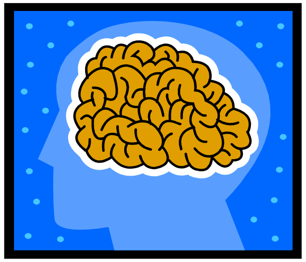
Operating System Practice

Che-Wei Chang

chewei@mail.cgu.edu.tw

Department of Computer Science and Information Engineering, Chang Gung University

Course Roadmap



Advanced Operating System Concepts

- Concepts and Implementation of File System
- Storage Management and I/O Devices
- System Protection and Security

Exercises on PC and Emulators

- Concepts of the Linux Kernel
- Real-Time System Knowledge
- Android Programing on Android Emulator

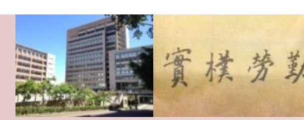
Embedded System Exercises

- Introduction to Embedded System
- Tools and Techniques to Build Embedded Systems
- Implementation on Embedded System Evaluation Boards

Introduction to Linux

Advantages of Linux

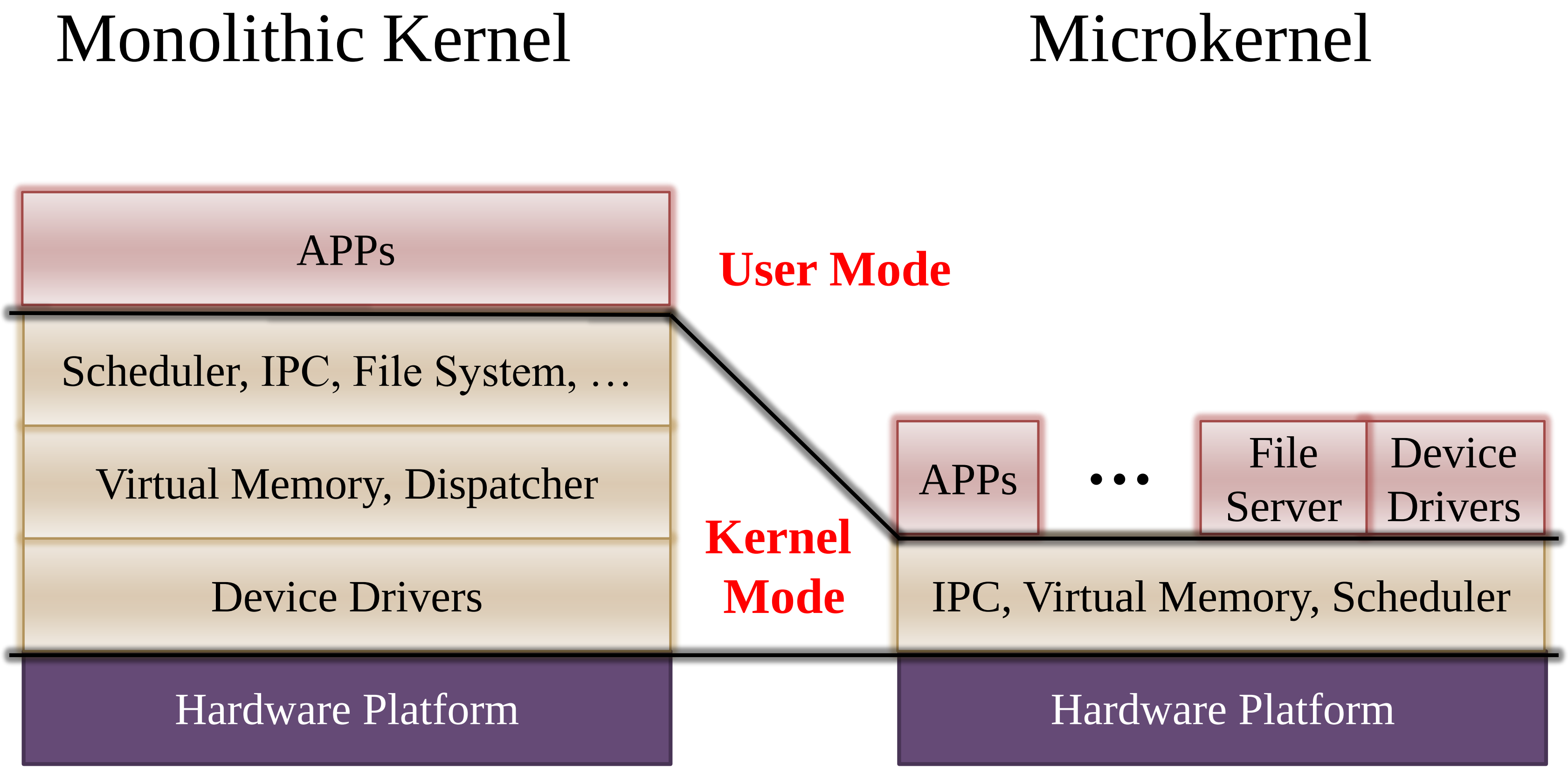
- Linux is free, both in source code and cost, due to the GPL
- Linux is fully customizable in all its components
- Linux can runs on low-end, inexpensive hardware platforms, e.g., one with 4 MB RAM
- Linux systems are stable
- The Linux kernel can be very small and compact
- Linux is highly compatible with many common applications and functions
- Linux is well-supported



