



reSilient coMputer archItectures
and LIfe Sciences



Politecnico
di Torino

Department of Control and
Computer Engineering



PROCESS MANAGEMENT

STEFANO DI CARLO

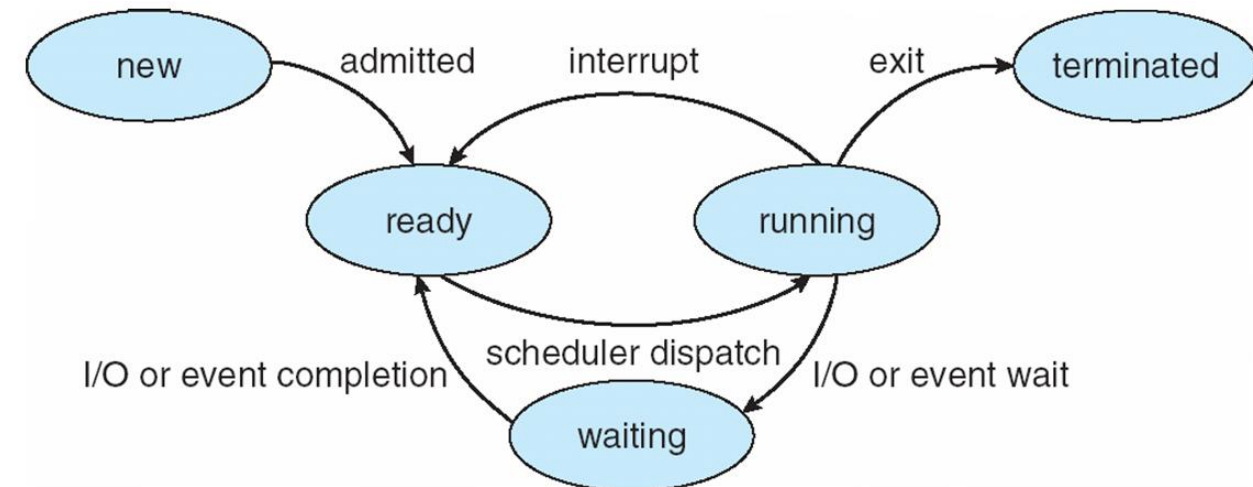
PROCESS DEFINITION

- ▶ **Process** – a program in execution
 - ▶ process execution must progress in sequential fashion.
 - ▶ No parallel execution of instructions of a single process
- ▶ Multiple parts
 - ▶ The program code, also called **text section**
 - ▶ Current activity including **program counter**, processor registers
 - ▶ **Stack** containing temporary data
 - ▶ Function parameters, return addresses, local variables
 - ▶ **Data section** containing global variables
 - ▶ **Heap** containing memory dynamically allocated during run time

PROCESS STATE

- ▶ As a process executes, it changes state
 - ▶ **New**: The process is being created
 - ▶ **Running**: Instructions are being executed
 - ▶ **Waiting**: The process is waiting for some event to occur
 - ▶ **Ready**: The process is waiting to be assigned to a processor
 - ▶ **Terminated**: The process has finished execution

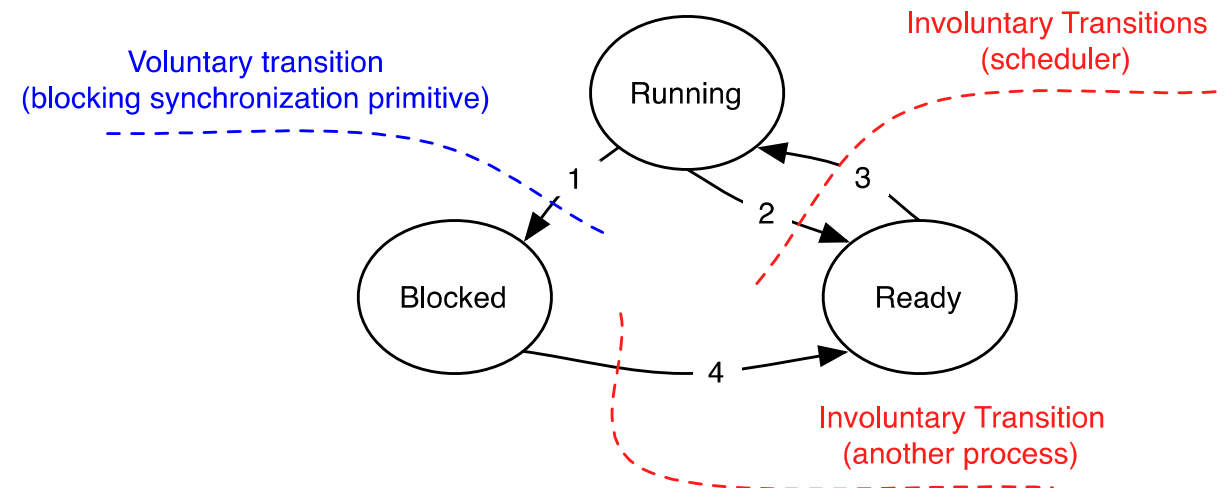
- ▶ Process State Diagram (PSD)



PSD TRANSITIONS

- ▶ Given N processes, and one processor, at any given time:
 - ▶ 1 process is in the running state
 - ▶ M processes can be blocked waiting for a resource to become available to resume the execution
 - ▶ $N-M-1$ are ready to be executed waiting to access the processor

- ▶ Transition 1 occurs when a process discovers that it cannot continue
 - ▶ For example it needs to use a portion of shared memory a now-ready process reserved for its own use

