Embedded Operating System

Embedded System Software Design

Prof. Ya-Shu Chen
National Taiwan University of Science and Technology

Objectives

- Multitasking
- Scheduling algorithms
- Resource managment
- Power management

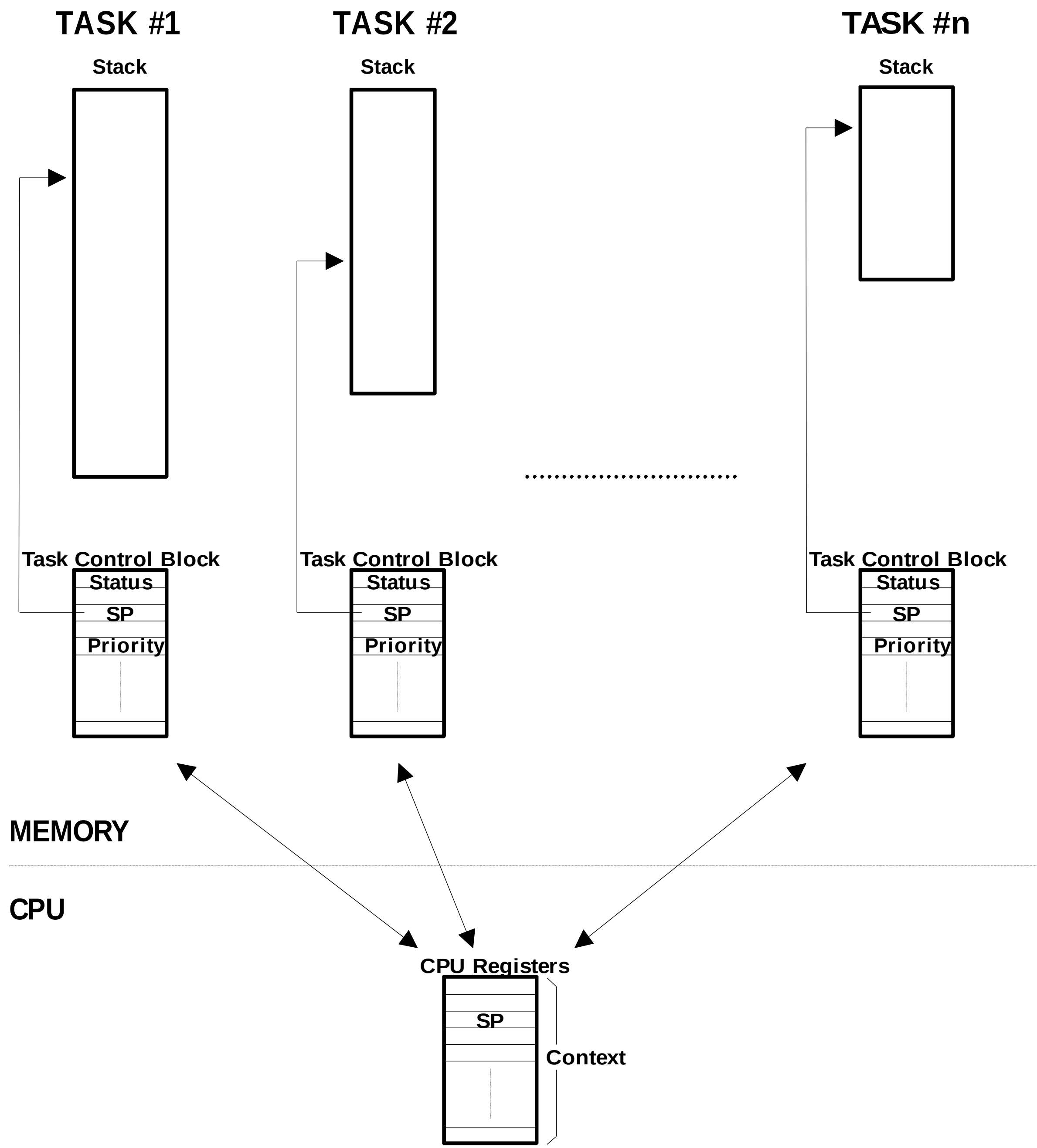
Multitasking

- The scheduler switches the attention of CPU among several tasks
  – Tasks logically execute concurrently by sharing the CPU
  – How much CPU share could be obtained by each task depends on the scheduling policy adopted

Task

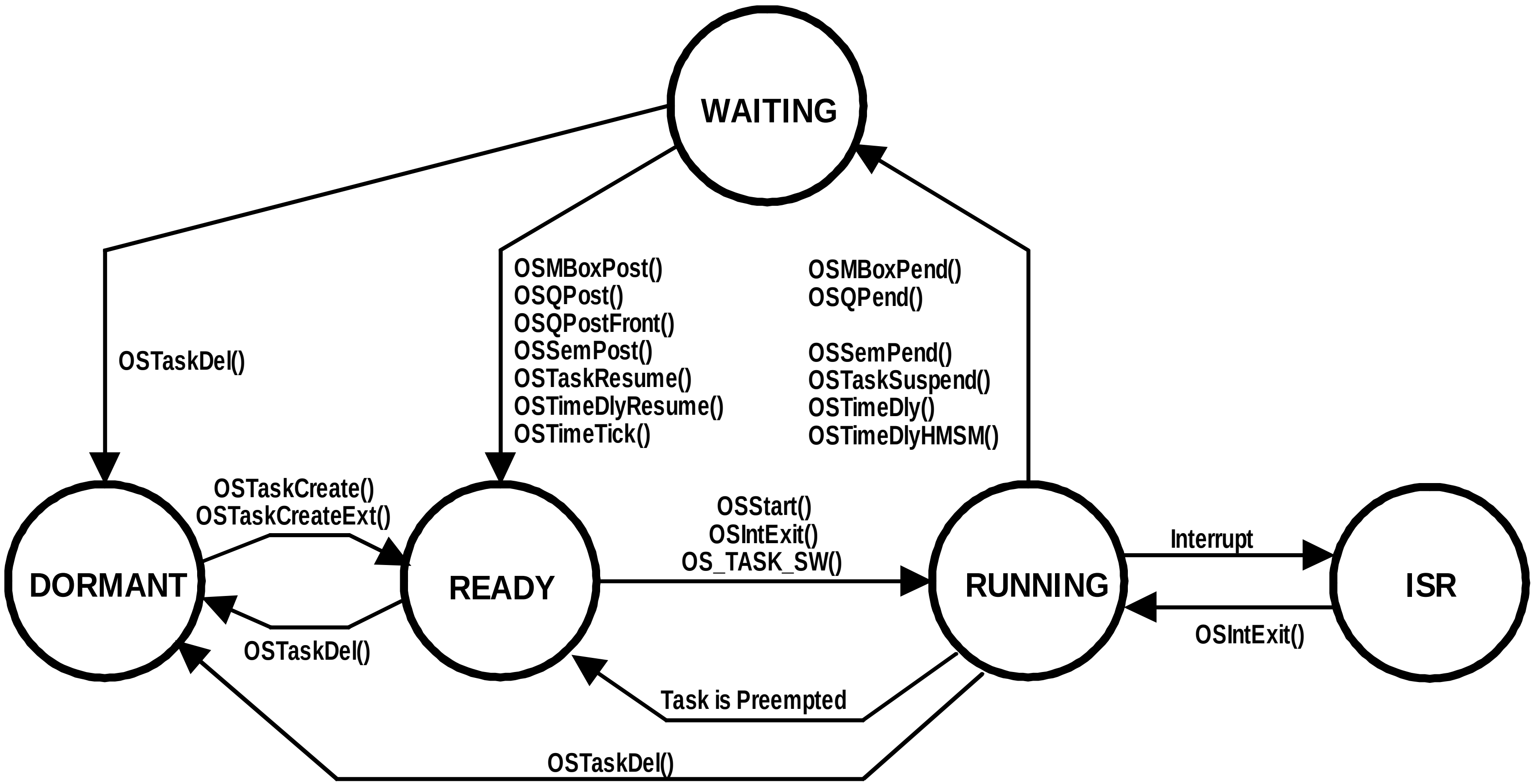
- Sometimes referred to as a process
  - An active entity that does computation

- From the OS point of view, a task is of a priority, a set of registers, its own stack, and some housekeeping information


TASK #1
TASK #2
TASK #n
Stack
Stack
Stack
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context

Task

- A thread or a process in practice. It is considered as an active/executable entity in a system.
- From the perspective of OS, a task is of a priority, a set of registers, its own stack area, and some housekeeping data.
- From the perspective of scheduler, a task is of a series of consecutive jobs with regular ready time (for periodic tasks, μC/OS-II).
- There are 5 states under μC/OS-II :
  - Dormant, ready, running, waiting, interrupted.


WAITING
OSTaskDel()
OSMBoxPost()
OSQPost()
OSQPostFront()
OSSemPost()
OSTaskResume()
OSTimeDlyResume()
OSTimeTick()
OSMBoxPend()
OSQPend()
OSSemPend()
OSTaskSuspend()
OSTimeDly()
OSTimeDlyHMSM()
OSTaskCreate()
OSTaskCreateExt()
DORMANT
READY
OSStart()
OSIntExit()
OS_TASK_SW()
RUNNING
Interrupt
ISR
OSIntExit()
OSTaskDel()
Task is Preempted
OSTaskDel()

Kernels

- The kernel is a part of a multitasking system, it is responsible for:
  - The management of tasks.
  - Inter-task communication.

