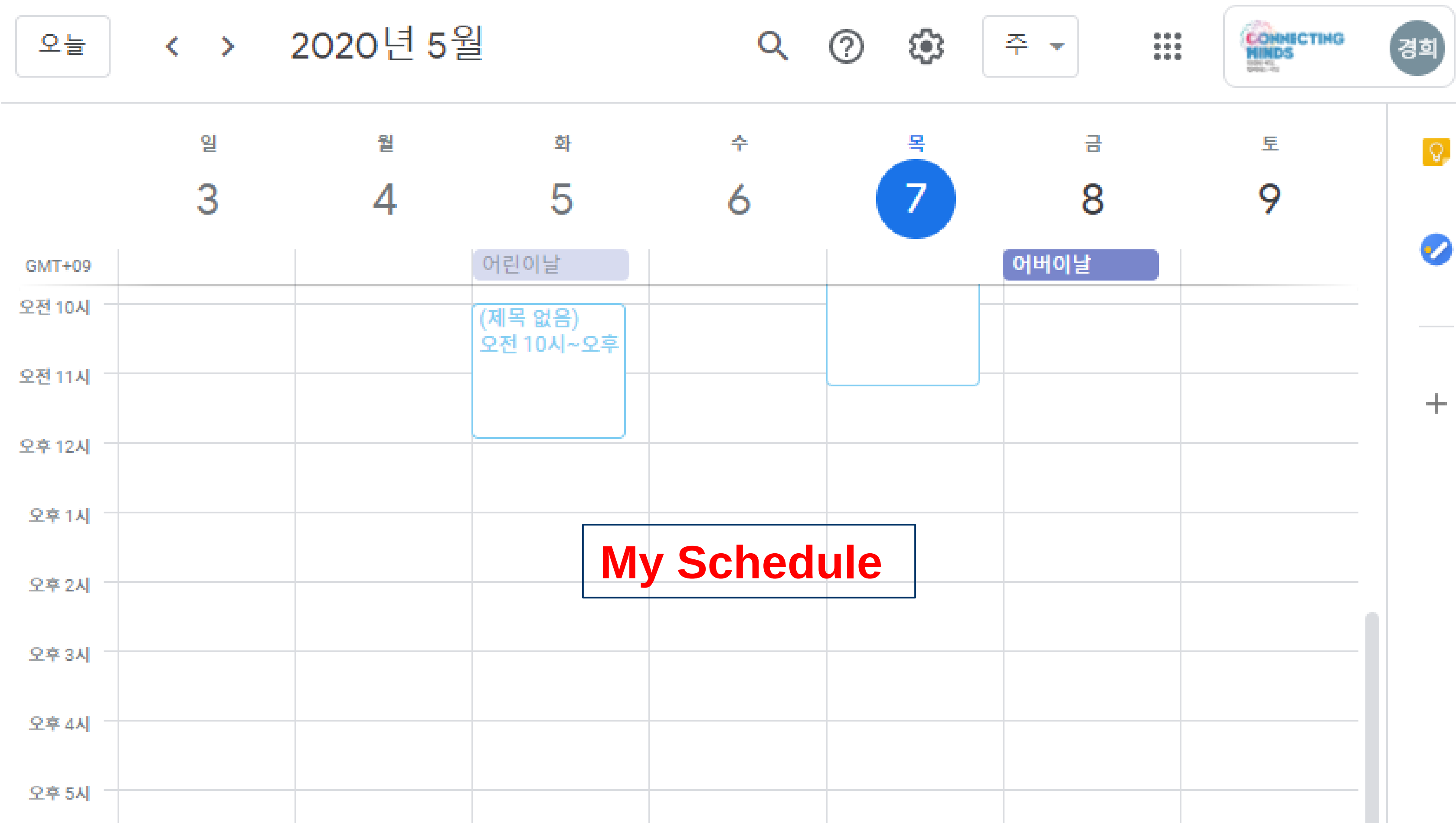
CH6. Process Scheduling

www.ajou.ac.kr
Department of Software
Kyunghee Choi

아주대학교

Contents

Scheduling/Schedule



Google Calendar

Scheduling/Schedule

< 진도 계획 >

주	강 의 주 제	언어	담당교수	수업방법	평가방법
1	강의소개 및 운영체제 개요	한	최경희	강의	
2	운영체제 구조	한	최경희	강의	
3	프로세스 관리 I	한	최경희	강의	
4	프로세스 관리 II	한	최경희	강의	
5	쓰레드	한	최경희	강의	
6	프로세스 동기화 I				
7	프로세스 동기화 II				
8	중간고사				
9	CPU 스케줄링				
10	데드락 관리				

운영체제 강의 스케줄

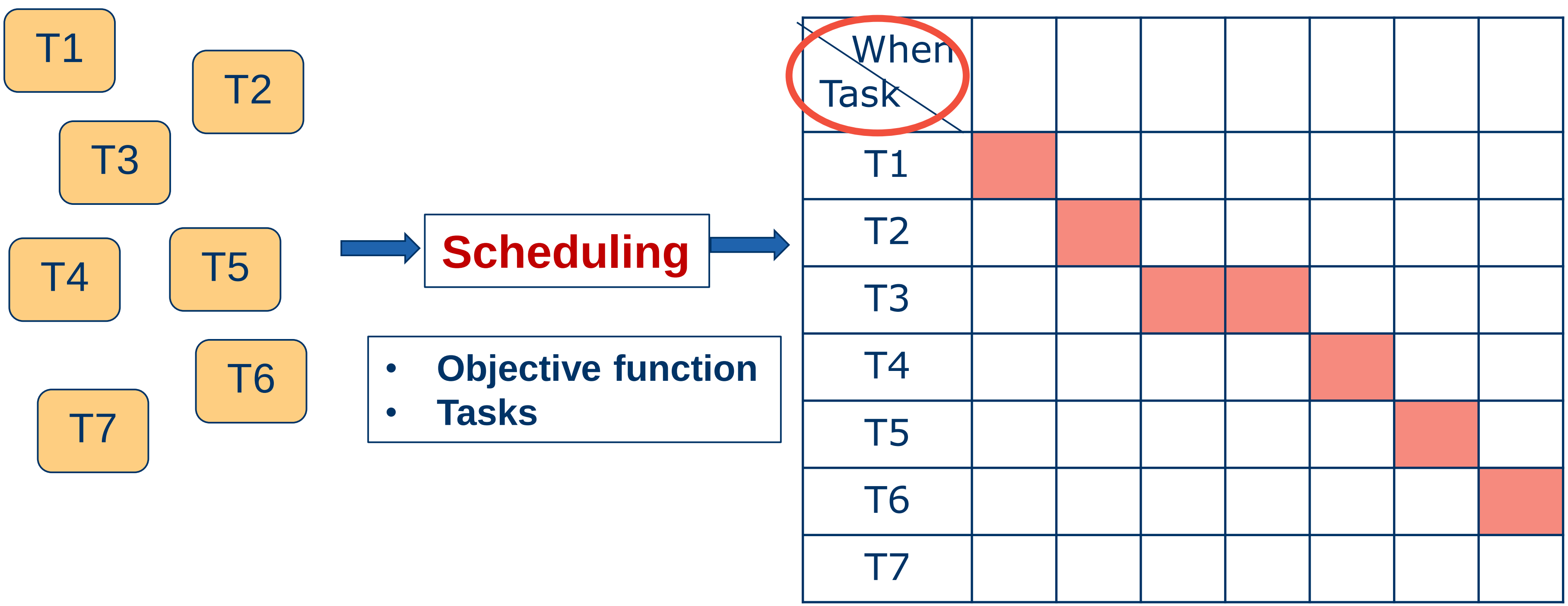
강의 내용	1	2	3	4	5	6	7	…
강의소개 및 운영체제 개요								
운영체제 구조								
프로세스 관리 I								
프로세스 관리 II								
쓰레드								
프로세스 동기화 I								
프로세스 동기화 II								

Scheduling/Schedule

- ❖ **작업 스케줄링**
- ❖ **공정스케줄링**
- ❖ **Scheduling in Kernel**
  - Packet Scheduling
  - IO Scheduling
  - CPU scheduling - process scheduling
  - Job Scheduling
- ❖ **Scheduling in Multimedia Application System**

Scheduling/Schedule

Key Points



1

Process Scheduling Criteria

Process Scheduling Criteria

- **CPU utilization**
  - keep the CPU as busy as possible (0-100%)
- **Throughput**
  - the number of processes that complete their execution per time unit
- **Turnaround time**
  - amount of time to execute a particular process
    (Turnaround time = completion time - submission time)
- **Waiting time**
  - amount of time a process has been waiting in the ready queue
    (neither CPU execution nor I/O blocking)
- **Response time**
  - amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Optimization Criteria