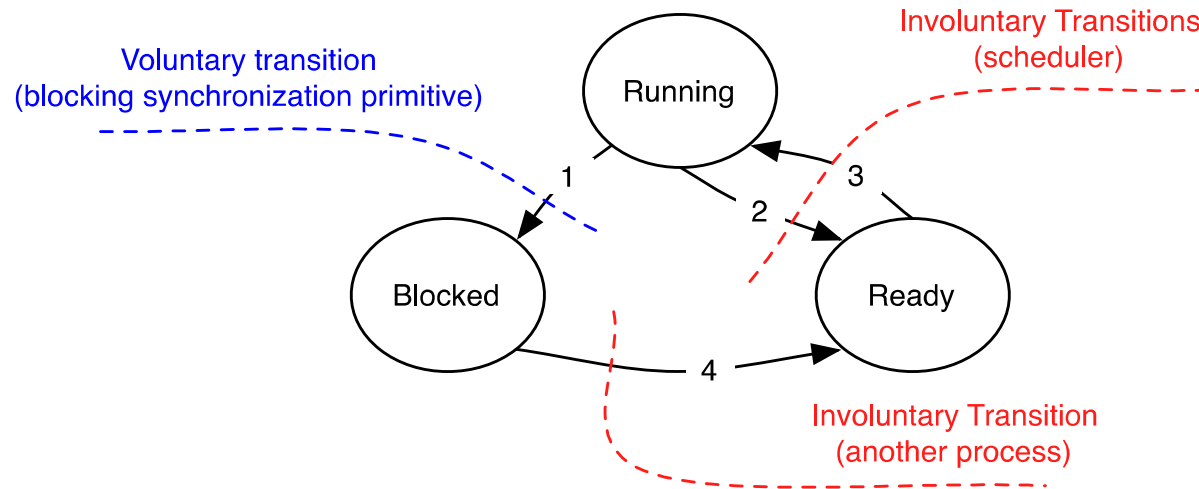


PSD TRANSITIONS

- ▶ Transition 4 occurs when the event the process was waiting for occurs
 - ▶ For example, the running process sets free the shared memory it previously locked, which the blocked process was waiting for

- ▶ Transitions 2 and 3 are caused by the operating system

- ▶ Preemptive scheduler: process is moved from running to ready after a certain time quantum (time slice) is expired
- ▶ Cooperative scheduler: process voluntary moves from running to ready



PROCESS CONTROL BLOCK (PCB)

- ▶ The CPU Manager describes each process using the PCB containing:

- ▶ Process state
- ▶ Program counter
- ▶ CPU registers
- ▶ CPU scheduling information
- ▶ Memory-management information
- ▶ Accounting information
- ▶ I/O status information

- ▶ Example of PCB

process state
process number
program counter
registers
memory limits
list of open files
...

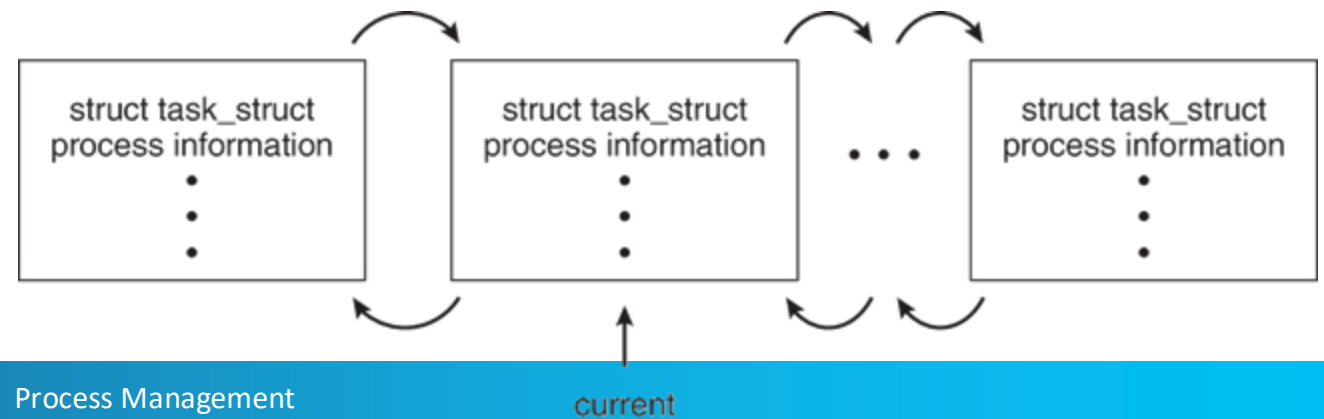
- ▶ One PCB is maintained for each process

process state	process state	process state	process state
process number	process number	process number	process number
program counter	program counter	program counter	program counter
registers	registers	registers	registers
memory limits	memory limits	memory limits	memory limits
list of open files	list of open files	list of open files	list of open files
...

PROCESS REPRESENTATION IN LINUX

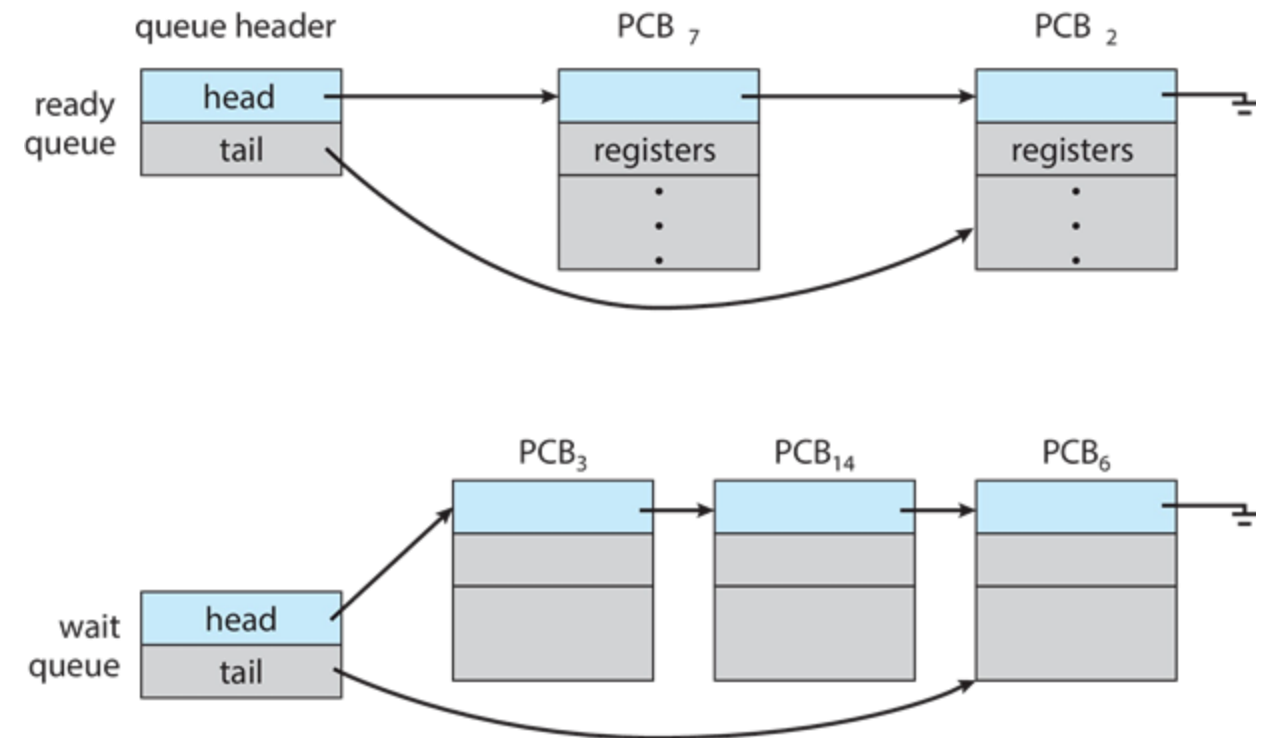
Represented by the C structure

```
struct task_struct {
    pid_t pid;          /* process identifier */
    long state;         /* state of the process */
    unsigned int time_slice /* scheduling
information */
    struct task_struct *parent; /* this
process's parent */
    struct list_head children; /* this
process's children */
    struct files_struct *files; /* list of open
files */
    struct mm_struct *mm;    /* address space
of this process */
}
```