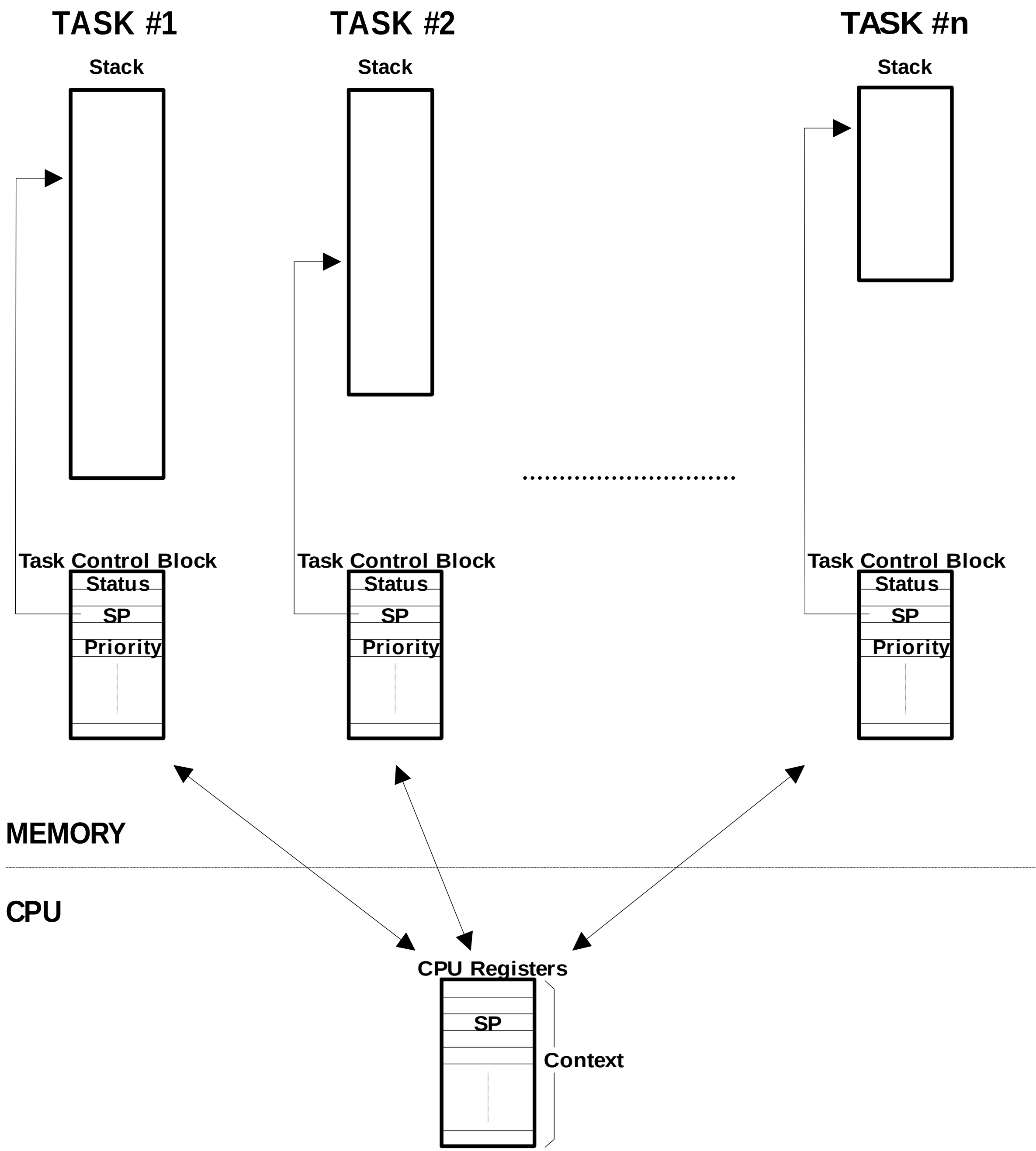
- The kernel imposes additional overheads to task execution.
  - Kernel services take time.
    - Semaphores, message queues, mailboxes, timing controls, and etc…
  - ROM and RAM space are needed.

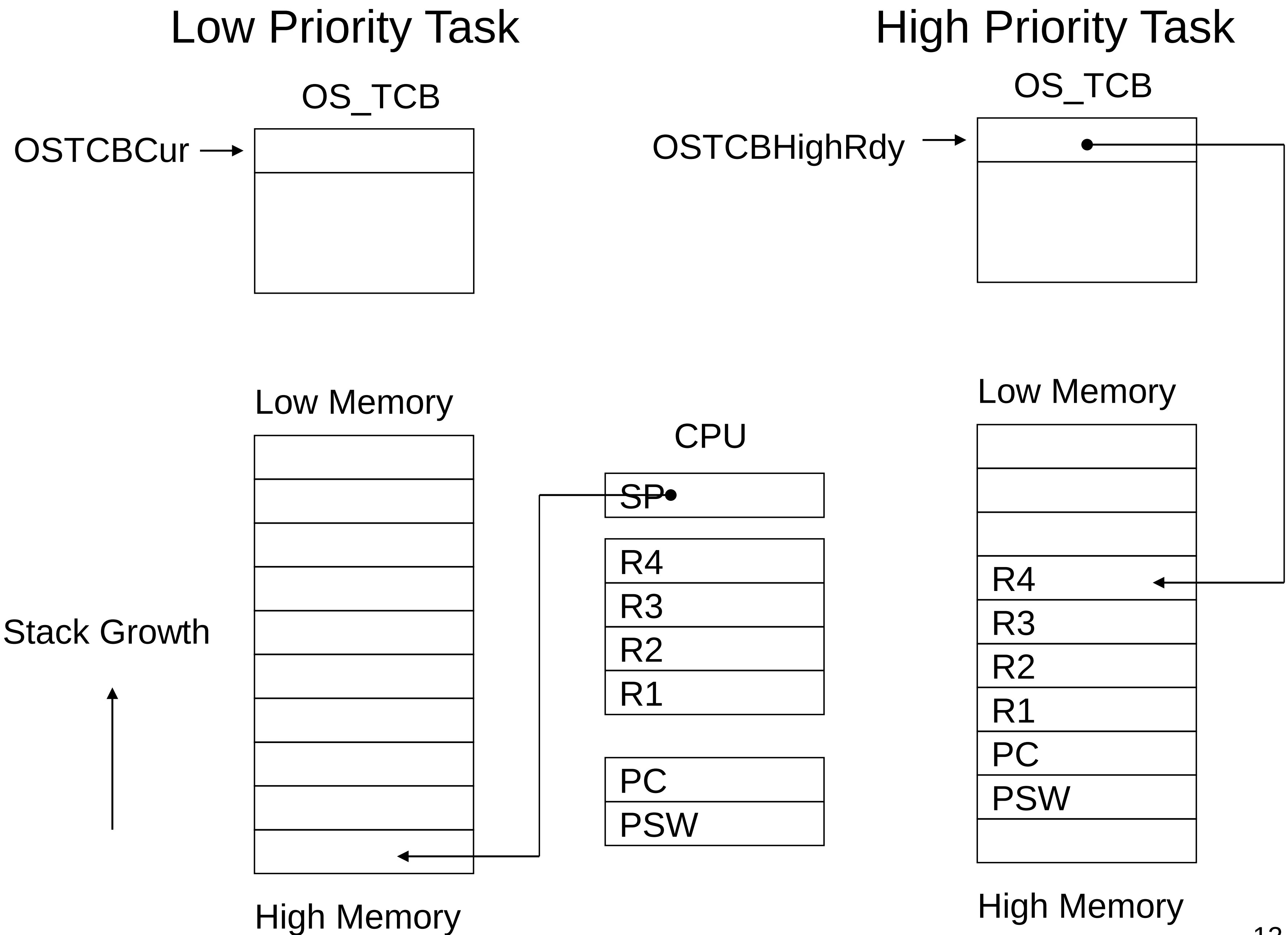
Context Switch

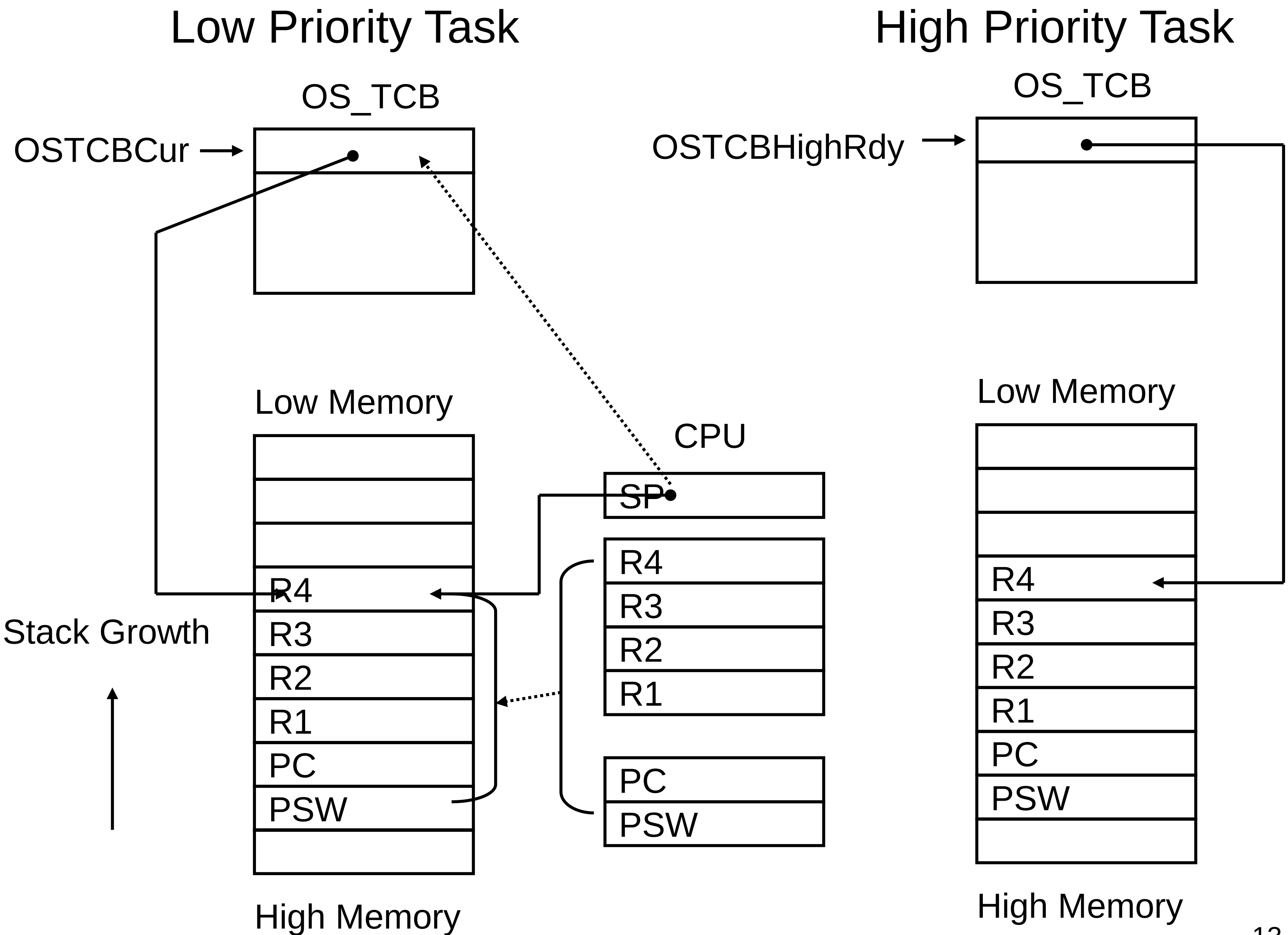
- It occurs when the scheduler decides to run a different task.
- The scheduler must save the context of the current task and then load the context of the task-to-run.
  – The context is of a priority, the contents of the registers, the pointers to its stack, and the related housekeeping data.