- **Optimization Criteria**
  - <u>Maximize</u> CPU utilization and throughput
  - <u>Minimize</u> turnaround time, waiting time, and response time
- **In most cases,**
  - optimize the <u>average</u> measures
- **Under some circumstances,**
  - optimize the <u>minimum or maximum</u> values, rather than average
- **For interactive systems**
  - minimize the <u>variance</u> in the response time

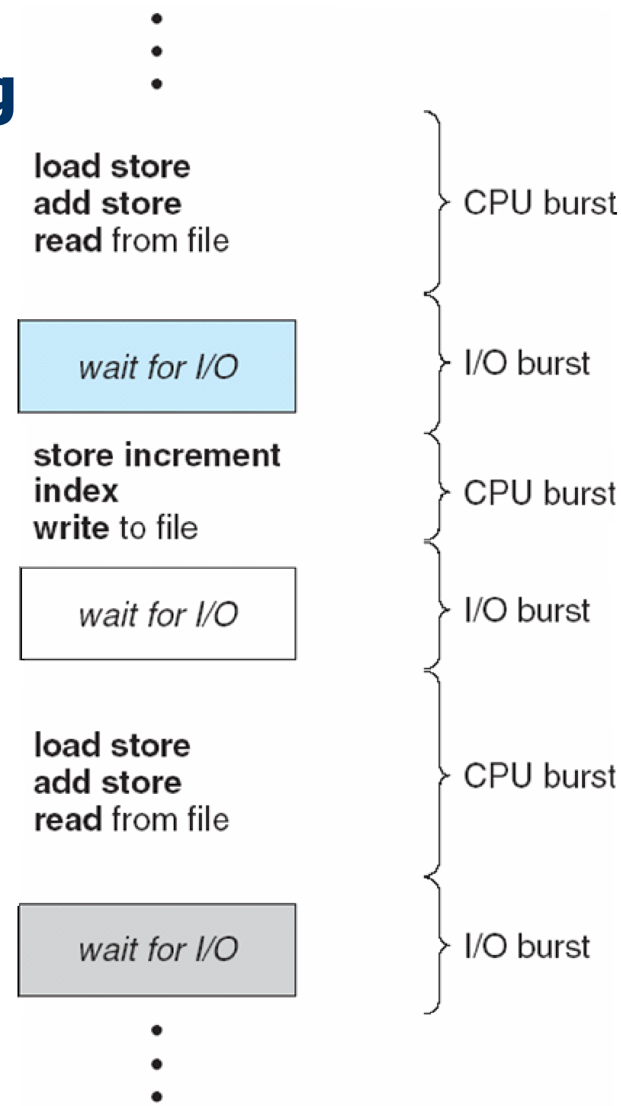
  → a system with reasonable and <u>predictable</u> response time

2 Basic Concepts

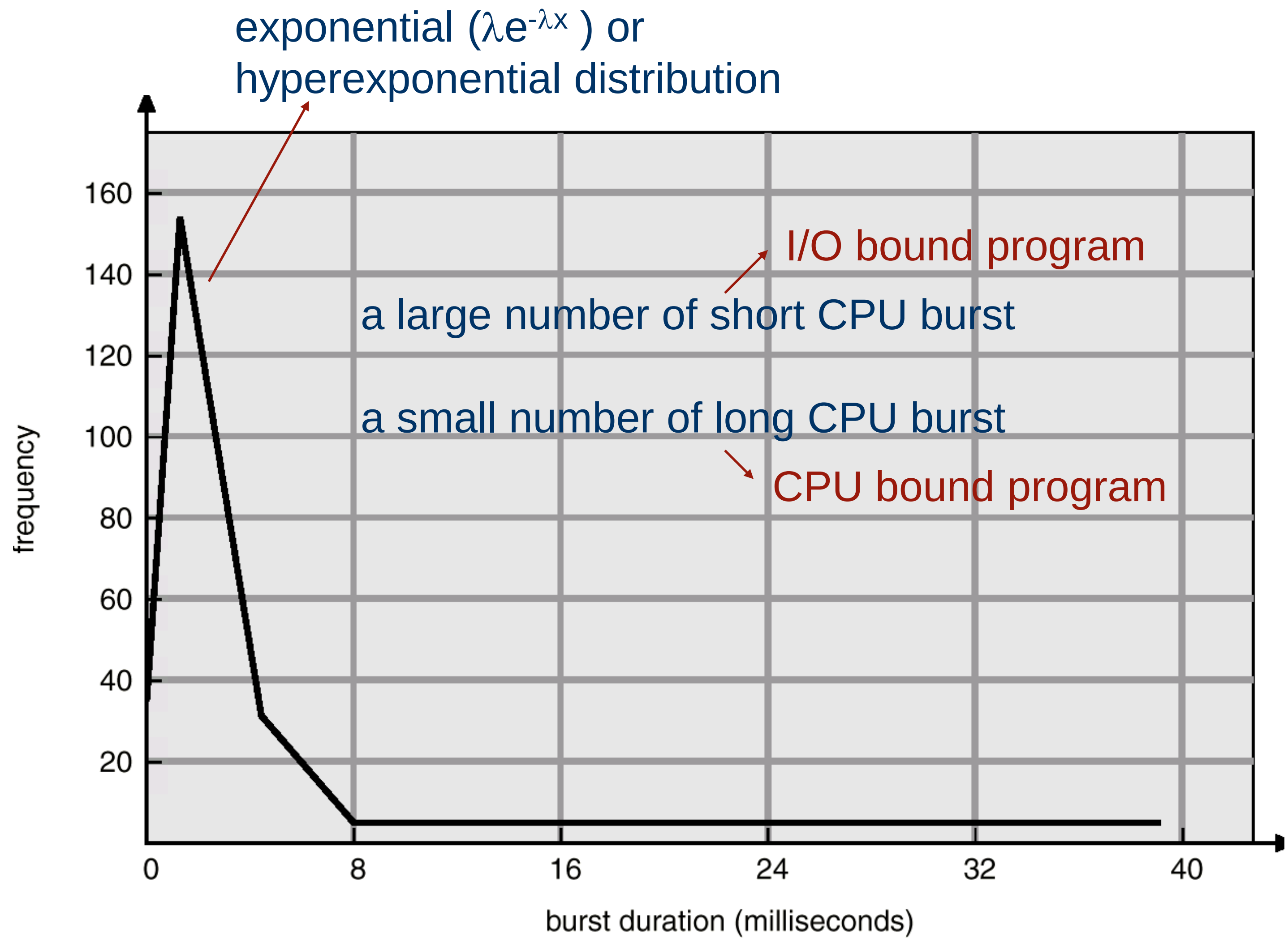
Basic Concepts

- **Objective of multiprogramming**
  - Maximize CPU utilization
- **CPU-I/O Burst Cycle**
  - Process execution consists of a cycle of CPU execution and I/O wait

    CPU burst I/O burst
- **CPU burst distribution**
  - (see next page)

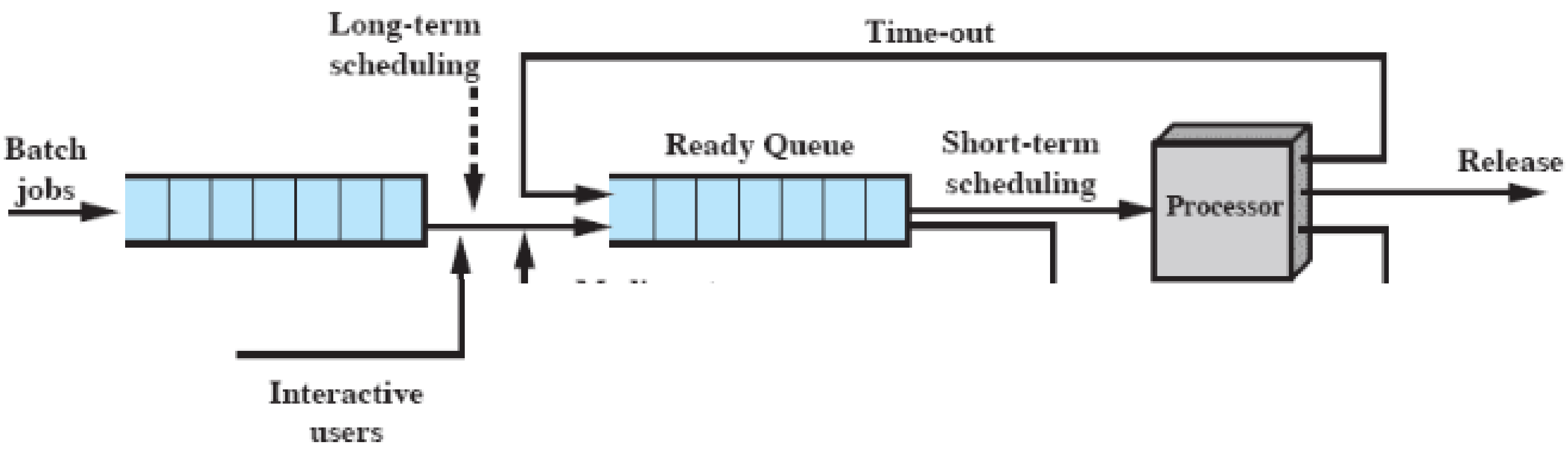


Histogram of CPU-burst Times



CPU Scheduler

- **CPU scheduler (short-term scheduler)**
  - Selects one of the processes in the ready queue, and allocates the CPU to that process.