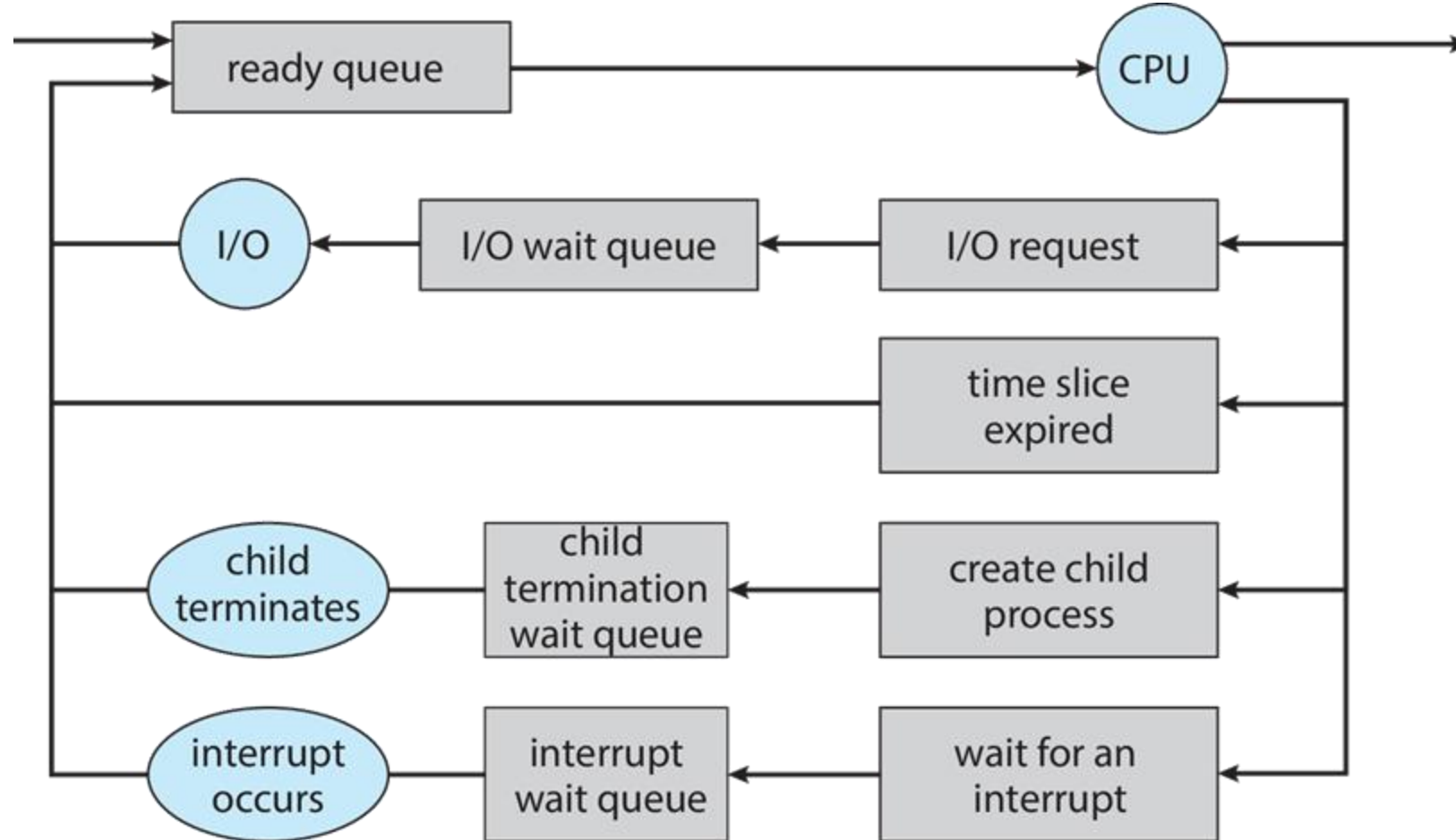


THE CPU SCHEDULER

- ▶ It is the part of the CPU manager that implements the process state transitions and decides which process must run
- ▶ The goal is to maximize CPU use
- ▶ Maintains scheduling queues of processes
 - ▶ Ready queue – set of all processes residing in main memory, ready and waiting to execute
 - ▶ Wait queues – set of processes waiting for an event (i.e., I/O)
 - ▶ Processes migrate among the various queues