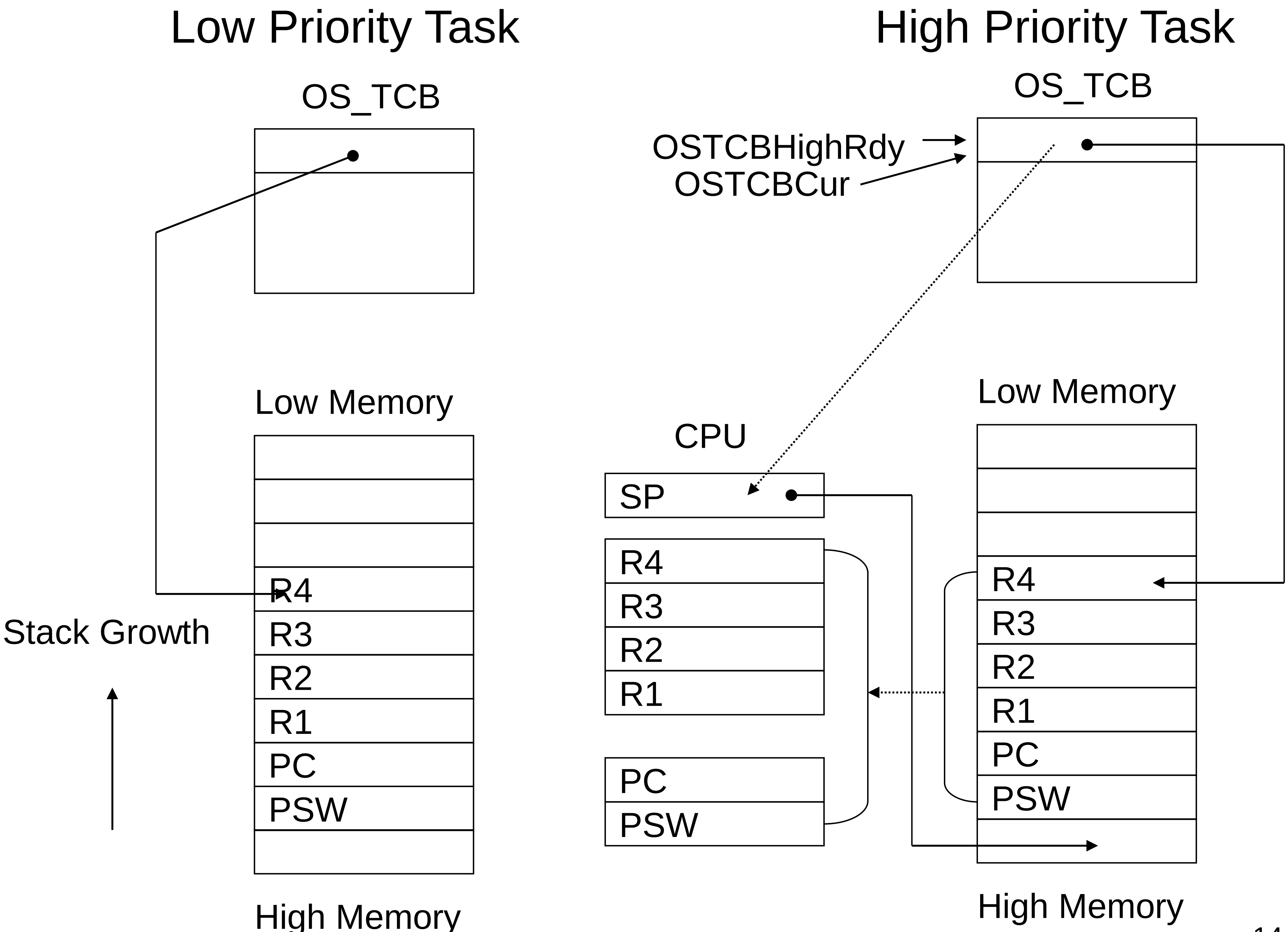
Context Switch

- Context-switches impose overheads on the task executions.
  - A practicable scheduler must not cause intensive context switches. Because modern CPU's have deep pipelines and many registers.
- For a real-time operating system, we must know how much time it takes to perform a context switch.
  - The overheads of context switch are accounted into high priority tasks. (blocking time, context switch time…)


TASK #1
TASK #2
TASK #n
Stack
Stack
Stack
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
Task Control Block
Status
SP
Priority
MEMORY
CPU
CPU Registers
SP
Context


Low Priority Task
High Priority Task
OS_TCB
OSTCBCur
OS_TCB
OSTCBHighRdy
Low Memory
CPU
SP
R4
R3
R2
R1
PC
PSW
Low Memory
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
High Memory


Low Priority Task
High Priority Task
OS_TCB
OSTCBCur
OS_TCB
OSTCBHighRdy
Low Memory
CPU
Low Memory
SP
R4
R3
R2
R1
PC
PSW
R4
R3
R2
R1
PC
PSW
R4
R3
R2
R1
PC
PSW
Stack Growth
High Memory
High Memory


Low Priority Task
High Priority Task
OS_TCB
OS_TCB
OSTCBHighRdy
OSTCBCur
Low Memory
CPU
Low Memory
SP
R4
R3
R2
R1
PC
PSW
Stack Growth
R4
R3
R2
R1
PC
PSW
R4
R3
R2
R1
PC
PSW
High Memory
High Memory

Non-Preemptive Kernels

- Context switches occur only when tasks explicitly give up control of the CPU.
  - High-priority tasks gain control of the CPU.
  - This procedure must be done frequently to improve the responsiveness.

- Events are still handled in ISR's.
  - ISR's always return to the interrupted task.

Non-Preemptive Kernels

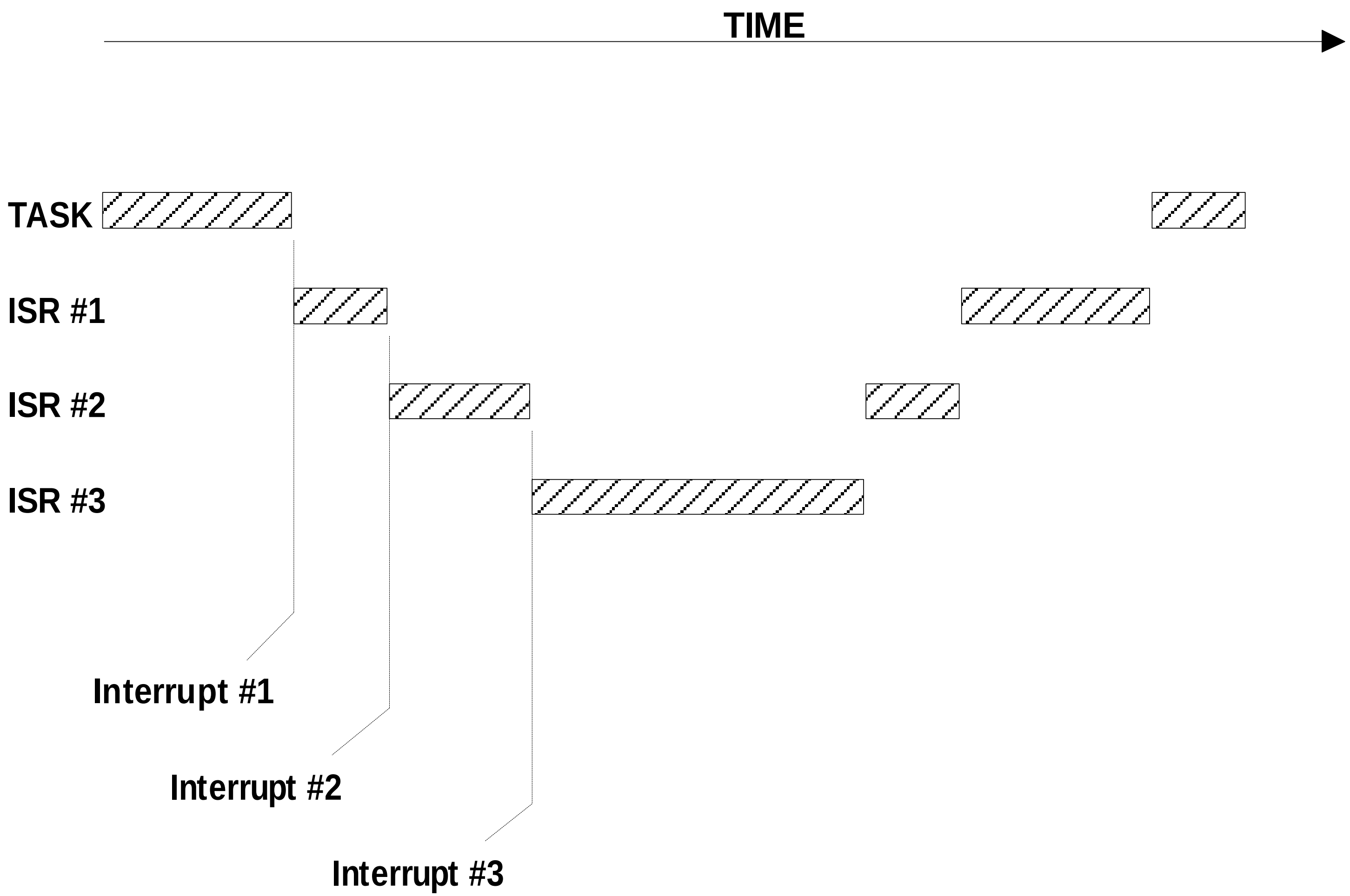
- Most tasks are race-condition free.
  - Non-reentrant codes can be used without protections.
  - In some cases, synchronizations are still needed.
- Pros: simple, robust.
- Cons: Not very responsive. There might be lengthy priority inversions.