When CPU Scheduling may take place ?

- **CPU scheduling decisions may take place <u>when</u> a process:**
  1. Switches from running to waiting state
     (e.g. I/O wait, child termination wait)
  2. Switches from running to ready state (e.g. time-out)
  3. Switches from waiting to ready state (e.g. I/O completion)
  4. Terminates

- case 1, 4 => there is ***<u>no choice</u>*** in terms of scheduling.
  a new process must be selected

- case 2, 3 => there is a **<u>choice</u>**.
  (a current process is in ready state)

Preemptive and Nonpreemptive Scheduling

- **Nonpreemptive (cooperative) scheduling:**
  - scheduling takes place *only* under cases 1 and 4
    - A processing in the running state will continue until it terminates or blocks itself
  - (ex) windows 3.x, old Mac OS
  - It is the only method that can be used on hardware platforms which do not have the special hardware (e.g. timer)
- **Preemptive scheduling:**
  - scheduling may take place under all cases
  - (ex) Most operating systems
  - It requires a special hardware (e.g. timer)

Problems of Preemptive scheduling

- **Preemptive scheduling incurs a cost associated <u>with coordination of access to shared data </u>→ synchronization mechanism (Chap 5)**
- **Race Condition**
  - When data are shared, one process updates that data and is preempted. After that, the second process tries to read that data
- **Preemption in user mode**
  - (e.g.) Two processes share data
    - While one is updating data, it is preempted and another tries to read or modify the data ➔ in an inconsistent state
- **Preemption in kernel mode:**
  - (e.g.) All kernel routines share the kernel data
    - The kernel may process a system call <u>on behalf of a process </u>and involves changing important kernel data.
      The process is preempted during these change and
      the kernel (or device driver) to read or modify the same data.
      ➔ in an inconsistent state (more dangerous)

Problems of Preemptive scheduling

- **Certain OSs simply deal with the preemption problem in kernel**
  - by waiting for
    (1) a system call to complete, or
    (2) a I/O block to take place
    before doing context-switching
  - This kernel-execution model is poor
    for supporting real-time computing and multi-processing.

Dispatcher

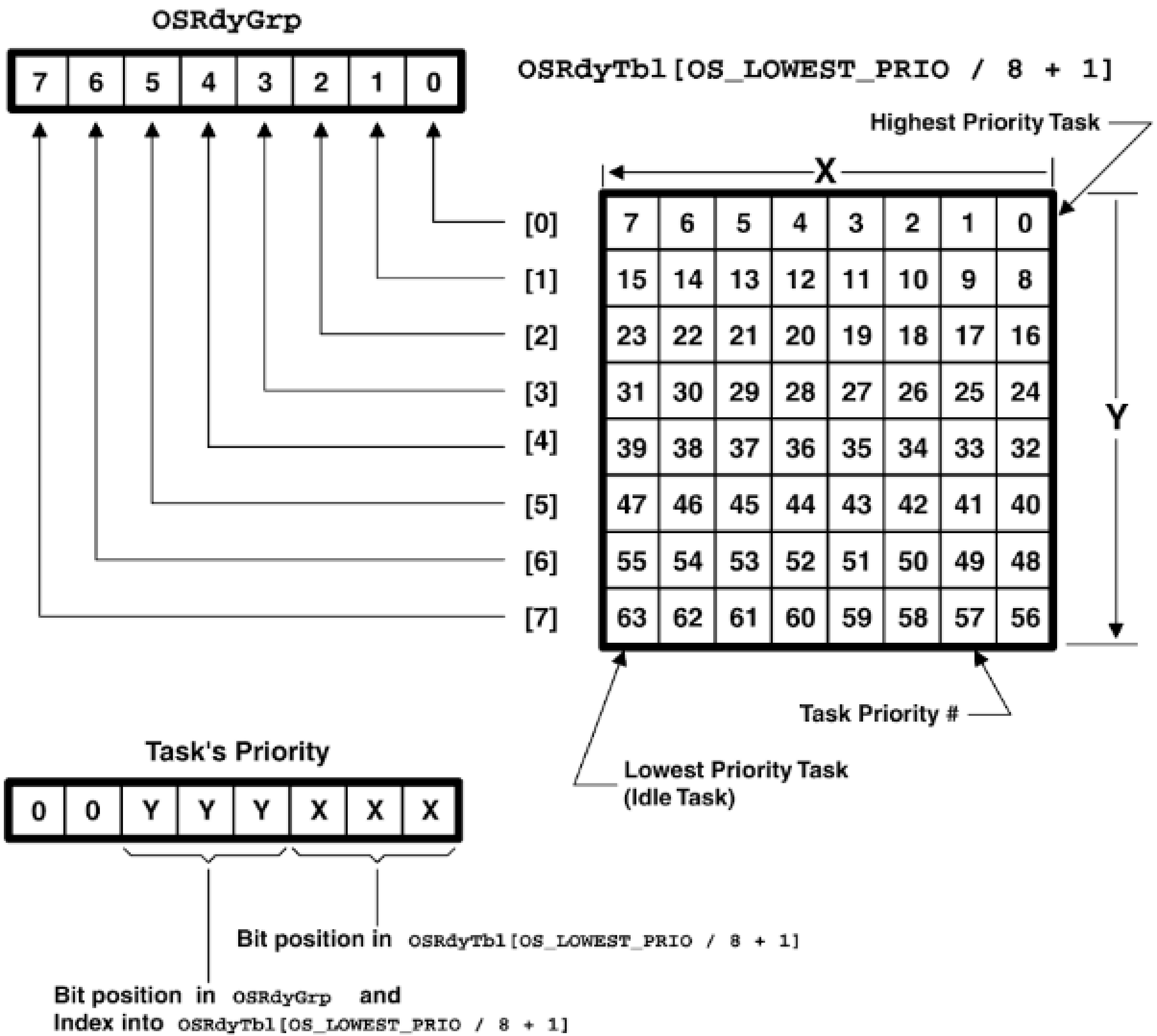
Dispatcher

- gives control of the CPU to the process selected by the short-term scheduler
- this function involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

Dispatch latency

- the time it takes for the dispatcher to *stop* one process and *start* another running.
- dispatch latency should be fast as possible

Dispatcher Example(uC/OS)



Dispatcher Example

```
OS_ENTER_CRITICAL();
if ((OSIntNesting == 0) && (OSLockNesting == 0)) {
    y             = OSUnMapTbl[OSRdyGrp];
    OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
    if (OSPrioHighRdy != OSPrioCur) {
        OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
        OSCtxSwCtr++;
        OS_TASK_SW();
    }
}
OS_EXIT_CRITICAL();
```

3 Scheduling Algorithms

Scheduling Algorithm

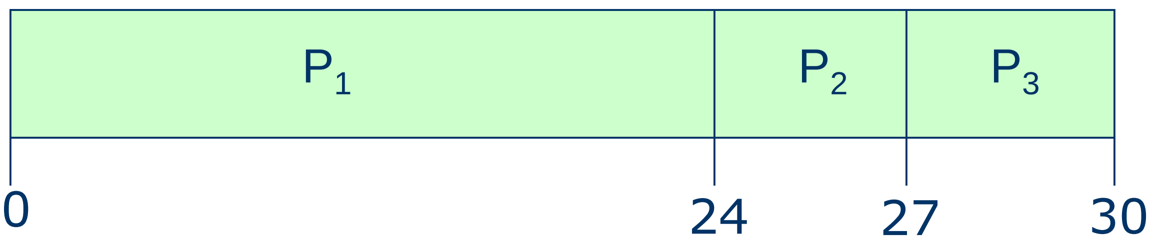
- First-Come First-Serve(FCFS) scheduling
- Shortest-Job-First (SJR) Scheduling
- Priority Scheduling
- Round Robin (RR)
- Multilevel Queue Scheduling

First-Come, First-Served (FCFS) Scheduling

- **Example:**

Process	Burst Time
P_1	24
P_2	3
P_3	3

- **Arrival order: P_1 , P_2 , P_3 (arrival time t=0)**
  - The *Gantt Chart* for the schedule is:

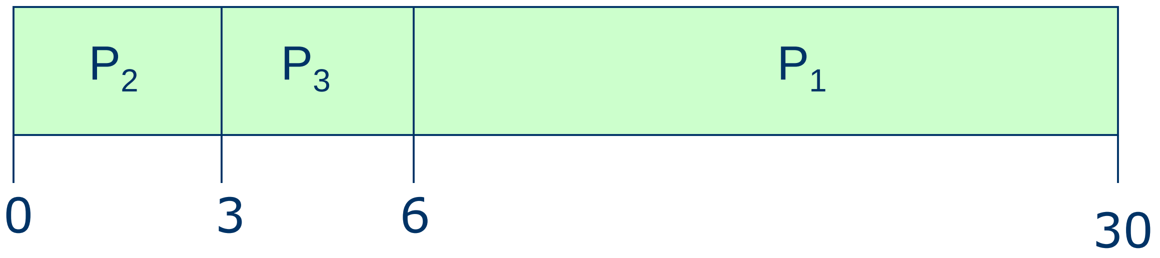


  - Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
  - Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

- **Arrival order: P_2 , P_3 , P_1 . (arrival time t=0)**

  The Gantt chart for the schedule is:



  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: (6 + 0 + 3)/3 = 3

    → Much better than previous case.