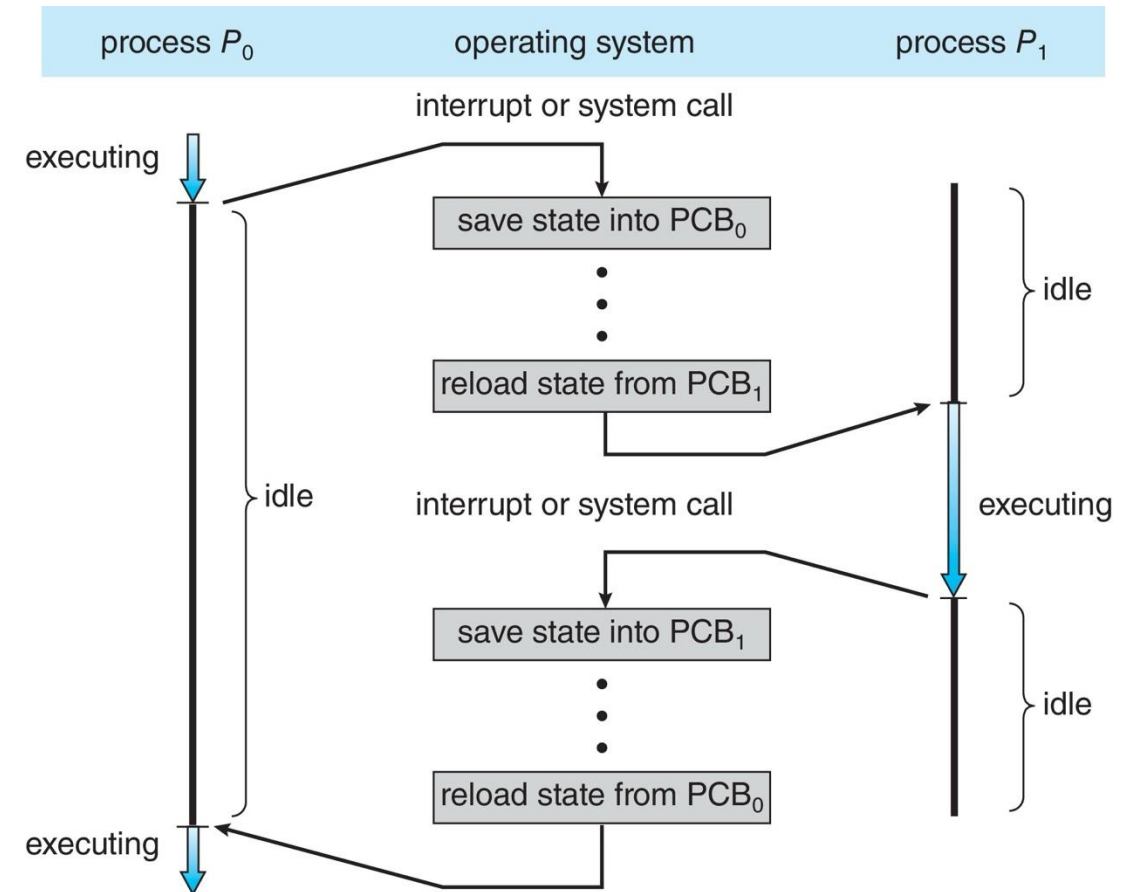


REPRESENTATION OF PROCESS SCHEDULING



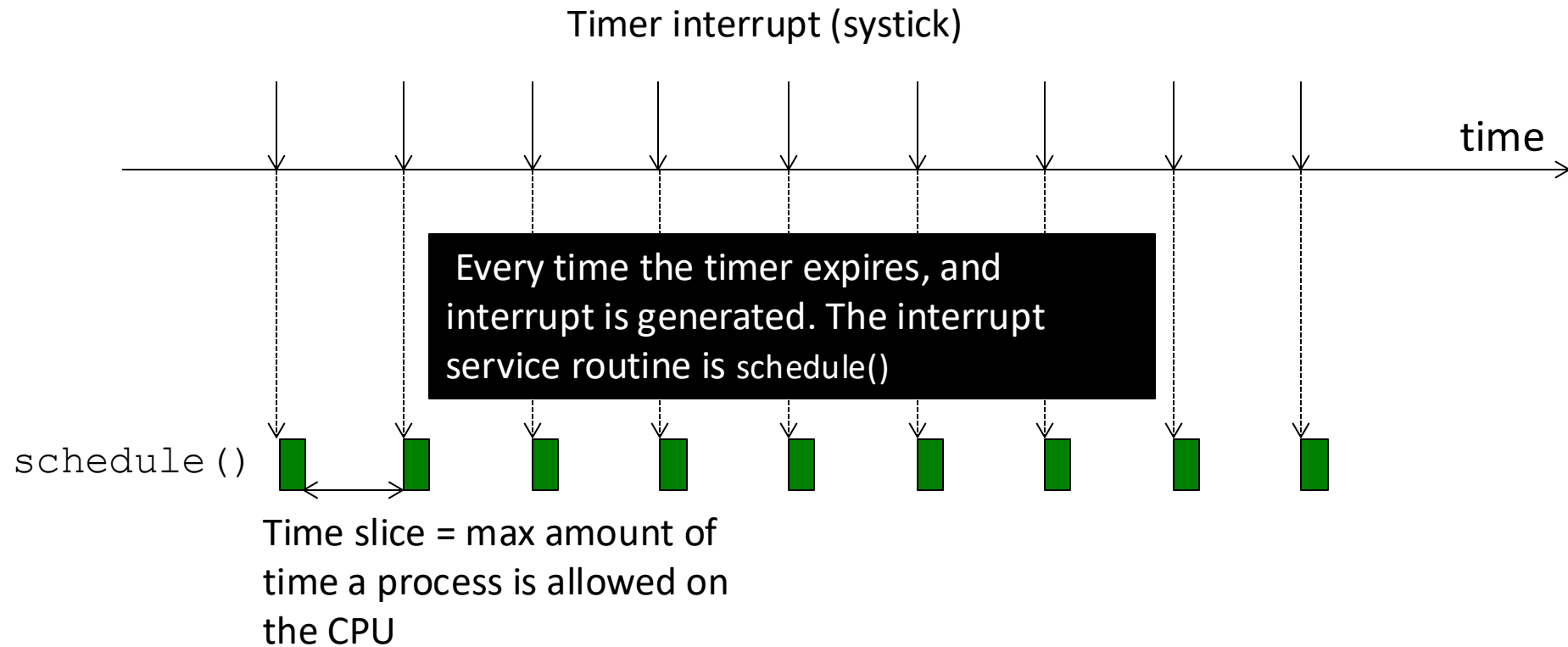
CONTEXT SWITCH

- ▶ 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**
- ▶ **Context** of a process represented in the PCB
- ▶ Context-switch time is pure overhead; the system does no useful work while switching
 - ▶ The more complex the OS and the PCB → the longer the context switch
- ▶ Time dependent on hardware support
 - ▶ Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once