Preemptive Kernels

- The benefit of a preemptive kernel is the system is more responsive.
  - The execution of a task is deterministic.
  - A high-priority task gain control of the CPU instantly when it is ready.
- ISR might not return to the interrupted task.
  - It might return a high-priority task which is ready.
- Concurrency among tasks exists. As a result, synchronization mechanisms (semaphores…) must be adopted to prevent from corrupting shared resources.
  - Preemptions, blocking, priority inversions.

Interrupts

- A hardware event to inform the CPU of an asynchronous event
  - clock tick (triggering scheduling), I/O events, hardware errors.
- The context of the current task is saved and the corresponding interrupt service routine (ISR) is invoked
- The ISR processes the event, and upon completion of the ISR, the program returns to
  - The background for a foreground/background system
  - The interrupted task for a non-preemptive kernel
  - The highest priority task ready to run for a preemptive kernel


TIME
TASK
ISR #1
ISR #2
ISR #3
Interrupt #1
Interrupt #2
Interrupt #3

Multiple Interrupts

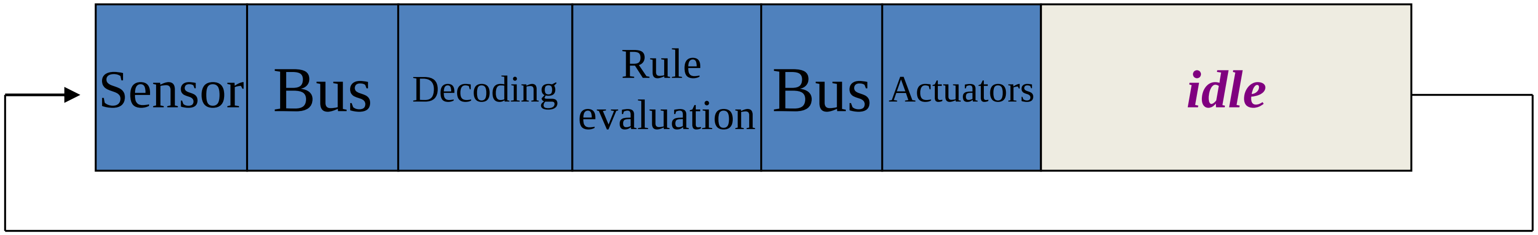
- A vector table manages devices to be handled by different code.
- An interrupt with higher priority is executed first

ISR

- ISRs should be as short as possible
- ISR should
  - Recognize the interrupt
  - Get status from the interrupting device
  - Signal a task to perform processing
- Overhead involved in signaling task

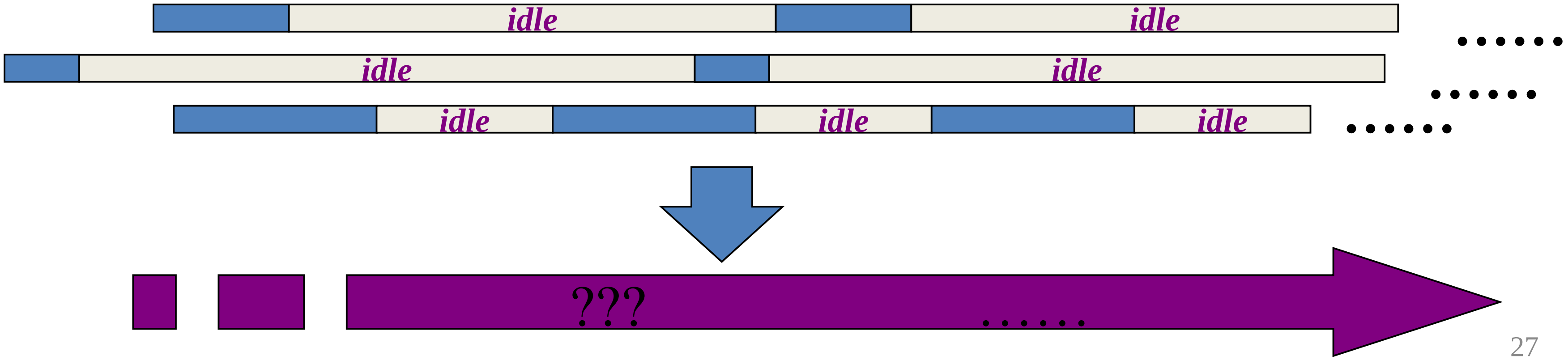
Cyclic Executive

- The system repeatedly exercises a static schedule
  - A table-driven approach
- Many existing systems still take this approach
  - Easy to debug and easy to visualize
    - Highly deterministic
  - Hard to program, to modify, and to upgrade
    - A program should be divided into many pieces (like an FSM)



Cyclic Executive

- The table emulates an infinite loop of routines
  - However, a single independent loop is not enough to many complicated systems
  - Multiple concurrent loops should be considered
- How large should the table be when there are multiple loops?



Cyclic Executive

- ***Definition***: Let the hyper-period of a collection of loops be a time interval which's length is the least-common-multiplier of the loops' lengths
  - Let the length of the hyper-period be abbreviated as "h"

- ***Theorem***: The number of routines to be executed in any time interval [t,t+x] is identical to that in [t+h,t+h+x]

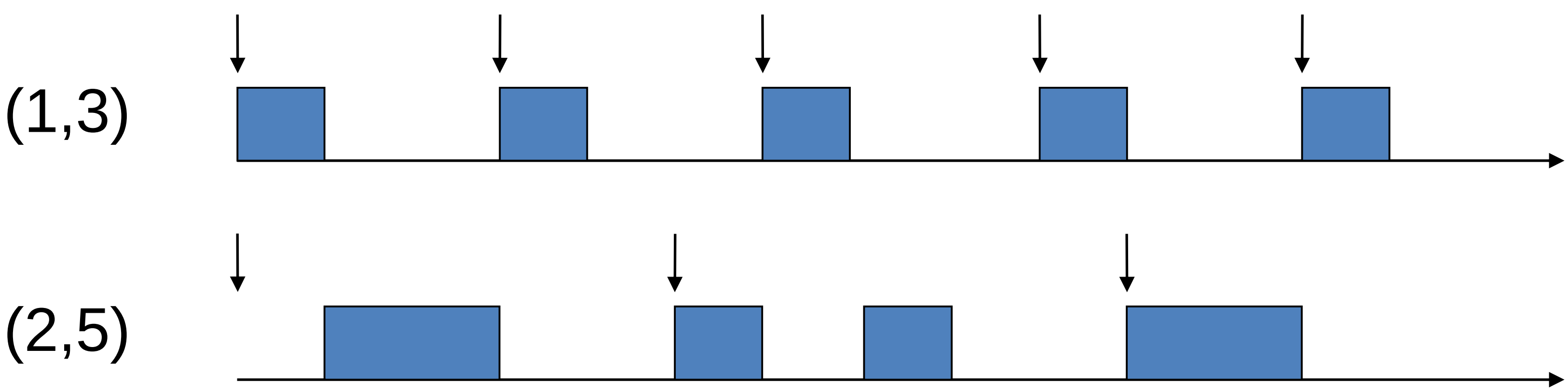
Round-Robin Scheduling

- A quantum is a pre-determined value

- Context switch occurs when:
  - The current task completes
  - The quantum for the current task is reached

Rate-Monotonic Scheduling

- Task-level fixed-priority scheduling
  - All jobs inherit its task's priority
  - Usually abbreviated as fixed-priority scheduling
- Tasks' priorities are inversely proportional to their period lengths