- ***Convoy effect***
  - short processes behind long process →

    short processes wait for the one long process to get off the CPU
  - This effect results in lower CPU and device utilization

Shortest-Job-First (SJF) Scheduling

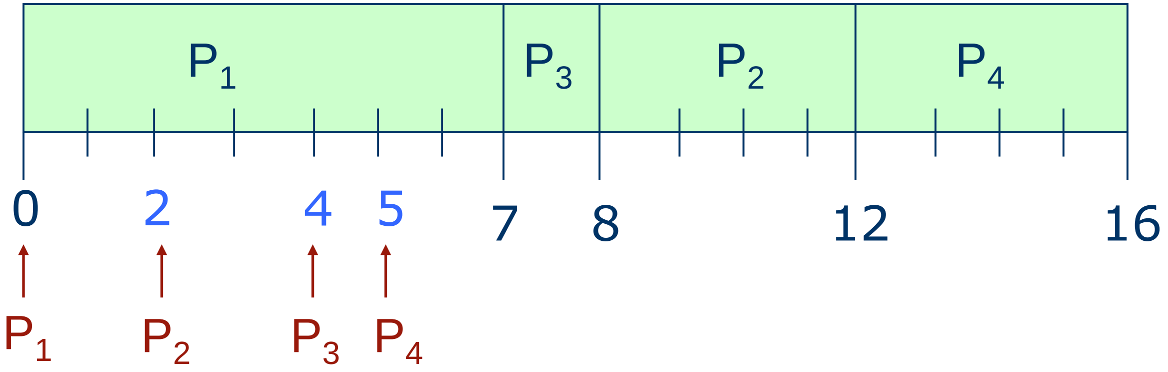
- **Shortest-Job-First (SJF) scheduling**
  (Shortest Process Next: SPN, Shortest Request Next: SRN)
  - each process is associated with the <u>length of its next CPU burst</u>
  - When the CPU is available,
    schedule the process with the *shortest* next CPU burst
  - it is difficult to know the next CPU burst → (solution) *prediction*
- **Two schemes:**
  - Nonpreemptive – The currently running process *cannot be preempted* until completes its CPU burst
  - Preemptive –
    if CPU burst of a new process < remaining time of a current process,
    then preempt → Shortest-Remaining-Time-First (SRTF)
    "preemptive SJF = SRTF"
- **SJF is optimal**
  - gives <u>*minimum* average *waiting time*</u> for a given set of processes

Example of Non-Preemptive SJF

- **Example**

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- **SJF (non-preemptive)**



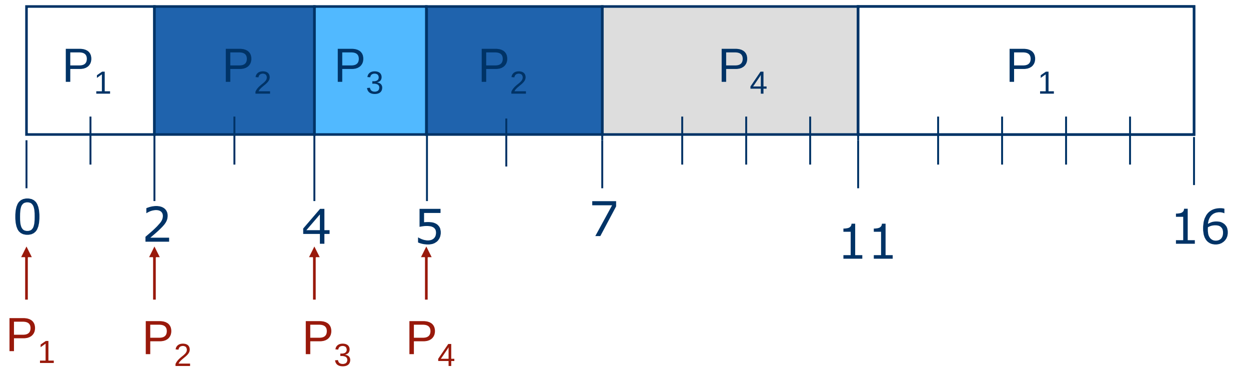
  - Average waiting time = (0 + 6 + 3 + 7)/4 = 4

Example of Preemptive SJF

Example

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (preemptive)

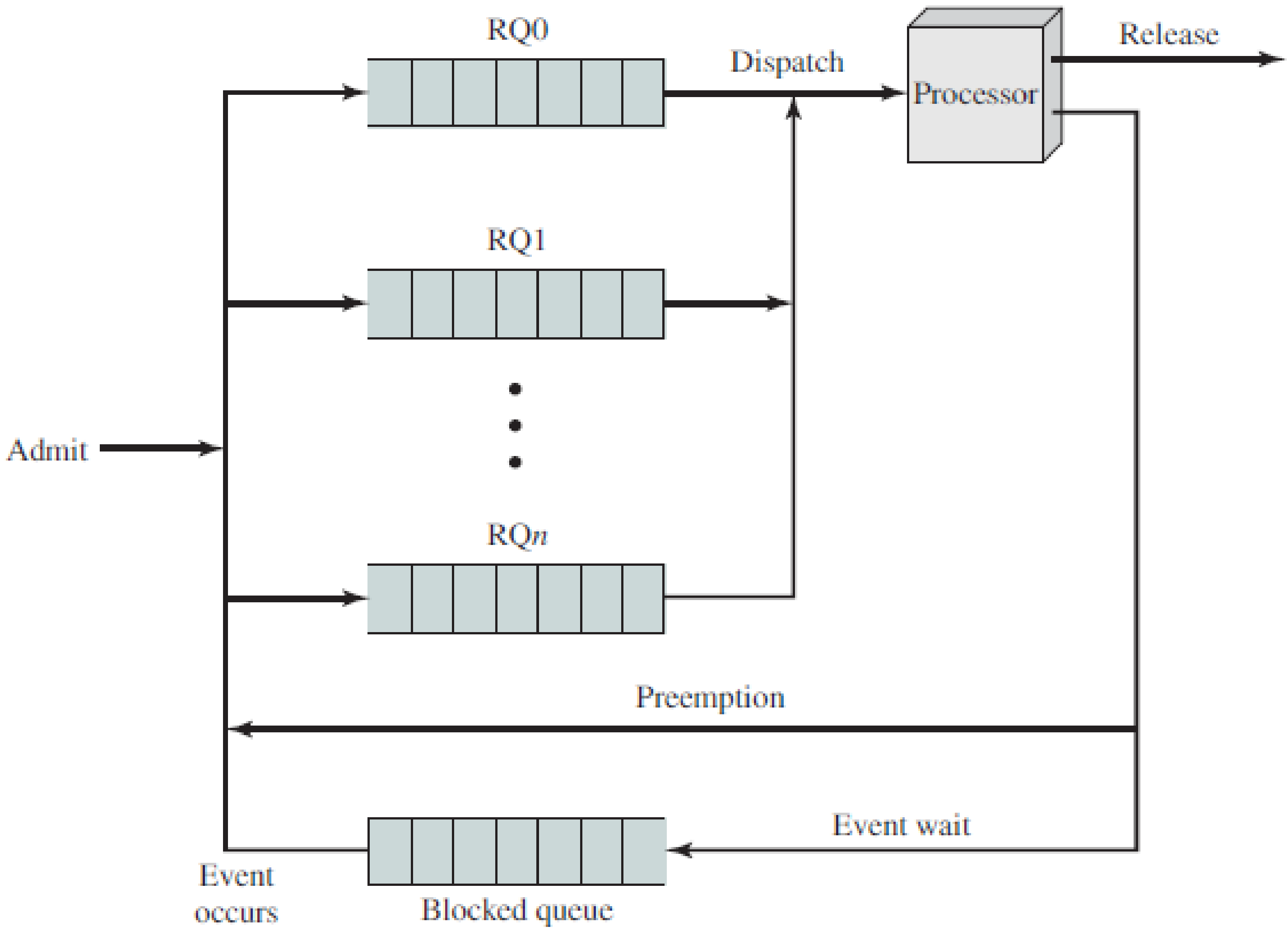


- Average waiting time = (9 + 1 + 0 +2)/4 = 3

Priority Scheduling

- **Priority Scheduling**
  - Each process is associated with a priority number (integer)
  - The CPU is allocated to the process with the highest priority (usually, smallest priority number ≡ highest priority)
- **Priorities can be defined**
  - internally: time limits, memory requirements, # of open files, ratio of I/O to CPU
  - externally: importance of process, fund type and amount, political factor
- **Two schemes**
  - preemptive
  - nonpreemptive
- **SJF is a special case of priority scheduling**
  - priority = the predicted next CPU burst time

Priority Queueing



Starvation

- **Problem of priority scheduling → Starvation (Indefinite blocking)**
  - low priority processes may *never* execute.

