes generated (**Note:** by convention, assume that each level of the tree is generated from left to right):

```
int pid = fork();

if(pid == 0) {

    pid = fork();
    if (pid == 0) {
        _
    }
    else {
        pid = fork();
        if (pid == 0) {
            _
        }
        else {
            pid = fork();
                if (pid == 0) {
                    _
                }
            }
        }
    }
}
```

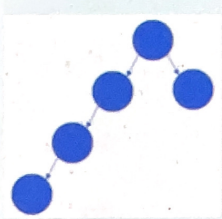
FORK() RETURNS 0 IN THE CHILD

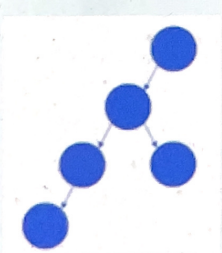


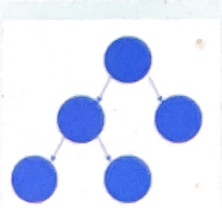
Scegli un'alternativa:

a. 

SPAWNS 1 CHILD FOR EACH FORK

b. 

c. 

d. 

CPU-bound processes that do **not** execute any I/O request:

Scegli un'alternativa:

- a. Could never *voluntarily* release the CPU
- b. Are typically "short" processes (i.e., they terminate very quickly)
- c. Have a *high* priority
- d. Have a *low* priority

I GUESS IT'S BECAUSE PROCESSES CAN ONLY BE SWAPPED VOLUNTARILY UPON AN I/O REQUEST, OR ELSE THEY GET SWAPPED BY THE TIMER OR BECAUSE THEY TERMINATED.

The term *concurrency* is used when:

Scegli un'alternativa:

- a. Multi-threaded processes are executed on *single-core* CPUs
- b. Multi-threaded processes are executed on *multi-core* CPUs
- c. Single-threaded processes are executed on *single-core* CPUs
- d. Single-threaded processes are executed on *multi-core* CPUs

CONCURRENCY = MULTIPLE TASKS THAT RUN IN OVERLAPPING TIME PERIODS.

IN MULTICORE, TASKS DON'T OVERLAP BECAUSE THEY ARE LITERALLY IN OTHER CORES.

Calling the method `wait()` on a semaphore whose value is 2:

Scegli un'alternativa:

- a. Decrements the value of the semaphore to 1 and lets the process/thread that executed the call continue running (conditioned on the CPU scheduling policies)
- b. Decrements the value of the semaphore to 1 and blocks the process/thread that executed the call
- c. Increments the value of the semaphore to 3 and lets the process/thread that executed the call continue running (conditioned on the CPU scheduling policies)
- d. Leaves the value of the semaphore to 2 and lets the process/thread that executed the call continue running (conditioned on the CPU scheduling policies)

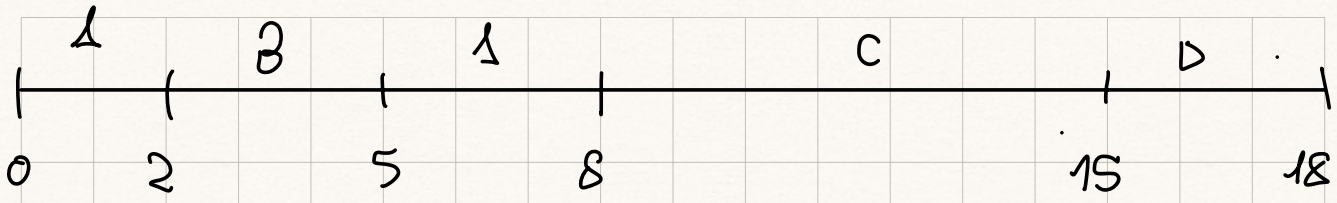
IF THE VALUE IS NOT ZERO THIS IS THE BEHAVIOUR

Compute the *average waiting time* of the following processes under the assumption of a *First Come First Served* (FCFS) scheduling policy. Moreover, at t=2, process A issues an I/O request that will be completed after 4 time units, i.e., at t=6. The time that it takes for each context switch is assumed to be negligible (i.e., approximately 0), and therefore must not be included in the computation of the average waiting time:

Job	$T_{arrival}$	T_{burst}
A	0	5
B	1	3
C	3	7
D	6	3

Scegli un'alternativa:

- a. 4.5
- b. 5.5
- c. 7.5
- d. 6.5



A = 3
B = 1
C = 5
D = 9

} 18

18/4 = 4,5 = AVERAGE WAITING TIME

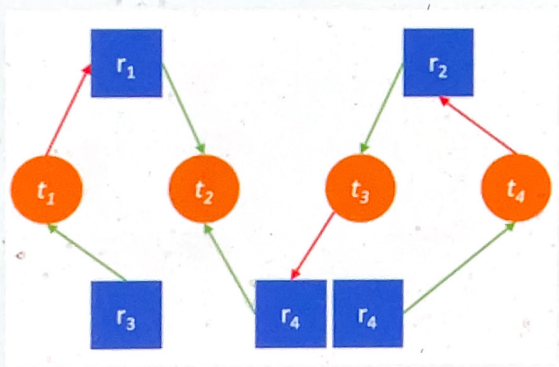
The communication between *threads* of the same process with respect to that amongst different processes:

Scegli un'alternativa:

- a. Is faster because threads (of the same process) share the same virtual address space
- b. Is slower because threads (of the same process) are managed by high-level libraries
- c. Is faster because threads (of the same process) do not execute context switch
- d. Is roughly the same and no one outperforms the other

FURTHER CHECKS WOULD BE NEEDED IF IS WASN'T IN THE SAME VIRTUAL ADDRESS SPACE.