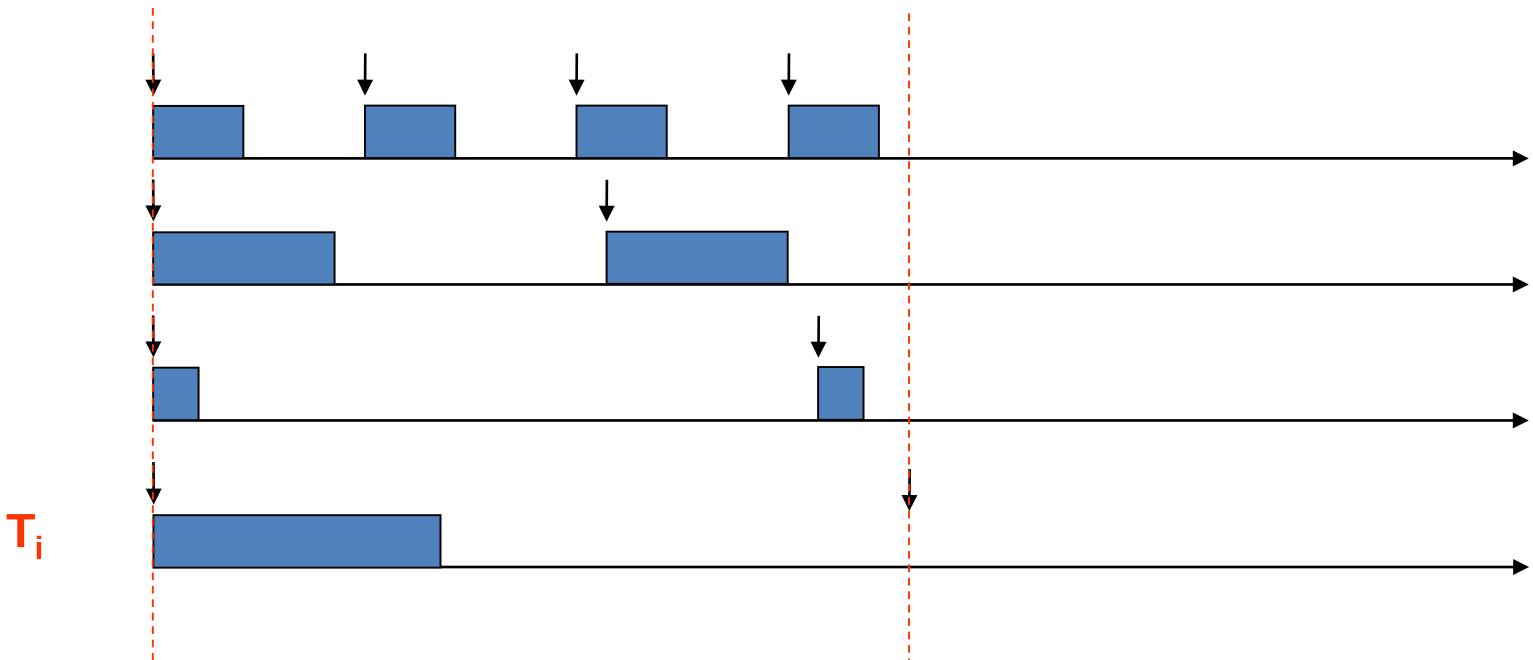


Rate-Monotonic Scheduling

- Critical instant (critical instance) of task T_i
  - A job $J_{i,c}$ of task T_i released at T_i's critical instant would have the longest response time
  - $J_{i,c}$ would be the one that is "hardest" to meet its deadline
  - If $J_{i,c}$ succeeds in satisfying its deadline, then any job of T_i always succeeds for any cases
    - Since in any other cases deadlines are easier to meet

Rate-Monotonic-Scheduling

- ***Theorem***: A critical instant of any task T_i occurs when one of its job $J_{i,c}$ is released at the same time with a job of every higher-priority task (i.e., in-phase).

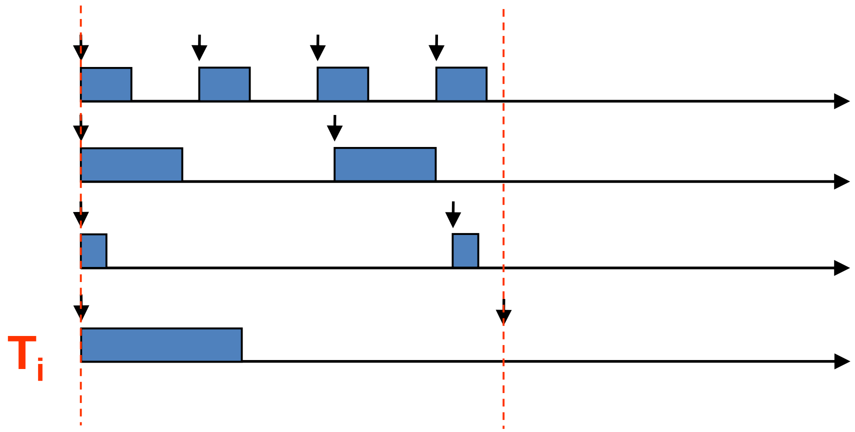


Rate-Monotonic Scheduling

- Response time analysis
  - The response time of the job of Ti at critical instant can be calculated by the following recursive function

$$r_0 = \sum_{\forall i} c_i$$

$$r_n = \sum_{\forall i} c_i \left\lceil \frac{r_{n-1}}{p_i} \right\rceil$$



  - Observation: the sequence of r_x, x>=0 may or may not converge

Rate-Monotonic Scheduling

- ***Theorem***: Given a task set={T_1,T_2,…,T_n}, if at critical instant the response time of the first job of task T_i, for each i, converges no later than p_i, then jobs never miss their deadlines

- Observations
  - If the task set survives critical instant, then it will survive any task phasing
  - The analysis is an exact schedulability test for RMS
  - Usually referred to as "Rate-Monotonic Analysis", RMA for short

Rate-Monotonic Scheduling

- Definition
  - Utilization factor of task T=(c,p) is defined as

$$\frac{c}{p}$$

  - CPU utilization of a task set $\{T_1, T_2, \ldots, T_n\}$ is

$$U = \sum_{i=1}^{n} \frac{c_i}{p_i}$$

  - Observation: if the total utilization exceeds 1 then the task set is not schedulable

Rate-Monotonic Scheduling

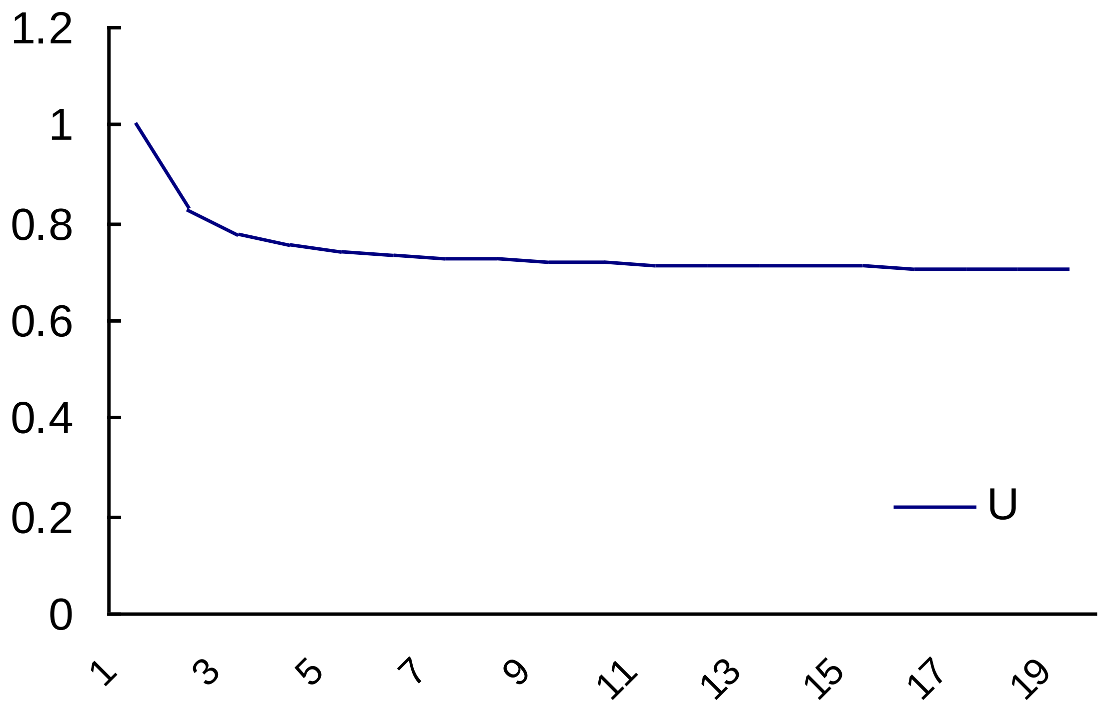
- ***Theorem***: [LL73] Given a task set $\{T_1, T_2, \ldots, T_n\}$. It is schedulable by RMS if

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq U(n) = n(2^{1/n} - 1)$$

- Observation:
  - If the test succeeds then the task is schedulable
  - A sufficient condition for schedulability

Rate-Monotonic Scheduling

- When x → infinitely large, U(x)→0.68



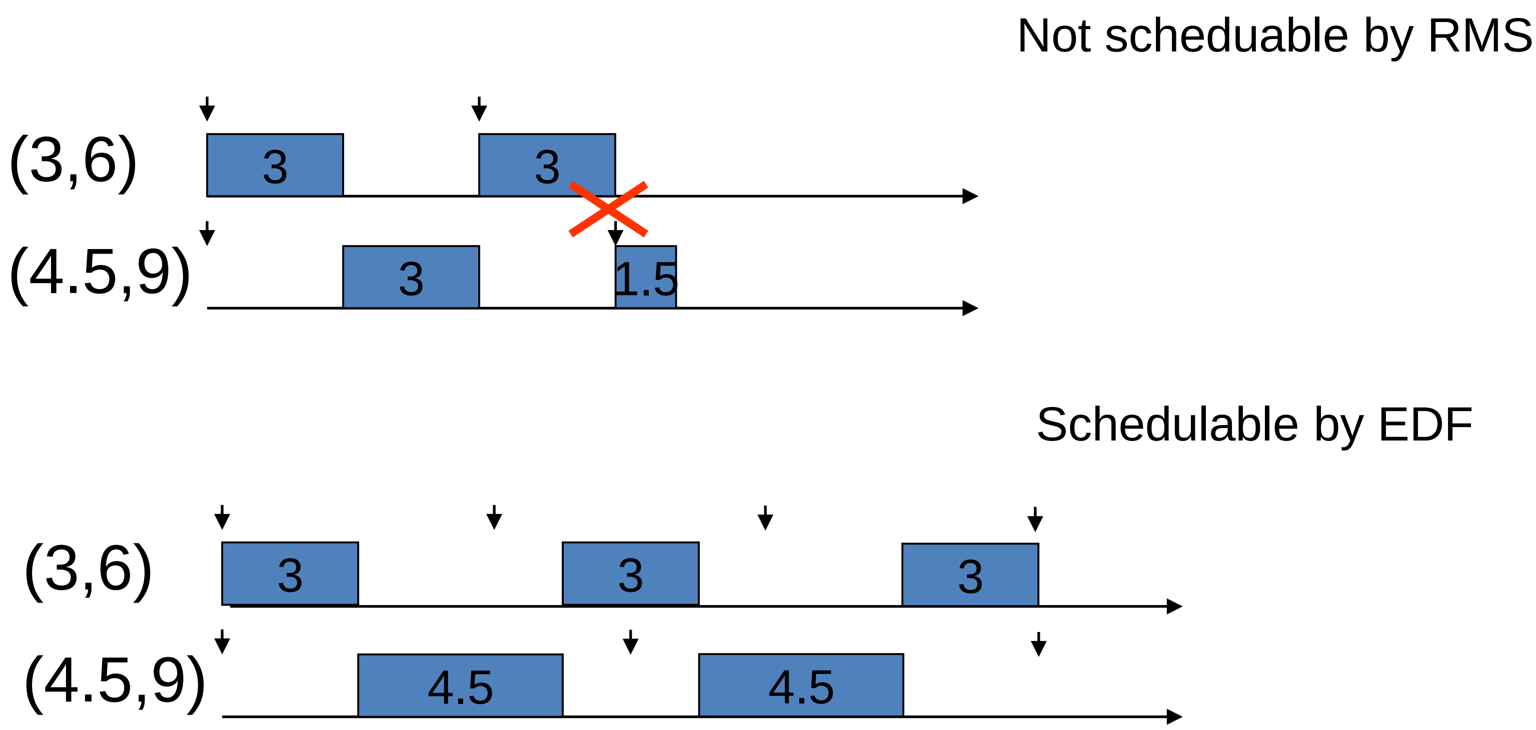
1	1
2	0.828427
3	0.779763
4	0.756828
5	0.743492
6	0.734772
7	0.728627
8	0.724062
9	0.720538
10	0.717735
11	0.715452
12	0.713557
13	0.711959
14	0.710593
15	0.709412