Different Type of Operating System Kernels

- Monolithic kernel
  - The entire operating system is working in kernel space
  - All parts of the kernel share the same kernel-level memory
  - Kernel components might affect other components
  - The Linux kernel is an example
- Microkernel
  - Kernel functions are partitioned into components
  - Communications are via inter process communication (IPC) protocol
  - The L4 microkernel is an example

Monolithic Kernel and Microkernel



Approaches for Virtualization

- Virtual Machines on an Host OS
  - For example: VMWare Workstation, Oracle VM VirtualBox
  - Easy to use and install
- Hypervisors on a Hardware Platform
  - For example: Xen
  - High perform with a very slim software layer
- Microkernel
  - For example: OKL4 Microkernel
  - Many functions to support the routines of an OS

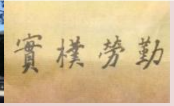
History of Linux (1/2)

- 1965: Multiplexed Information and Computing Service (Multics)
  - It is a mainframe timesharing operating system
  - It is developed by Bell Lab, MIT and GE
  - It shows the vision and concept of operating systems
- 1973: Uniplexed Information and Computing System (UNIX)
  - It has been re-written in C to be portable and quite popular
  - It became closed source in 1979

History of Linux (2/2)

- 1984: Minix
  - It is on X86 architecture
  - It is originally for education
- 1991: Linux 0.02
  - It runs on X86
  - It is open source
  - It can be compiled by gcc
  - Everyone can contribute new code to it

Hello everybody out there using minix- I'm doing a (free) operation system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones.



Features of Linux

- Monolithic kernel
  - It is large and complex
  - Most commercial Unix variants are monolithic
- Dynamically linked module
  - It is able to automatically load and unload modules on demand
- Kernel threading
  - A kernel thread is an execution context that can be independently scheduled
  - Context switches between kernel threads are usually much less expensive than context switches between ordinary processes
- Multithreaded application support
- Preemptive kernel
- Multiprocessor support
- Filesystem support

Design Principles

- Linux is a multiuser, multitasking system with a full set of UNIX-compatible tools
- Its file system adheres to traditional UNIX semantics, and it fully implements the standard UNIX networking model
- Main design goals are speed, efficiency, and standardization
- Linux is designed to be compliant with the relevant POSIX documents

Kernel Modules

- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel
- A kernel module may typically implement a device driver, a file system, or a networking protocol
- The module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL
- Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in
- Three components to Linux module support
  - module management
  - driver registration
  - conflict resolution