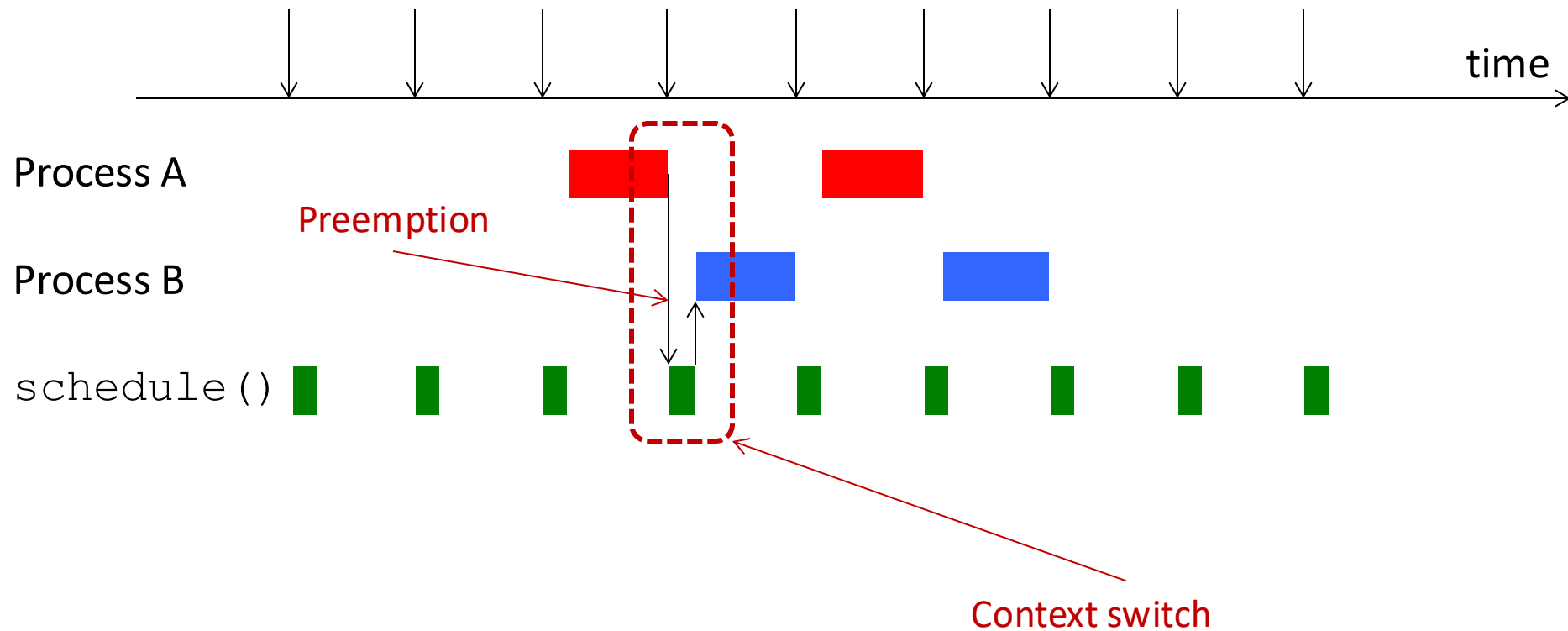
SCHEDULER ACTIVATION

- It is called periodically (via timer interrupt) or in response to events



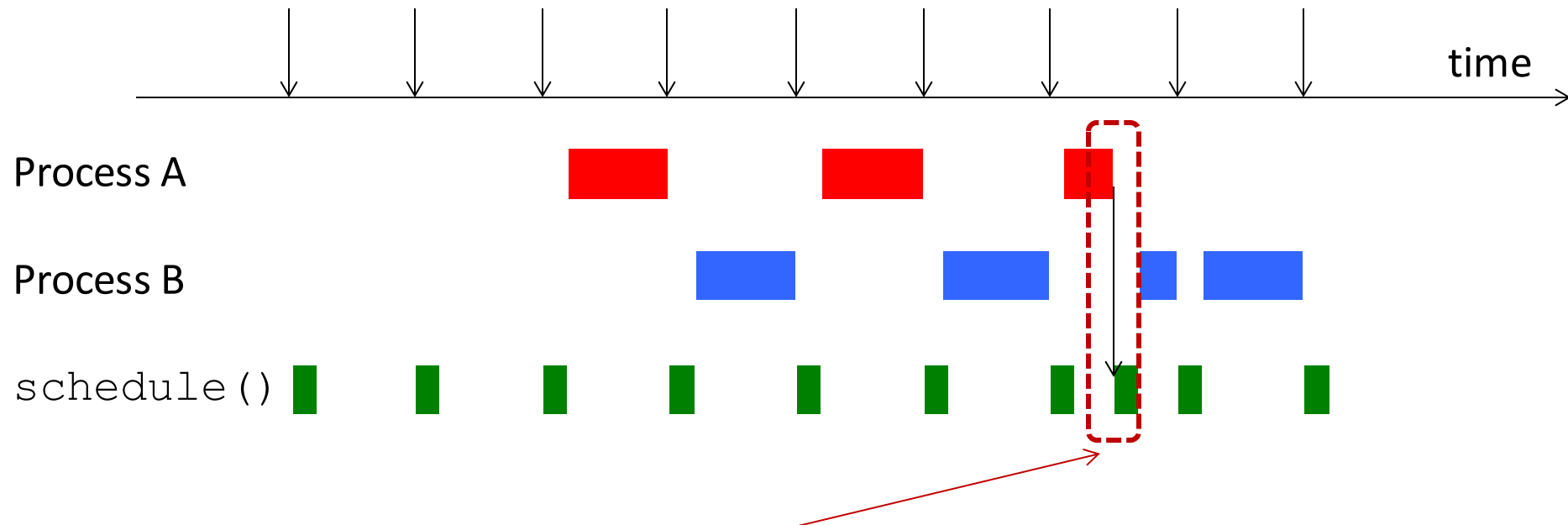
PREEMPTION

- ▶ Operation performed to evict a running process from the CPU
- ▶ Example: upon reaching the end of the time slice, process A is moved from running to ready, while process B is moved from ready to running



BLOCKING

- ▶ Preemption may take place as a result of an operation the running task performs
- ▶ Example: Process A is moved to blocked state, while process B is set running



Process A executed an instruction the blocks it

WHAT DOES A PROCESS (OR TASK) LOOK LIKE?