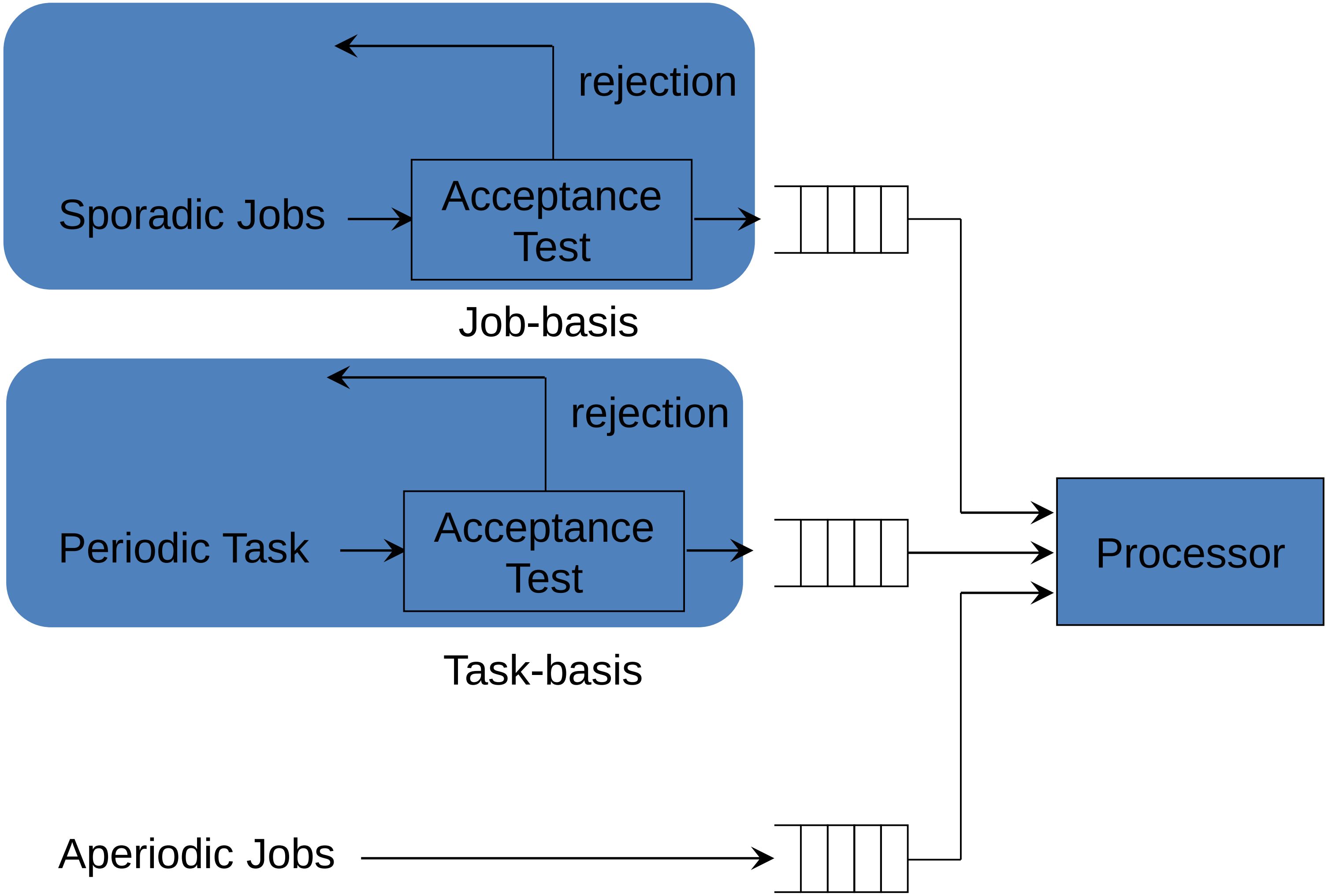
Earliest-Deadline-First Scheduling

- Definition
  - Feasible
    - A set of tasks is said to be feasible if there is some way to schedule the tasks without any deadline violations
  - Schedulable
    - Given a scheduling algorithm A
    - A set of tasks is said to be schedulable if algorithm A successfully schedule the tasks without any deadline violations
- Observations
  - A feasible task set may not be schedulable by RMS
  - If a task set is schedulable by some algorithm A, then it is feasible

Earliest-Deadline-First Scheduling

- Example




rejection
Sporadic Jobs
Acceptance Test
Job-basis
rejection
Periodic Task
Acceptance Test
Task-basis
Processor
Aperiodic Jobs

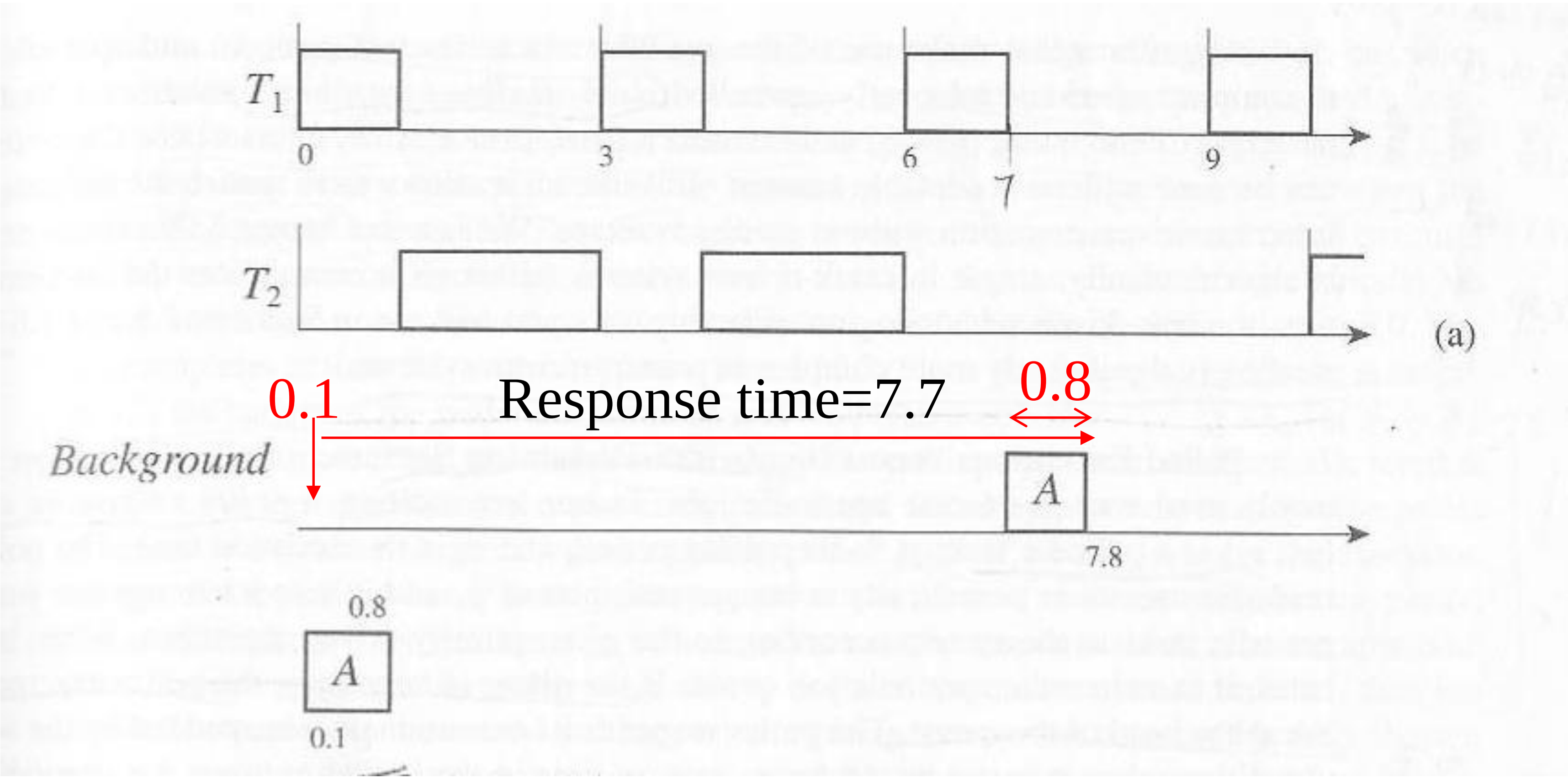
Handling Aperiodic Jobs

- Approaches
  - Background execution
    - Improvement: slack stealing
  - Interrupt-driven execution
    - Improvement: slack stealing
  - Polled execution
    - Improvement: Bandwidth-preserving servers