Module Management

- Supports loading modules into memory and letting them talk to the rest of the kernel
- Module loading is split into two separate sections:
  - Managing sections of module code in kernel memory
  - Handling symbols that modules are allowed to reference
- The module requestor manages currently unloaded modules
  - It also regularly queries the kernel to see whether a dynamically loaded module is still in use
    - Unload a module when it is no longer actively needed

Driver Registration

- Allows modules to tell the rest of the kernel that a new driver has become available
- The kernel maintains dynamic tables of all known drivers, and provides a set of routines to allow drivers to be added to or removed from these tables at any time
- Registration tables include the following items:
  - Device drivers
  - File systems
  - Network protocols
  - Binary format

Major and Minor Numbers

- Major number
  - Each device driver is identified by a unique major number
  - This number is assigned by the Linux Device Registrar
- Minor number
  - This uniquely identifies a particular instance of a device
  - If there are three devices with the same device driver, they will have the same major number but different minor numbers
- mknod [device name][bcp] [Major] [Minor]
  - b: block devices
  - c: character devices
  - p: a FIFO file

Process Management

- Linux process management separates the creation of processes and the running of a new program into two distinct operations
  - The `fork()` system call creates a new process
  - A new program is run after a call to `exec()`
- A process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
- Process properties fall into three groups:
  - Identity
  - Environment
  - Context

Process Identity

- Process ID (PID)
  - The unique identifier for the process
  - It is used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process
- Credentials
  - Each process must have an associated user ID and one or more group IDs that determine the process's rights to access system resources and files
- Namespace
  - Each process is associated with a specific view of the filesystem hierarchy

Process Environment

- The process's environment is inherited from its parent
  - The argument vector lists the command-line arguments used to invoke the running program; conventionally starts with the name of the program itself
  - The environment vector is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values
- Passing environment variables among processes and inheriting variables by a process's children are flexible
- The environment-variable mechanism provides a customization of the operating system for each process

Process Context

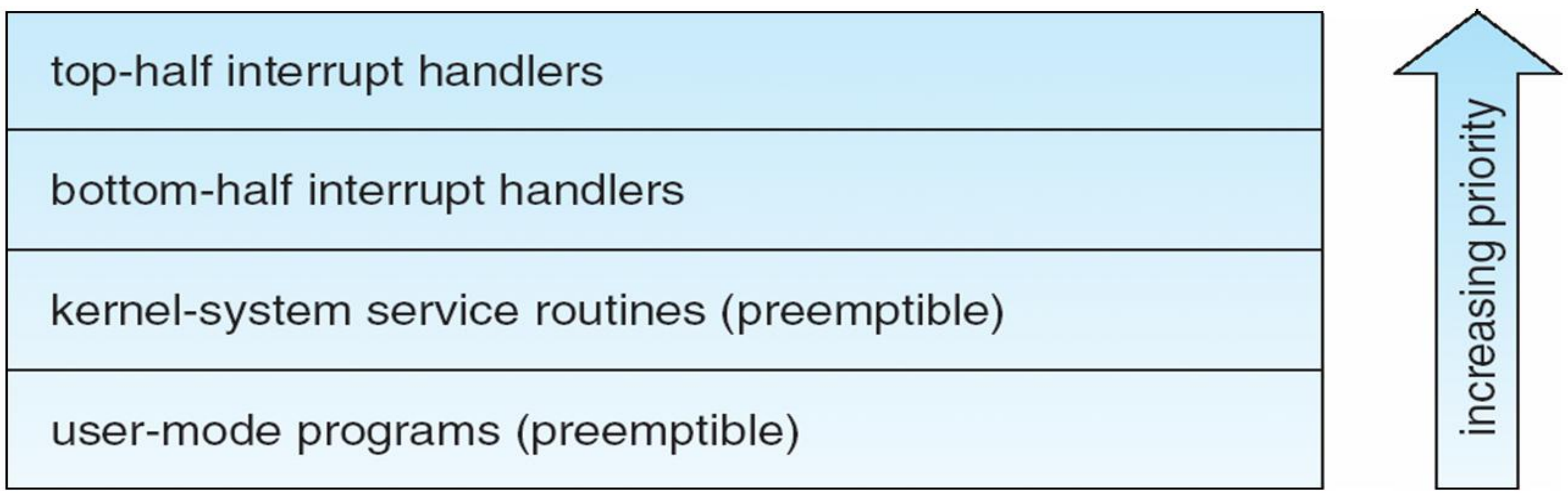
- The (constantly changing) state of a running program at any point in time
- The scheduling context is the most important part of the process context; it is the information that the scheduler needs to suspend and restart the process
- The signal-handler table defines the routine in the process's address space to be called when specific signals arrive
- The virtual-memory context of a process describes the full contents of the its private address space