- ▶ Basic task
 - ▶ Sequence of statements executed once for each instance of the task (e.g., `do_instance()`)
 - ▶ Instance of the task = execution of the function `x()`
 - ▶ The task starts with the first instruction of function `x()`
 - ▶ The task terminates after the last statement of function `x()`
 - ▶ An initialization function `x_init()` is executed once for setting up the memory used by task `x`
 - ▶ If task `x` needs to keep in memory data to be used by different instances, global variables shall be used (e.g., `persistent_data`)

```
int persistent_data;
```

```
Task x()
```

```
{  
    do_instance();  
}
```

```
x_init()
```

```
{  
    initialization();  
    ...  
}
```

WHAT DOES A PROCESS (OR TASK) LOOK LIKE?

- ▶ Extended task
 - ▶ It is a function that starts once, and never ends
 - ▶ Data can be local variables
 - ▶ Initialization operations are performed once, before starting the end-less loop
 - ▶ The end-less loop implements the operation of the task (e.g., `do_instance()`)
 - ▶ The end-less loop typically contains a statement to block the task until it is needed (e.g., `WaitEvent()`)
 - ▶ A certain amount of time is elapsed
 - ▶ A resource becomes available
 - ▶ ...
 - ▶ When the task is blocked others may run

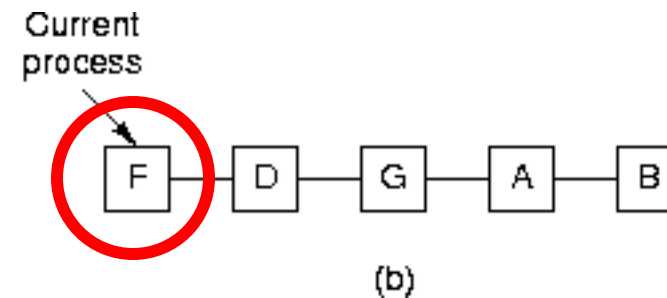
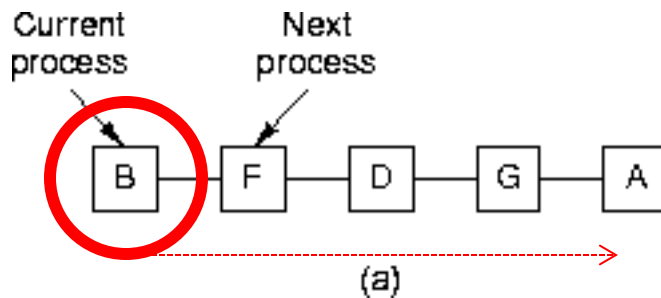
```
Task x()
{
    int local;

    initialization();

    for (;;) /* End-less loop */
    {
        WaitEvent();
        do_instance();
    }
}
```

SCHEDULING ALGORITHMS

- ▶ It is the criteria to pick up a process among those ready to make it running
- ▶ Simplest form of scheduling: Round Robin
 - ▶ Processes are inserted into a FIFO queue {B, F, D, G, A}
 - ▶ The top of the queue is executed (see figure a)
 - ▶ When exiting from running state it is queued to the last position of the queue (see figure b)



ROUND ROBIN SCHEDULER

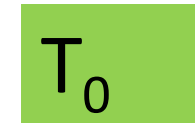
► Possible implementation

```
Schedule(ReadyList R, RunningTask T)
{
    T->TCB.state = READY;
    save_context(T->TCB);

    append_to_list( T, R );

    Q = top_of_list( R );
    restore_context(Q->TCB);
    Q->TCB.state = RUNNING;
}
```

Running
Task



List of Ready
Tasks R



Top of the list



ROUND ROBIN SCHEDULER

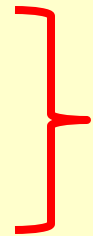
► Possible implementation

```

Schedule(ReadyList R, RunningTask T)
{
    T->TCB.state = READY;
    save_context(T->TCB);

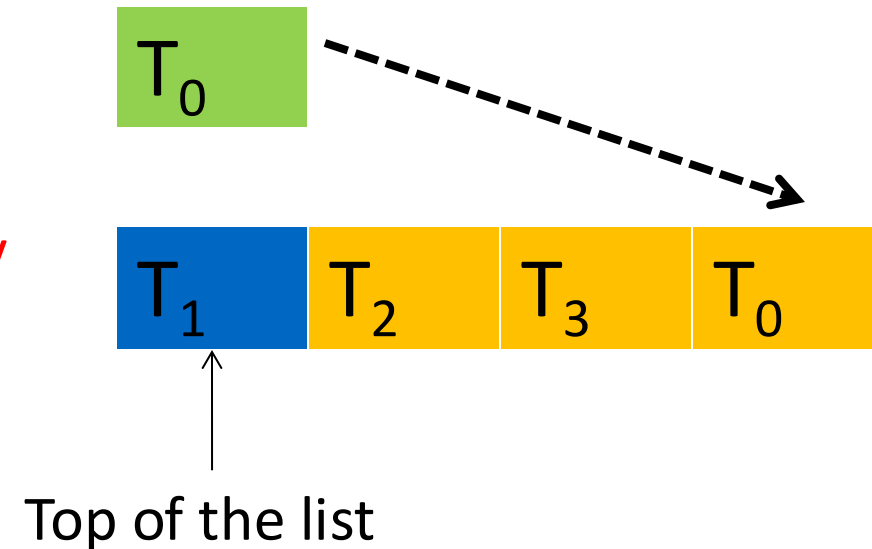
    append_to_list( T, R );

    Q = top_of_list( R );
    restore_context(Q->TCB);
    Q->TCB.state = RUNNING;
}
    
```