- **Solution of starvation → aging**
  - **Aging**: a technique of gradually increasing the priority of processes that wait in the system for a long time

Round Robin (RR) Scheduling

- **Round Robin scheduling (processor sharing)**
  - Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
  - After this time has elapsed, the process is <u>preempted</u> and added to the end of the ready queue
- **Maximum waiting time**
  - For n ready processes, time quantum q ,

    max. waiting time = (n-1)q
- **Performance**

s: context switch time

  - q large $\Rightarrow$ FIFO (FCFS)
  - q small $\Rightarrow$ q must be large with respect to context switch time, otherwise, overhead is too high

Example: RR with Time Quantum = 20

- **Example**

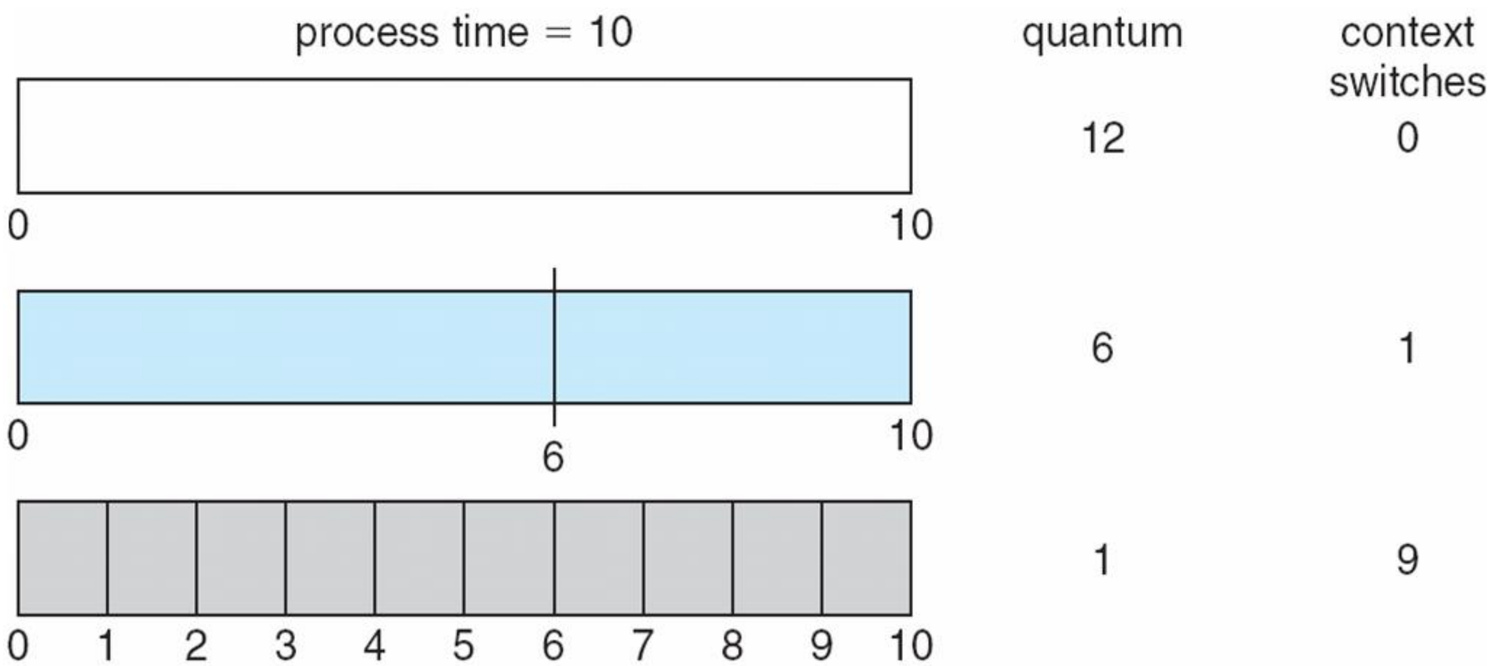
Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- **The Gantt chart is:**

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

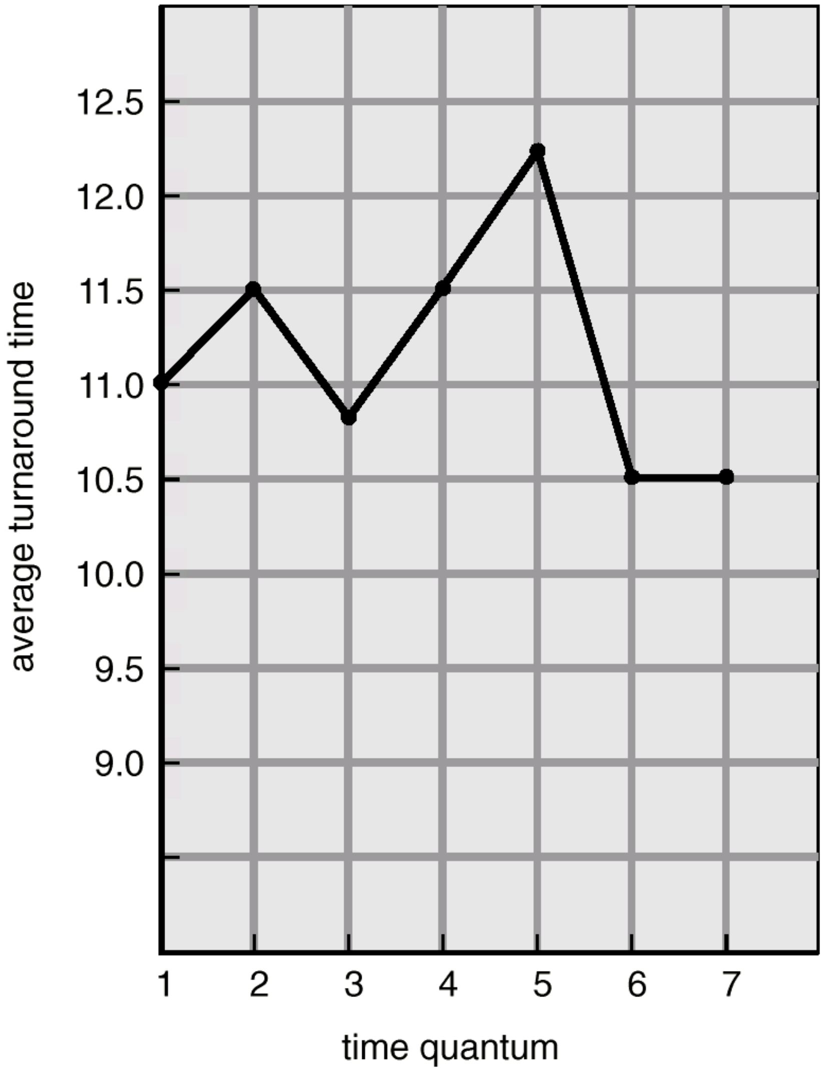
0 20 37 57 77 97 117 121 134 154 162

Time Quantum and Context Switch Time



- **Smaller time quantum increases context switches**
- **A rule of thumb:**
  - 80% of CPU bursts should be shorter than quantum time