Kernel Synchronization

- Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section
  - Big kernel lock
  - The kernel guarantees that it can proceed without the risk of concurrent access of shared data structures
- Interrupt service routines are separated into a *top half* and a *bottom half*
  - The top half is a normal interrupt service routine, and runs with recursive interrupts disabled
  - The bottom half runs with all interrupts enabled

Interrupt Protection Levels

- Each level may be interrupted by code running at a higher level, but will never be interrupted by code running at the same or a lower level.



Process Scheduling

- Linux uses two process-scheduling algorithms
  - A time-sharing algorithm
  - A real-time algorithm for tasks where absolute priorities are more important than fairness
- For time-sharing processes, Linux uses a prioritized, credit based algorithm
- Linux implements the FIFO and round-robin real-time scheduling classes

Executing and Loading User Programs

- Linux maintains a table of functions for loading programs
  - it gives each function the opportunity to try loading the given file when an exec system call is made
- The registration of multiple loader routines allows Linux to support both the ELF and a.out binary formats
- Initially, binary-file pages are mapped into virtual memory
  - Only when a program tries to access a given page will a page fault result in that page being loaded into physical memory
- An ELF-format binary file consists of a header followed by several page-aligned sections
  - The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory

Department of Computer Science and Information Engineering, Chang Gung University

Proc File System

- The proc file system does not store data, rather, its contents are computed on demand according to user file I/O requests
- When data is read from one of these files, proc collects the appropriate information, formats it into text form and places it into the requesting process's read buffer
- cat /proc/cpuinfo will get the CPU information
  - vendor ID
  - CPU family, CPU cores
  - cache size, TLB size
  - …

Real-Time Systems

An Example of Real-Time Designs

- A camera periodically takes a photo
- The image recognition result will be produced before the next period
- If there is an obstacle, the train automatically brakes