Handling Aperiodic Jobs

- Background execution
  - Handle aperiodic jobs whenever there is no periodic jobs to execute
    - Extremely simple
    - Always produce correct schedule
    - Poor response time

Background Execution

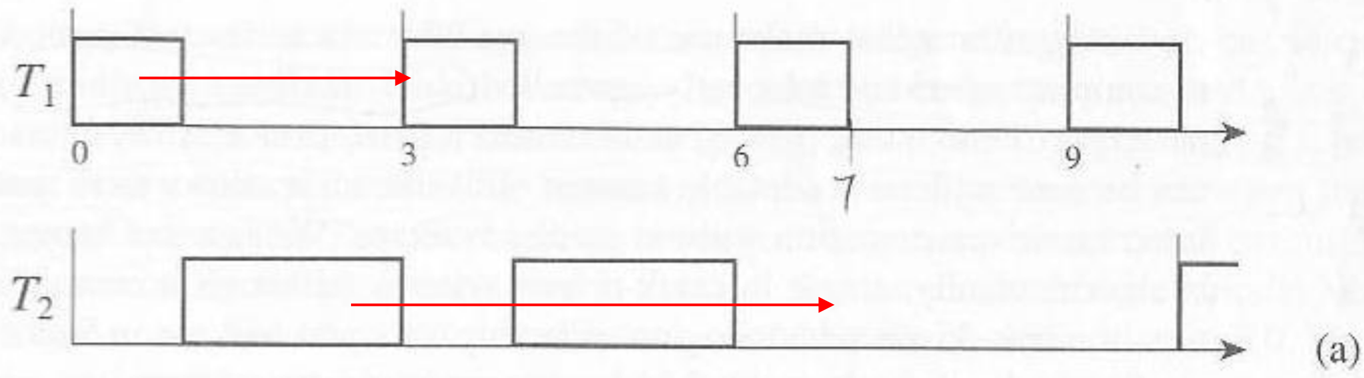


Handling Aperiodic Jobs

- Interrupt execution
  - An obvious extension to background execution
  - On arrivals, aperiodic jobs immediately interrupts the execution of any periodic jobs
  - Fastest response time
  - Potentially damage the schedulability of periodic jobs

Interrupt Execution

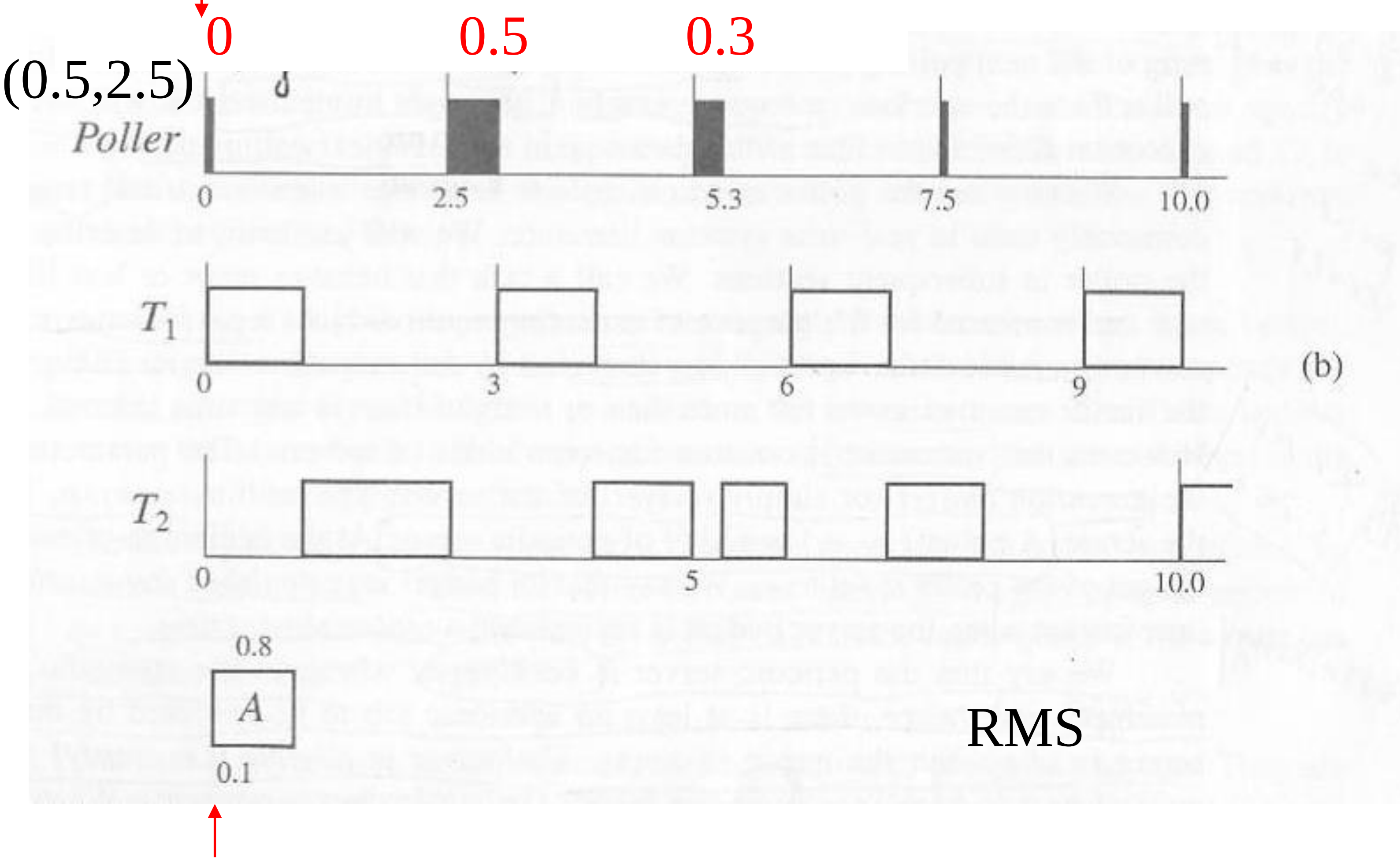
Execution time=2.1



Handling Aperiodic Jobs

- Polled execution
  - A purely periodic task (polling server) to serve a queue of aperiodic jobs
  - When the polling server gains control of the CPU, it services aperiodic jobs in the queue
  - If the queue becomes empty, the polling server suspends immediately
    - The queue is not checked until the next polling period

The polling server drops all its budget at time 0 since there is no ready aperiodic jobs



Aperiodic job A arrives at time 0.1

Resources

- An entity used by a task.
  - Memory objects
    - Such as tables, global variables …
  - I/O devices.
    - Such as disks, communication transceivers.
- A task must gain exclusive access to a shared resource to prevent data (or I/O status) from being corrupted.
  - Mutual exclusion.

Critical Sections

- A portion of code must be indivisible
  – To protect shared resources from being corrupted due to race conditions
  – Could be implemented by using interrupt enable/disable or IPC mechanisms
    - Semaphores, events, mailboxes, etc

Reentrant Functions

- Reentrant functions can be invoked simultaneously without corrupting any data.
  - Reentrant functions use either local variables (on stacks) or synchronization mechanisms (such as semaphores).

```
void strcpy(char *dest, char *src)
  {
      while (*dest++ = *src++) {
          ;
      }
      *dest = NUL;
  }
```

Non-Reentrant Functions

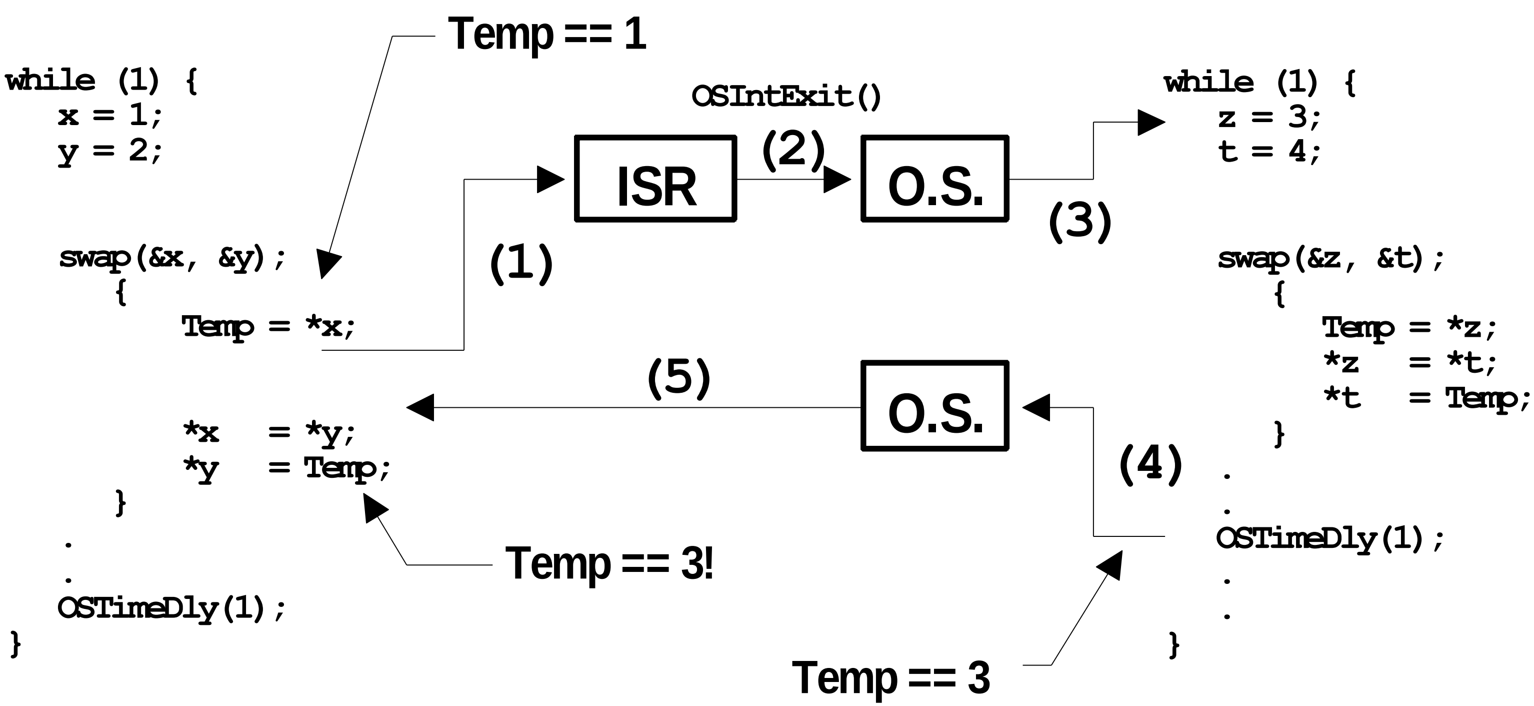
- Non-Reentrant functions might corrupt shared resources under race conditions.

```
int Temp;

  void swap(int *x, int *y)
  {
      Temp = *x;
      *x   = *y;
      *y   = Temp;
  }
```