Turnaround Time Varies With The Time Quantum



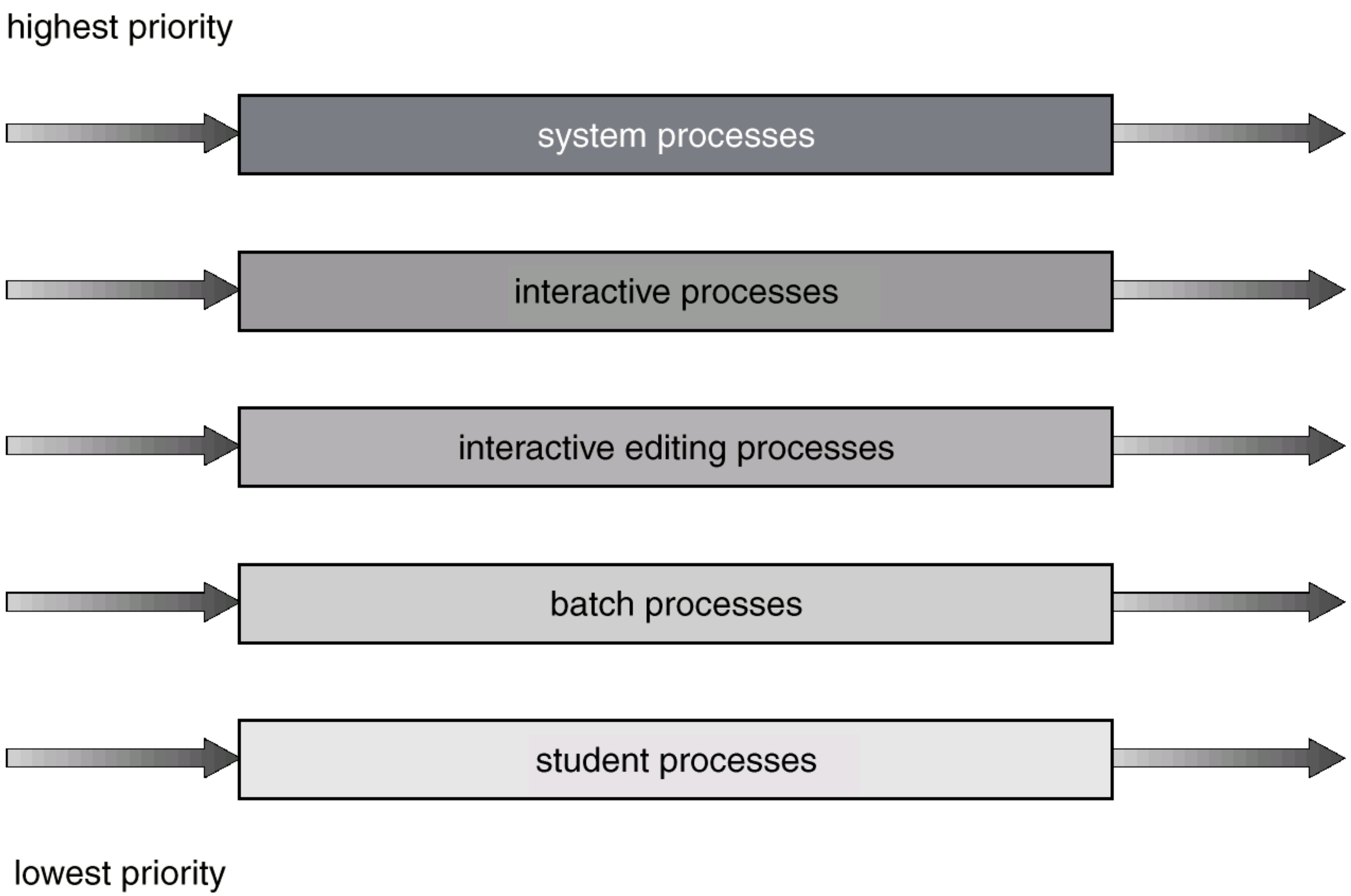
process	time
P_1	6
P_2	3
P_3	1
P_4	7

- **turnaround time does *not necessarily* improve as time-quantum size increases**

Multilevel Queue Scheduling

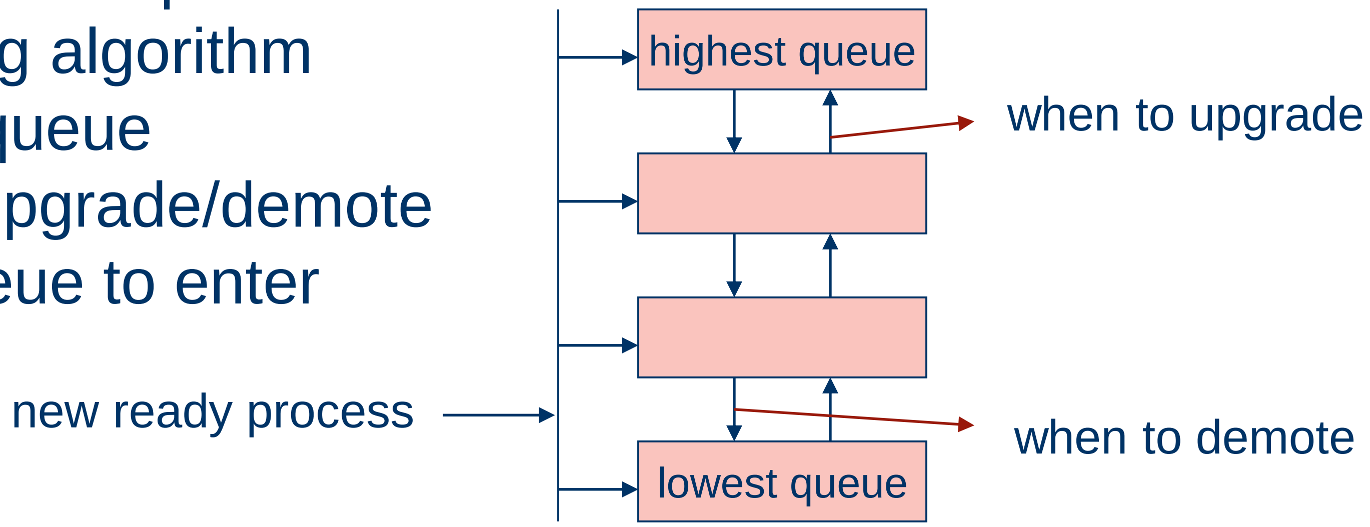
- **Multilevel Queue Scheduling**
  - Ready queue is partitioned into separate queues:
  - Each queue has its *own* scheduling algorithm,
    e.g. foreground (interactive) – RR
    background (batch) – FCFS
- **Scheduling must be done between the queues**
  - Fixed priority scheduling
    - serve all from foreground, then from background
    - Possibility of *starvation*
  - Time slice
    - each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - e.g. 80% to foreground in RR, 20% to background in FCFS

Multilevel Queue Scheduling



Multilevel Feedback Queue Scheduling

- **A process can move between the various queues**
  - **Aging** can be implemented this way ➔ flexible
- **Multilevel-feedback-queue scheduler defined by the following parameters:**
  - the number of queue
  - scheduling algorithm for each queue
  - when to upgrade/demote
  - which queue to enter

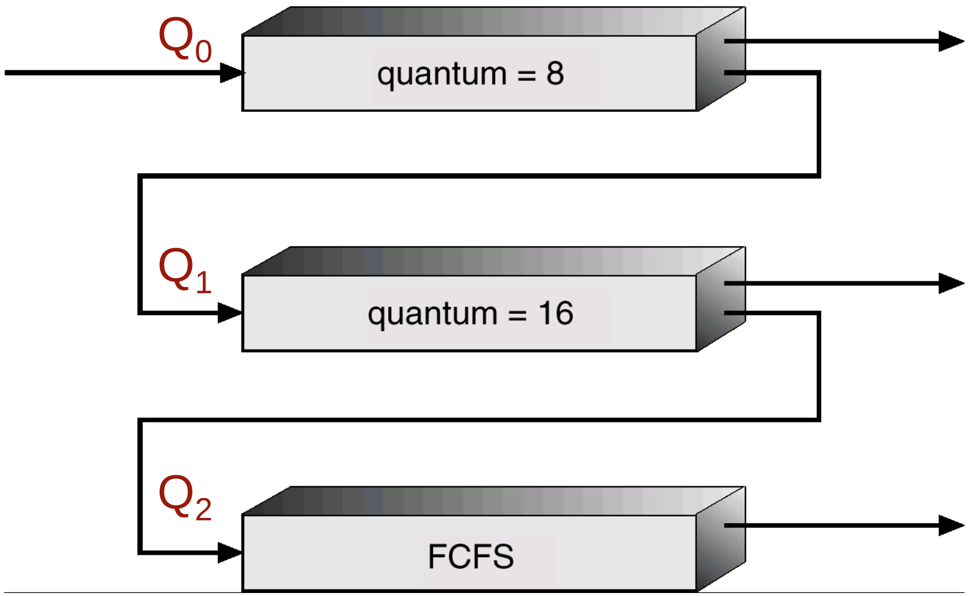


Multilevel Feedback Queues

(high priority)

I/O bound or
Interative processes

CPU bound or
Batch processes



Example:

- A new job => queue Q_0 (q=8, FCFS)
- If it does not finish in 8 ms, the job => queue Q_1 (q=16, FCFS)
- If it still does not complete, the job => queue Q_2
- priority: $Q_0 > Q_1 > Q_2$
    - The process in Q_2 are run, only when Q_0 and Q_1 are empty

4

Thread Scheduling

Thread Scheduling

- **Distinction between user-level threads and kernel-level threads**
- **Process-contention scope (PCS) - for user-level threads**
  - On system using many-to-one mapping or many-to-many mapping, the thread library schedules user-level threads to run on an available LWP (lightweight process)
  - competition for CPU takes place among threads in the same process
  - Typically, PCS is done according to priority
- **System-contention scope (SCS) - for kernel-level threads**
  - the kernel schedules kernel-level threads onto available CPU
  - competition for CPU takes place among all threads in the system
  - Systems using one-to-one mapping schedule threads using only SCS
- **Pthread scheduling policies - allow specifying during thread creation**
  - PTHREAD_SCOPE_PROCESS: PCS scheduling (many-to-many)
  - PTHREAD_SCOPE_SYSTEM: SCS scheduling (one-to-one)