Running
Task

List of Ready
Tasks R



ROUND ROBIN SCHEDULER

► Possible implementation

```

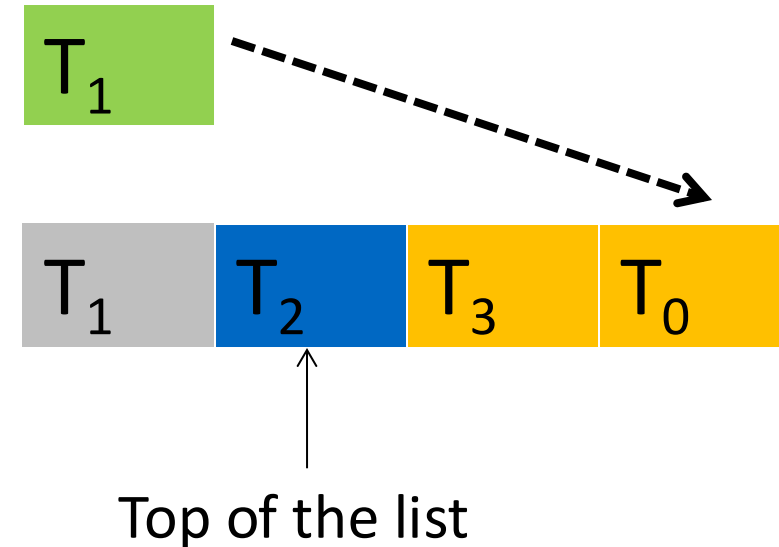
Schedule(ReadyList R, RunningTask T)
{
    T->TCB.state = READY;
    save_context(T->TCB);

    append_to_list( T, R );

    Q = top_of_list( R );
    restore_context(Q->TCB);
    Q->TCB.state = RUNNING;
}
    
```

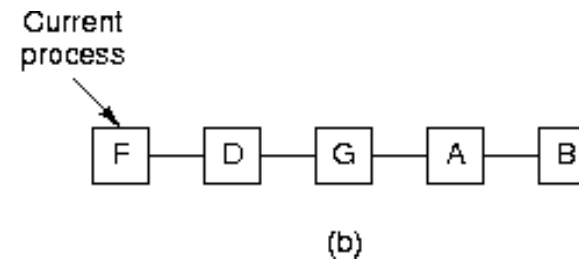
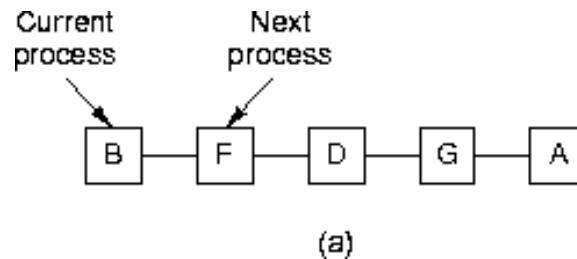
Running
Task

List of Ready
Tasks R



SCHEDULING ALGORITHMS

- ▶ In Round Robin all processes are equal



- ▶ What if B has a more important task to do with respect to the others?
 - ▶ After it run, it has to wait 4 time slices before running again
- ▶ Solution:
 - ▶ To differentiate processes assigning a weight factor → priority
 - ▶ To adopt a scheduling approach based on priority → priority-based scheduling

PRIORITY-BASED SCHEDULER

```
Schedule (ReadyList R, RunningTask T)
{
    Q = top_of_list( R );

    if( Priority(Q) > Priority(T) )
    {
        T->TCB.state = READY;
        save_context(T->TCB);

        append_to_list( T, R );

        restore_context(Q->TCB);
        Q->TCB.state = RUNNING;
    }
}
```

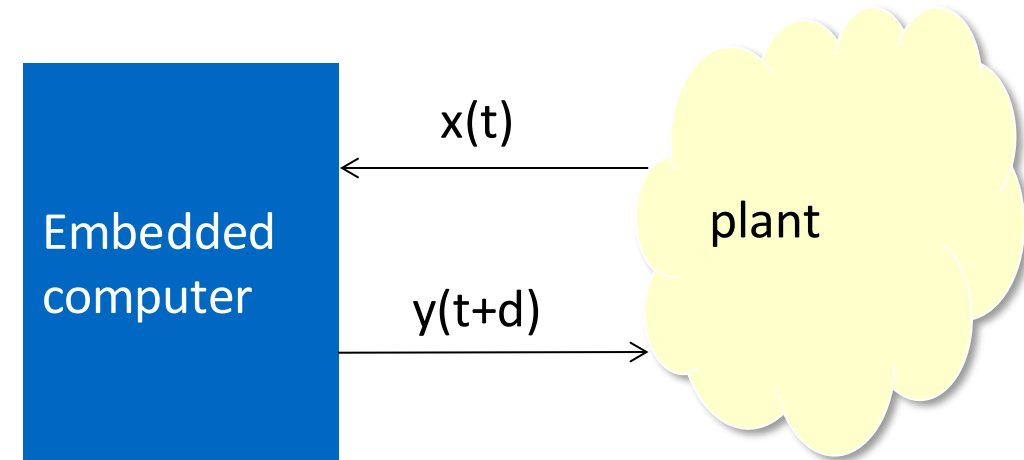
- ▶ Preemption of a running task & context switch happen only if a ready task exists whose priority is greater than that of the running task
- ▶ If the running task has priority equal to that of the highest priority ready task preemption does not happen
- ▶ Highest priority running task is preempted when it has to be blocked

SCHEDULING ALGORITHMS

- ▶ A priority function is defined which returns numerical value T for process p :
 - ▶ $T = \text{Priority}(p)$
- ▶ Static priority: unchanged for lifetime of p
- ▶ Dynamic priority: changes at runtime