Time of a Period = 150/50 = 3s

Distance of a Period = (400 -100)/2 = 150m

Braking: $-12.5m/s^2$

Max Seed: 50m/s

Distance to Stop

25x(50/12.5)=100m

150m 150m

Period	Period	100m

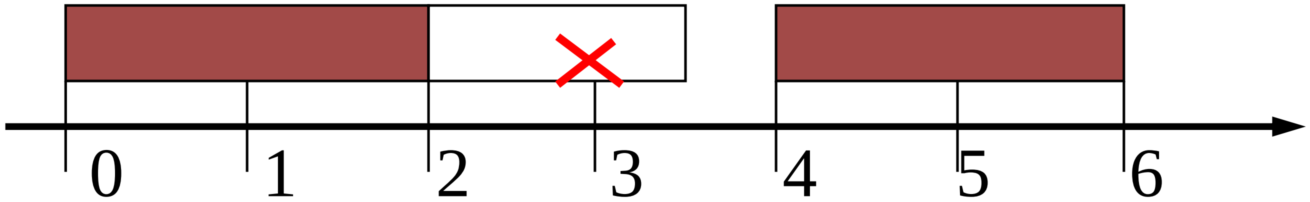
Event Detection Recognition Stop

Camera Range: 400m

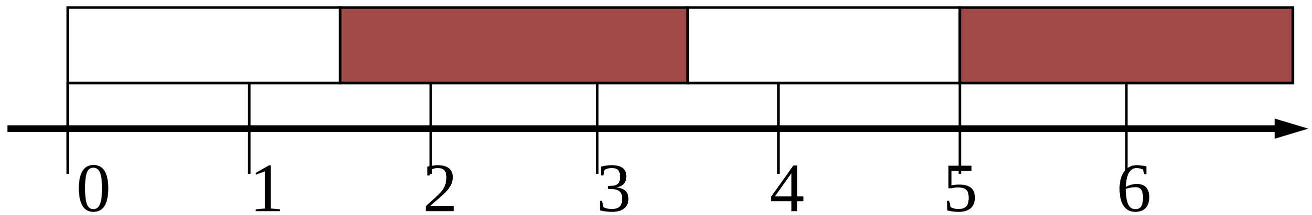
Multiple Real-Time Tasks

- Playing piano: 2 days per 4 days
  Playing chess : 1.5 days per 3 days

- Case 1: Playing piano is always more important



- Case 2: Doing whatever is more urgent



Tentative Assumptions

- Processes are independent
- Processes are all periodic
- The deadline of a request is its next request time
- A scheduler consists of a priority assignment policy and a priority-driven scheduling mechanism

Reference: C.L. Liu and James. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment,” JACM, Vol. 20, No.1, January 1973, pp. 46-61

Definitions

- The **response time** of a request for a process is the time span between the request and the end of the response to that request
- A **critical instant** of a process is an instant at which a request of that process has the longest response time
- A **critical interval** for a process is the time interval between the start of a critical instant and the deadline of the corresponding request of the process

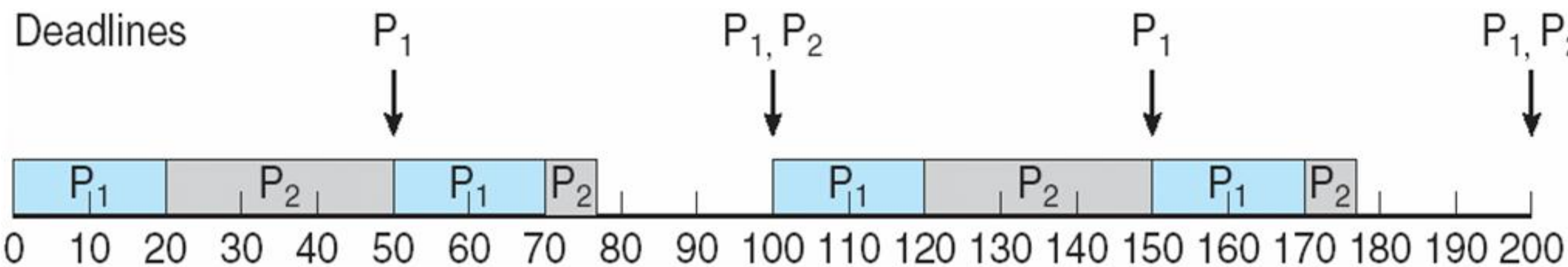
➔A critical instant for any process occurs whenever the process is requested simultaneously with requests for all higher priority processes

An observation: If a process can complete its execution within its critical interval, it is schedulable at all time!

A Static Scheduling Algorithm—Rate Monotonic Scheduling

- A static priority is assigned to each task based on the inverse of its period
  - A task with shorter period ➔ higher priority
  - A task with longer period ➔ lower priority
  - For example:
    - P_1 has its period 50 and execution time 20
    - P_2 has its period 100 and execution time 37

    ➔P_1 is assigned a higher priority than P_2



Property of Rate Monotonic Scheduling

- The **rate monotonic** (RM) priority assignment assigns processes priorities according to their request rates
  - If a feasible fixed priority assignment exists for some process set, then the rate monotonic priority assignment is feasible for that process set

→ The optimal fixed priority assignment

Proof. Exchange the priorities of two tasks if their priorities are out of RMS order.