The following *Resource Allocation Graph* (RAG) indicates a system that:



Scegli un'alternativa:

- a. Depends on the choices made by the CPU scheduler
- b. Is presenting a deadlock
- c. Is **not** presenting any deadlock
- d. It is impossible to answer

IF THERE IS NO CYCLE, THERE IS NO DEADLOCK.
GREEN ARROWS ARE ALLOCATIONS, RED ARROWS ARE REQUESTS
t_is ARE THREADS, r_is ARE RESOURCE TYPES.

A GREEN ARROW MEANS THAT r_i HAS ALLOCATED ITSELF TO SOME THREAD, WHILE RED ARROWS MEAN THAT THREADS ARE WAITING TO ACCESS THAT RESOURCE, THAT MAY BE ALLOCATED TO SOMEONE ELSE.

NOW, IF THERE IS A CYCLE, THEN, IT MAY STILL HAVE NO DEADLOCK, IT DEPENDS ON THE TOPOLOGY.

A compiler generates the virtual (i.e., logical) address 576 to access a certain physical memory location. Assuming that address binding is performed via *static relocation* with base physical address b=24, what will be the final physical address referenced?

Scegli un'alternativa:

- a. 576
- b. 552
- c. There is not enough data to answer this question
- d. 600

ADDRESS BINDING CAN BE PERFORMED BY A COMPILER THROUGH THOSE MODALITIES.

COMPILE TIME: THE COMPILER ALREADY KNOWS THE PHYSICAL LOCATION OF THE PROCESS, SO LOGICAL ADDRESS == PHYSICAL ADDRESS.
(ABSOLUTE CODE)

LOAD TIME: COMPILER GENERATES STATICALLY RELOCATABLE CODE.
(STATIC RELOCATION)
IT MEANS THAT ALL THE ADDRESSES CAN BE OFFSETTED, WITH THE SAME VALUE.
SO THE WHOLE.

EXECUTION TIME: PROGRAM CAN BE MOVED AROUND IN MAIN MEMORY DURING EXECUTION.
(DYNAMIC RELOCATION OR VIRTUAL ADDRESSES)

Suppose a process P requires a block of free memory of 99 KiB in order to be **contiguously** allocated in main memory. The list of free blocks contains the following items: A, B, C, D, whose size are respectively: 102 KiB, 99 KiB, 256 KiB, and 128 KiB. Which free block will be allocated for P, assuming a *Worst-Fit* allocation policy?

Scegli un'alternativa:

- a. D
- b. A
- c. B
- d. C ✗

WORST FIT → BIGGEST BLOCK

BEST FIT → SMALLEST BLOCK THAT FITS

FIRST FIT → FIRST BLOCK THAT FITS

The size (i.e., the number of entries) of the *page table*:

Scegli un'alternativa:

- a. Is *directly* proportional to the (fixed) page size
- b. Adapts to the memory requests made by each process
- c. Depends on the (fixed) page size ✗
- d. Changes dynamically from process to process

IF THE SPACE FOR THE PAGE TABLE IS FIXED, THEN THE BIGGER THE PAGES ARE, THE LESS THEY SECOND.

OF COURSE.

IT IS FIXED BECAUSE THE PHYSICS SPACE IS ALWAYS THE SAME.

Consider a system that uses 21-bit long virtual addresses, 16-bit long physical addresses, and memory paging where each page has size 2 KiB = 2,048 bytes. What is the maximum size of physical memory supported by the system?

Scegli un'alternativa:

- a. 32 KiB
- b. 64 KiB ✗
- c. 2 MiB
- d. There is no limit to the maximum size of physical memory supported

WHY?

Consider a memory M whose size is 4 KiB, namely 4,096 bytes. Assuming the *word size* is 2 bytes (i.e., the smallest addressable unit is 2 bytes) and M uses paging where each page has size S = 128 bytes, how many bits are necessary for specifying the page number (p) and the offset (within each page), respectively?

Scegli un'alternativa:

- a. p=7; offset=5
- b. p=5; offset=6 ✗
- c. p=6; offset=5
- d. p=5; offset=7

$2^2 \rightarrow 4$ $2^3 \rightarrow 8$ $2^4 \rightarrow 16$ $2^5 \rightarrow 32$

$2^6 \rightarrow 64$

128/2 = 64 WORD PER PAGE

SO OFFSET MUST COVER 64 NUMBERS, $2^6 = 64$ SO 6 BITS ARE ENOUGH.

WE ALREADY HAVE THE ANSWER, 32

4096 / 128 = 32 PAGES IN TOTAL,

TO COVER 32 NUMBERS WE NEED 5 BITES,

Consider the memory access time is t_{MA} = 60 nsec. and the time for handling a page fault is t_{FAULT} = 5 msec. What will be the (upper-bound) value of the probability of page fault (p) if we want to guarantee that the expected memory access time is *at most* 20% slower than t_{MA}? (**Note:** 1 msec = 10^3 microsec = 10^6 nsec)

Scegli un'alternativa:

- a. ~0,0000024%
- b. There is not enough data to answer this question
- c. ~0,000024%
- ✗ d. ~0,00024%

$60 \cdot 1.20 = 60 \cdot (1-x) + 5000000 \cdot x$

$60 \cdot 1.20 = 60 - 60x + 5000000x$

$72 = 60 - 60x + 5000000x$

$12 = 4999940x$

$\frac{12}{} = x = 0.000002$

4999940

RESULT IS IN PERCENTAGE (%), SO WE NEED TO DO x100.

0,00024 %