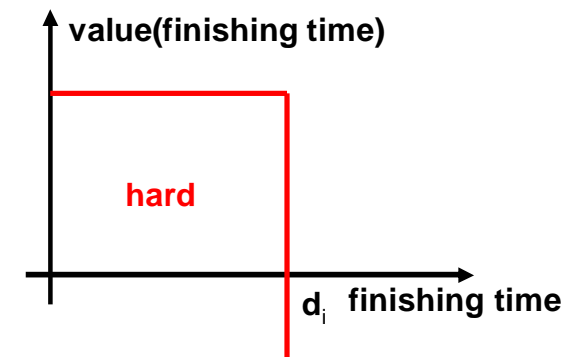
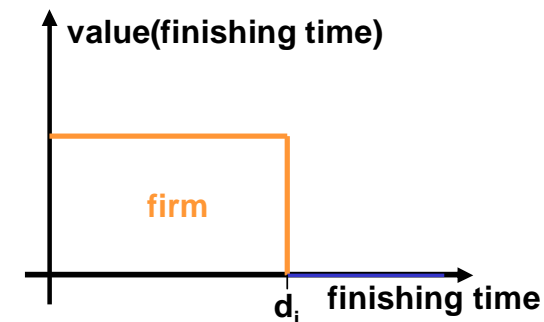
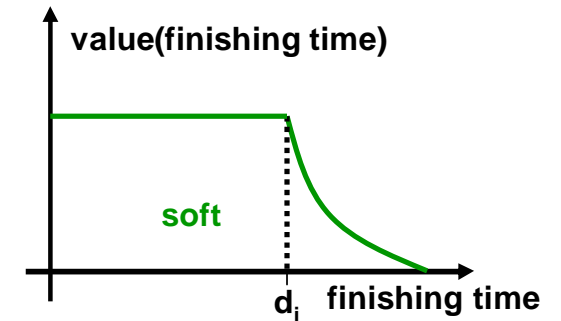
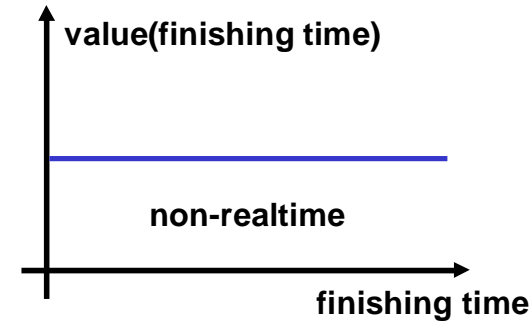
TYPE OF SYSTEMS

- ▶ Non real-time system: computer that has to respond to external events correctly
 - ▶ Example: given $x(t)$ at time $t \rightarrow$ output y must be delivered anytime
- ▶ Real-time system: computer that has to respond to external events both correctly and within a finite, specified period of time called deadline
 - ▶ Example: given $x(t)$ at time $t \rightarrow$ output y must be delivered no later than $t+d$
 - ▶ Right result too late is as bad as giving wrong or no result



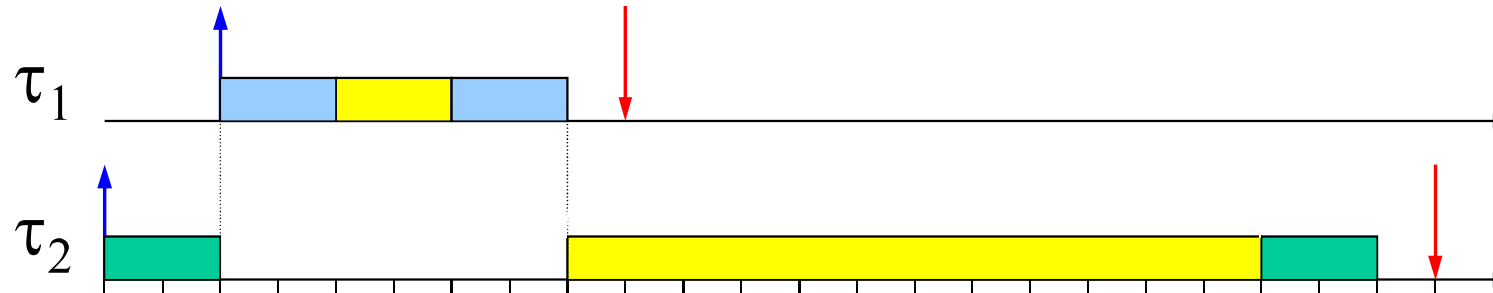
REAL TIME FLAVORS

- ▶ A process is **hard real time** if missing its deadline may cause catastrophic consequences on the environment under control
- ▶ A process is **firm real time** if missing its deadline makes the result useless, but missing does not cause serious damage
- ▶ A process is **soft real time** if meeting its deadline is desirable (e.g. for performance reasons) but missing does not cause serious damage

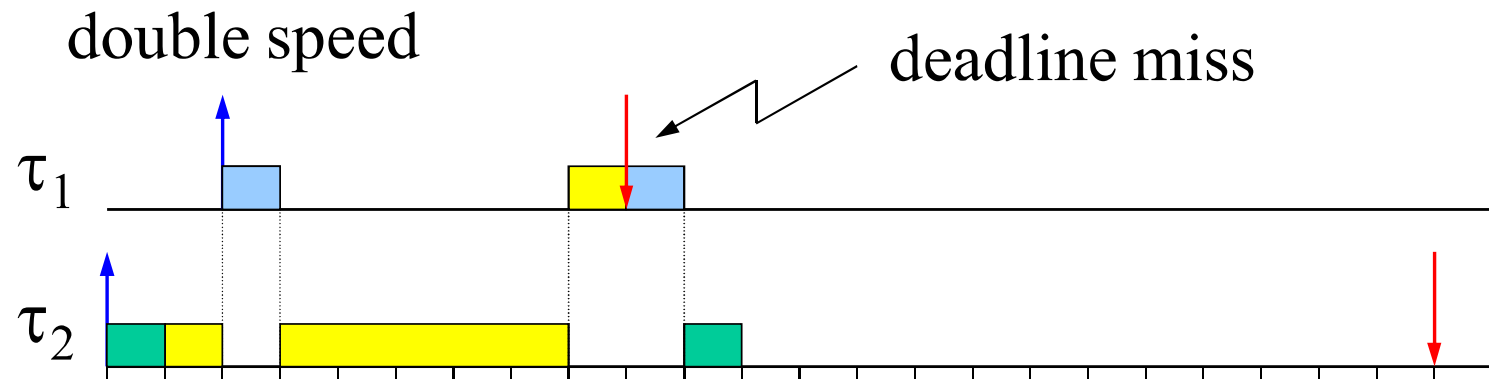


REAL TIME VS FAST

- ▶ The objective of a real-time system is to guarantee the timing behavior of each individual process



- ▶ The object takes to complete



QUESTIONS?

THANK YOU!