Pthread scheduling

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
void *runner(void *param); /* the thread runs in this function */

int main(int argc, char *argv[])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];     /* the thread identifier */
    pthread_attr_t attr;        /* set of attributes for the thread */

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm PROCESS or SYSTEM */
    /* On Linux, Mac OS X, only PTHREAD_SCOPE_SYSTEM is supported.
     * Solaris supports both. */
    if (pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM) != 0)
        fprintf(stderr, "Unable to set scheduling scope\n");
    /* now create the threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],&attr,runner,NULL);
    /* Now join on each thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

void *runner(void *param)
{
    /* do some work */
    ...
    pthread_exit(0);
}
```

5

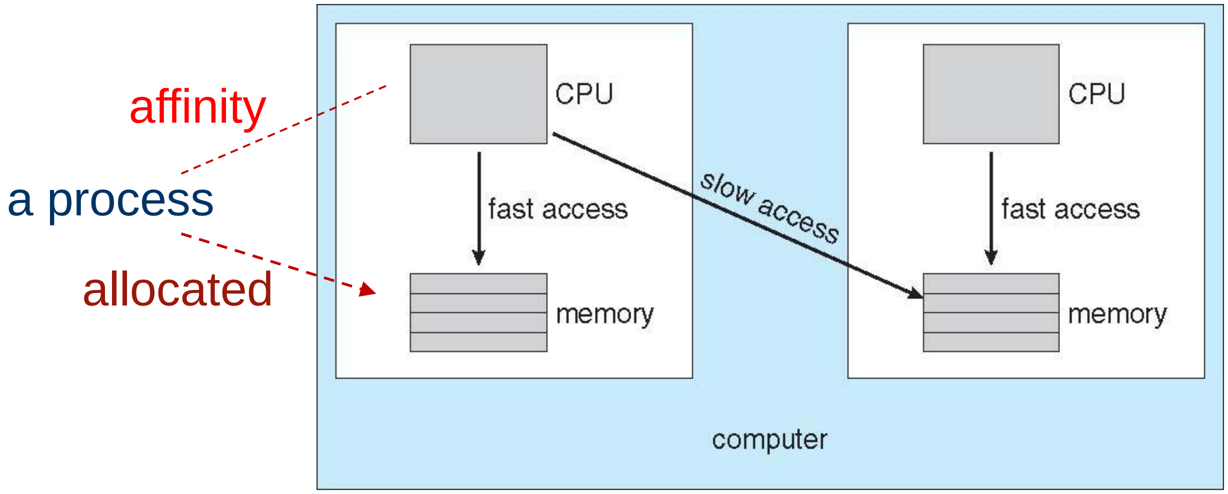
Multiple-Processor Scheduling

Multiple-Processor Scheduling

- **Multiple-processor scheduling**
  - CPU scheduling more complex when multiple CPUs are available
- **Multiprocessor system**
  - system in which the processors are identical (homogeneous)
- **Multiple-processor scheduling approaches**
  - **Asymmetric** multiprocessing
    - only one processor(master processor) accesses the system data structures, alleviating the need for data sharing
    - the other processors execute only user code
    - reduce the need for data sharing → simple
  - **Symmetric** multiprocessing (SMP) - self-scheduling
    - each processor is self-scheduling
    - ready queue
      - a common ready queue, or
      - separate queue for each processor (in most contemporary OS's supporting SMP)

Processor Affinity

- **Processor Affinity - a process has an <u>affinity</u> for the processor on which it is currently running;**
  - The most SMP systems avoid migration of processes from one processor to another, instead attempt to keep a process running on the same processor
  - Because of the high cost of invalidating and repopulating caches
- **Forms of processor affinity**
  - **soft affinity** - possible to migrate between processors
  - **hard affinity** - specify a processor or processor sets
- **NUMA(non-uniform memory access) and CPU scheduling**



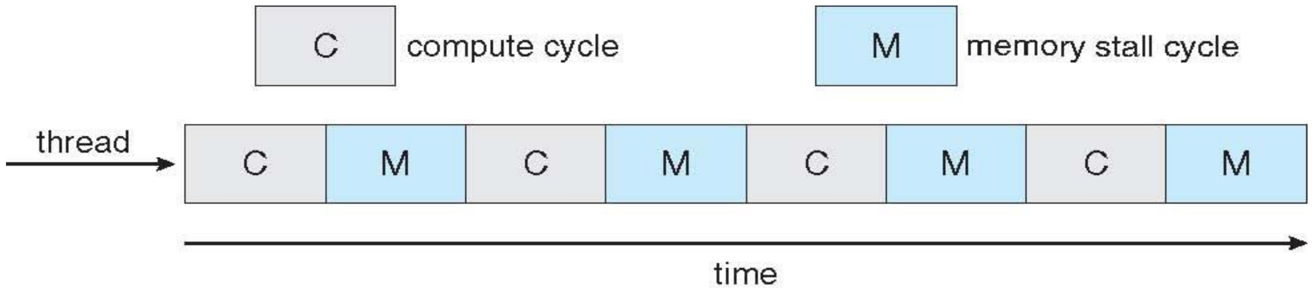
Load Balancing

- **Load balancing**
  - keep the workload evenly distributed across all processors
- **On systems with common ready queue**
  - load balancing is often unnecessary
- **On systems with separate private queue – most contemporary OS**
  - two approaches to load balancing