Task A	Task B	Task C
Period: 10 Priority: 1	Period: 23 Priority: 2	Period: 15 Priority: 3

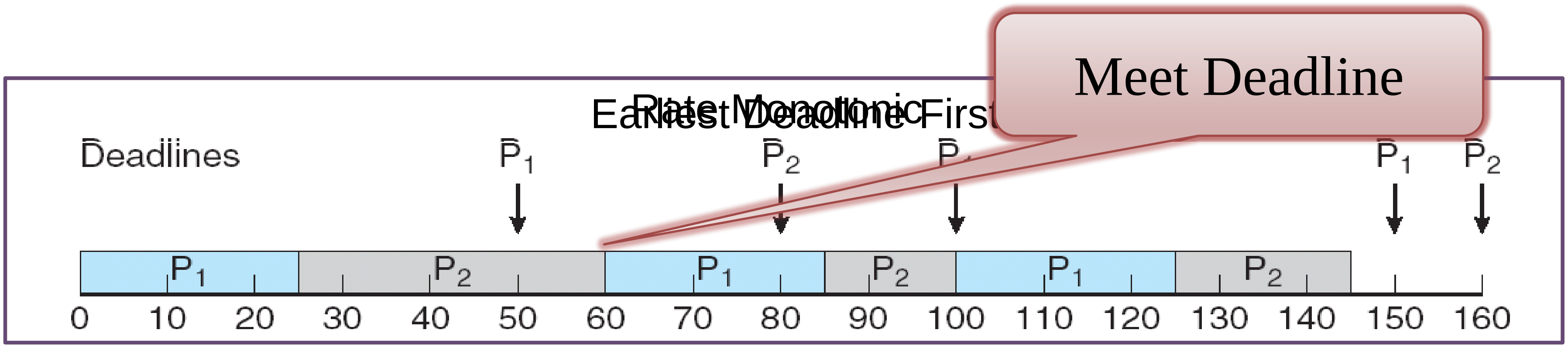
If it is feasible

Task A	Task C	Task B
Period: 10 Priority: 1	Period: 15 Priority: 2	Period: 23 Priority: 3

RM must be feasible

A Dynamic Scheduling Algorithm—Earliest Deadline First Scheduling

- Dynamic priorities are assigned according to deadlines
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
  - For example:
    - P_1 has its period 50 and execution time 25
    - P_2 has its period 80 and execution time 35



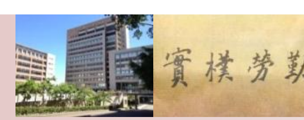
Real-Time Analysis

- For a task τ_i d with the period P_i and the execution time C_i, the utilization U_i of τ_i is defined as $U_i = \frac{C_i}{P_i}$
- For a real-time task set **T** the total utilization of the task set is $\sum_{\tau_i \in \mathbf{T}} U_i$
- If $\sum_{\tau_i \in \mathbf{T}} U_i \leq 69\%$, Rate Monotonic Scheduling can schedule all tasks in **T** to meet all deadlines
  - More precisely, for n tasks, the i-th task can meet deadline if

$$\sum_{j=1}^{i} \frac{c_j}{p_j} \leq i\left(2^{1/i} - 1\right)$$

- If and only if $\sum_{\tau_i \in \mathbf{T}} U_i \leq 100\%$, Earliest Deadline First Scheduling can schedule all tasks in **T** to meet all deadlines

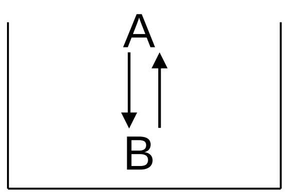
Reference: C.L. Liu and James. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," JACM, Vol. 20, No.1, January 1973, pp. 46-61



Scheduling Overheads

- **Context Switching**
  - Needed either when a process is preempted by another process, or when a process completes its execution
  - Stack Discipline

    If process A preempts process B, process A must complete before process B can resume



If it is obeyed, charge the cost of preemption (context switching cost) once to the preempting process!