LOW PRIORITY TASK

HIGH PRIORITY TASK



(1) When swap() is interrupted, TEMP contains 1.

(2) The ISR makes the higher priority task ready to run, so at the completion of the ISR, the kernel is invoked to switch to this task. The high priority task sets TEMP to 3 and swaps the contents of its variables correctly. (i.e., z=4 and t=3).

(3) The high priority task eventually relinquishes control to the low priority task be calling a kernel service to delay itself for one clock tick.

(4) The lower priority task is thus resumed. Note that at this point, TEMP is still set to 3! When the low priority task resumes execution, the task sets y to 3 instead of 1.

Non-Reentrant Functions

- There are several ways to make the code reentrant:
  - Declare TEMP as a local variable.
  - Disable interrupts and then enable interrupts.
  - Use a semaphore.

Mutual Exclusion

- Mutual exclusion must be adopted to protect shared resources.
  - Global variables, linked lists, pointers, buffers, and ring buffers.
  - I/O devices.

- When a task is using a resource, the other tasks which are also interested in the resource must not be scheduled to run.

- Common techniques used are:
  - disable/enable interrupts
  - performing a test-and-set instruction
  - disabling scheduling
  - using synchronization mechanisms (such as semaphores).

Mutual Exclusion

- Disabling/enabling interrupts:
  - OS_ENTER_CRITICAL() and OS_EXIT_CRITICAL()
  - All events are masked since interrupts are disabled.
  - Tasks which do not affect the resources-to-protect are also postponed.
  - Must not disable interrupt before calling system services.

```
void Function (void)
{
    OS_ENTER_CRITICAL();
    .
    .     /* You can access shared data in here */
    .
    OS_EXIT_CRITICAL();
}
```

Mutual Exclusion

- Disabling/Enabling Scheduling:
  - No preemptions could happen while the scheduler is disabled.
  - However, interrupts still happen.
    - ISR's could still corrupt shared data.
    - Once an ISR is done, the interrupted task is always resumed even there are high priority tasks ready.
  - Rescheduling might happen right after the scheduler is re-enabled.
  - Higher overheads!

```
void Function (void)
{
    OSSchedLock();
    .     /* You can access shared data
    .        in here (interrupts are recognized) */
    OSSchedUnlock();
}
```

Mutual Exclusion

- Semaphores:
  - Provided by the kernel.
  - Semaphores are used to:
    - Control access to a shared resource.
    - Signal the occurrence of an event.
    - Allow tasks to synchronize their activities.
  - Higher priority tasks which does not interested in the protected resources can still be scheduled to run.

```
OS_EVENT *SharedDataSem;
void Function (void)
{
    INT8U err;
    OSSemPend(SharedDataSem, 0, &err);
    .     /* You can access shared data
    .        in here (interrupts are recognized) */
    OSSemPost(SharedDataSem);
}
```

Semaphores

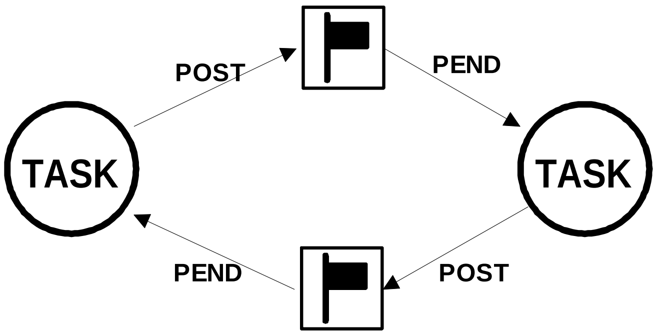
- Semaphores:
  - Three kinds of semaphores:
    - Counting semaphore (init >1)
    - Binary semaphore (init = 1)
    - Rendezvous semaphore (init = 0)
  - On event posting, a waiting task is released from the waiting queue.
    - The highest-priority task.
    - FIFO (not supported by μC/OS-II)
  - Interrupts and scheduling are still enabled under the use of semaphores.

Synchronization

- Two semaphores could be used to rendezvous two tasks.
  - It can not be used to synchronize between ISR's and tasks.
  - For example, a kernel-mode thread could synchronize with a user-mode worker thread which performs complicated jobs.

Synchronization

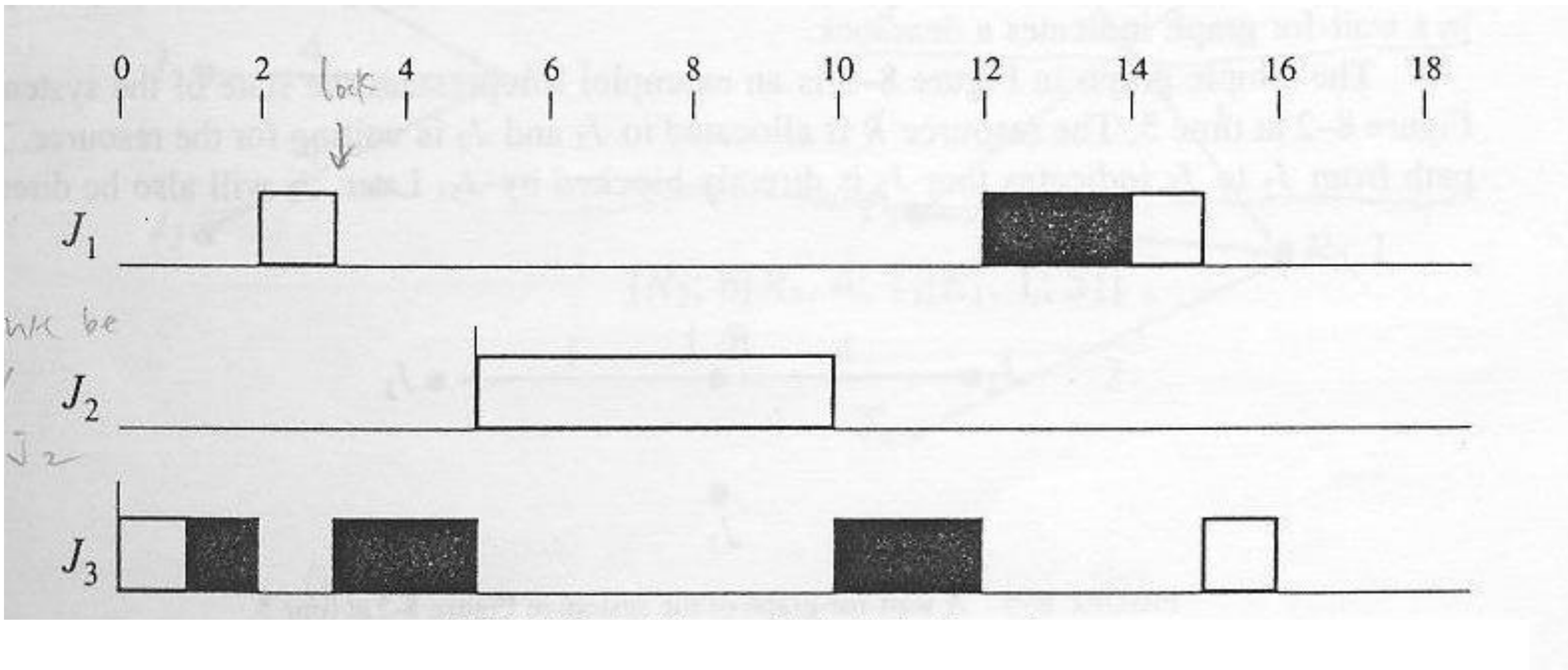
```
Task1()
{
    for (;;) {
        Perform operation;
        Signal task #2;                  (1)
        Wait for signal from task #2;    (2)
        Continue operation;
    }
}
```



** Semaphores are both initialized to 0

```
Task2()
{
    for (;;) {
        Perform operation;
        Signal task #1;                  (3)
        Wait for signal from task #1;    (4)
        Continue operation;
    }
}
```

Priority inversion



Deadlocks

- Tasks circularly wait for certain resources which are already locked by another tasks.
  - No task could finish executing under such a circumstance.
- Deadlocks in static systems can be detected and resolved in advance.
- Deadlocks are not easy to detect and resolve in a on-line fashion.
  - A brute-force way to avoid deadlocks is to set a timeout when acquiring a semaphore.
  - The elegant way is to adopt resource synchronization protocols.
    - Priority Ceiling Protocol (PCP), Stack Resource Policy (SRP)

Summary

- Scheduling algorithms
- Resource management