    - **push** migration:
      - periodically checks & if an imbalance is found, then migrate processes from overloaded to idle or less-busy processors
    - **pull** migration
      - an idle processor pulls a waiting task from a busy processor
  - two migration approaches are not mutually exclusive (both cannot happen at the same time)
- **load balancing often counteracts the benefits of processor affinity**

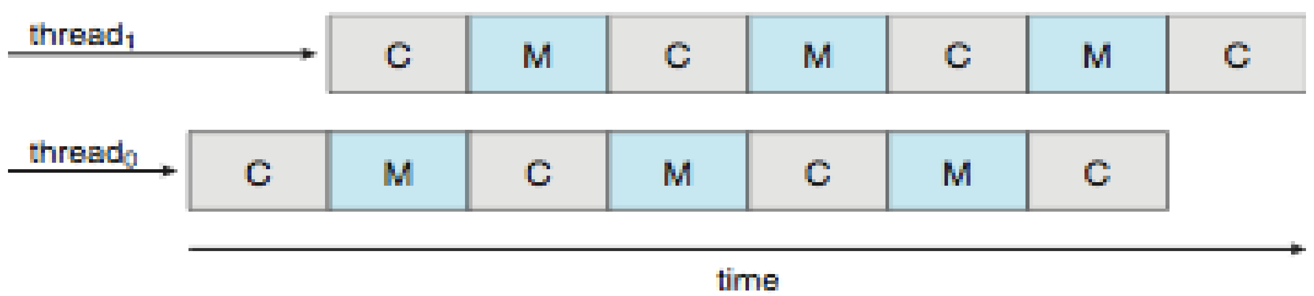
Multicore Processors

Multithreaded processor

- multiple logical processor on a physical processor (core)
- each logical processor has its own registers → fast context switching
- Takes advantage of memory stall to make progress on another thread while memory retrieve happens



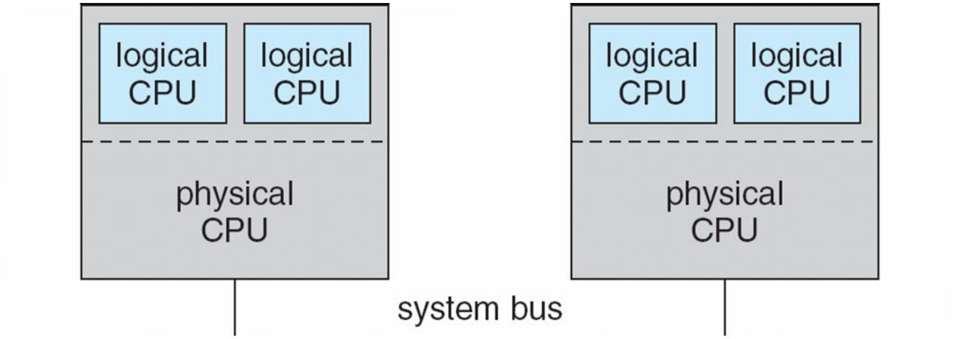
single threaded processor



multi-threaded processor

Multicore Processors (cont')

- **Multicore processor**
  - multiple processor core on same physical chip



a multithreaded multicore processor

- **Two different levels of scheduling**
  - choose which software thread to run on each hardware thread (logical processor) – by OS (use any scheduling algorithm)
    - → coarse-grained multithreading
    - → the cost of switching between threads is high
  - choose which hardware thread to run on the core – by hardware
    - → fine-grained multithreading
    - → the cost of switching between threads is small

6

Real-time Scheduling

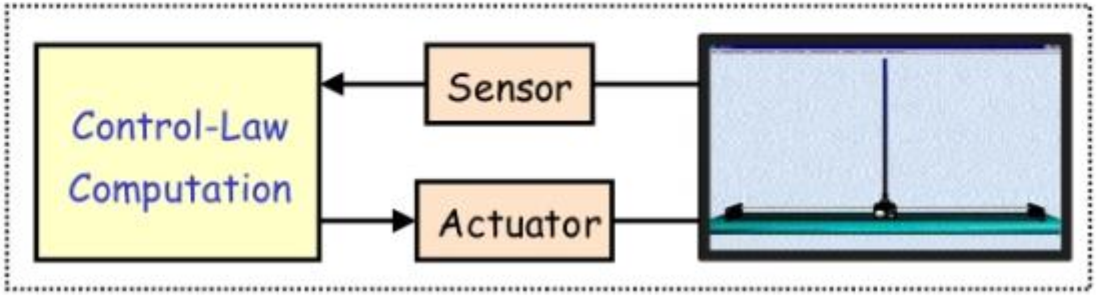
Real-time Systems

❖ Example of Real-time System

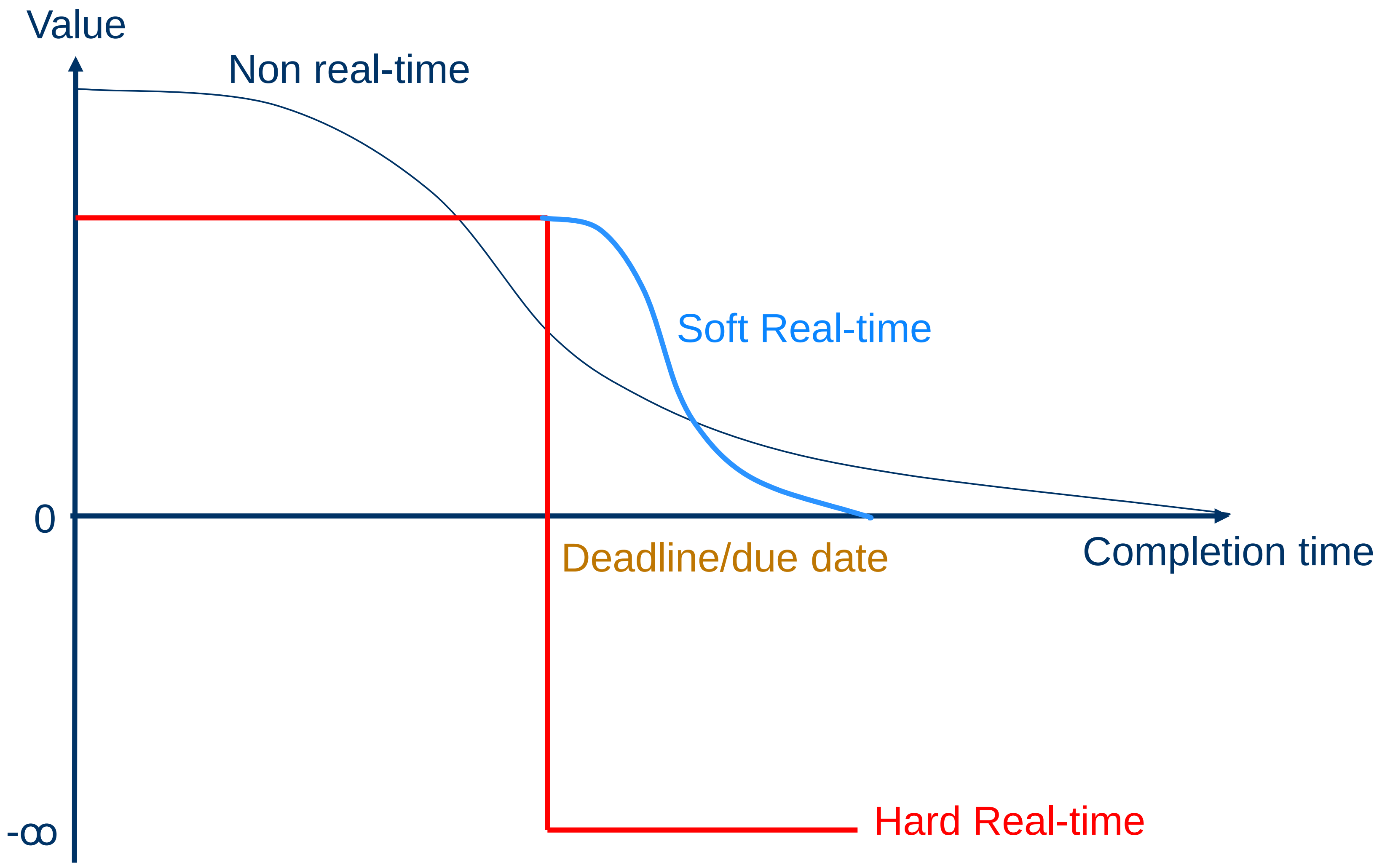
- Air Traffic Control Systems
- Networked Multimedia Systems
- Command Control Systems
- Space Shuttle
- Missile
- Train Control System
- Electronic Control System (Engine, Transmission, Headlight, ABS, etc)
- Weapons
- Factory Automation System

❖ Correctness of System

- Logically correct result of computation
- Time at which the results are produced



Real-time Systems



Real-Time Scheduling

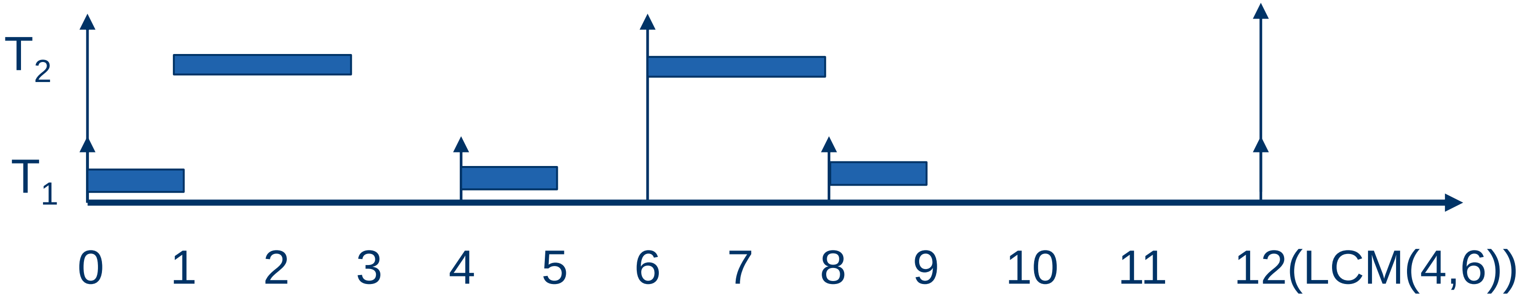
	C_i	P_i
T_1	1	4
T_2	2	6

C_i: Computation Time
P_i: Period

RM(Rate Monotonic) Scheduling
- Fixed priority
- Shorter period, higher priority

Not Schedulable

Schedulable



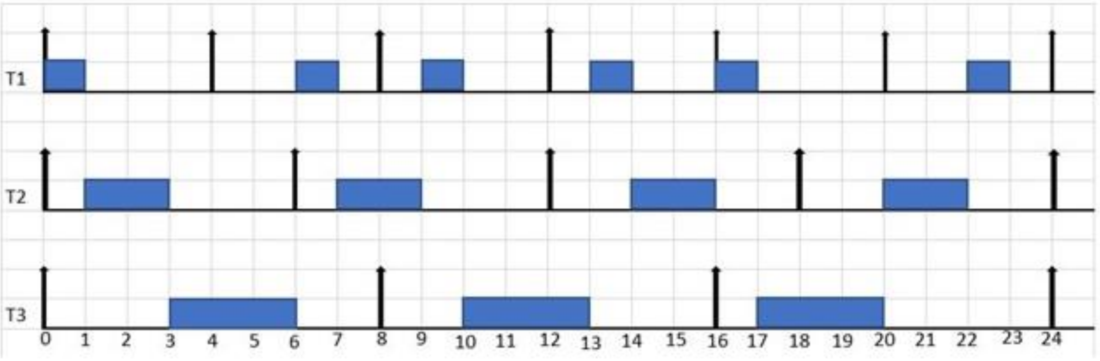
Real-Time Scheduling

	C_i	P_i
T_1	1	4
T_2	2	6
T_3	3	8

EDF(Earliest Deadline First) Scheduling
- Dynamic Priority
- Earlier Deadline, higher priority

Not Schedulable

Schedulable



Schedulability Analysis

- **Schedulability Analysis**
  - Total Utilization $U = \sum U_i = \sum (C_i/P_i)$
  - If $(U \leq n(2^{1/n}-1)$, schedulable by RMS
  - If $(U \leq 1)$, schedulable by EDF
- $\lim n(2^{1/n}-1) = \ln 2 \approx 0.693$