Least Slack Time Algorithm

- The least slack time algorithm (LST), which assigns processes priorities inversely proportional to their slack times is also optimal if context switching cost can be ignored
  - The slack time of a process is d(t) - t - c(t)
    - t: current time
    - d(t): deadline
    - c(t): remaining execution time
  - An example
    - The time t = 0, two task have the same deadline 20
    - Task 1 has c(t) = 7, and task 2 has c(t) = 8

$t_1 \rightarrow 13$ $t_1 \rightarrow 11$ $t_1 \rightarrow 11$ $t_1 \rightarrow 9$

$t_2 \rightarrow 12$ $t_2 \rightarrow 12$ $t_2 \rightarrow 10$ $t_2 \rightarrow 10$

Task 2	Task 1	Task 2	Task 1	Task 2	Task 1	Task 2	

So many context switches!

Task 2	Task 1

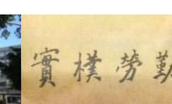
Process Synchronization

Basic Concept

- Processes might share non-preemptible resources or have precedence constraints
- Papers for discussion:
  - L. Sha, R. Rajkumar, J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Transactions on Computers, 1990.
  - A.K. Mok, "The Design of Real-Time Programming Systems Based on Process Models," IEEE Real-Time Systems Symposium, Dec 1994.

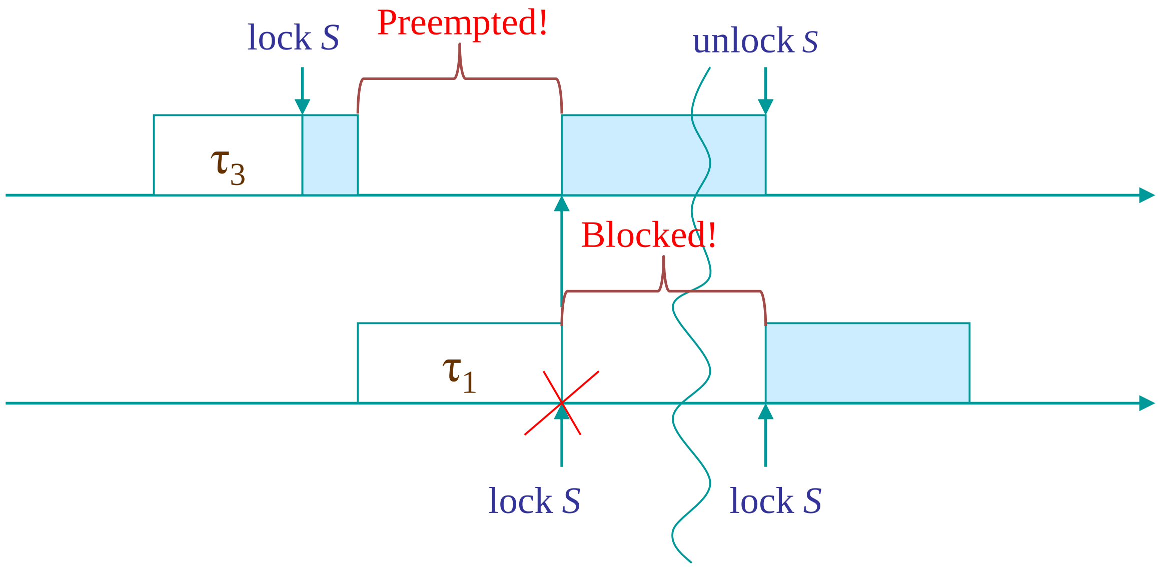
Process Synchronization

- Motivation
  - Can we find an efficient way to analyze the schedulability of a process set (systematically)
  - What kinds of restrictions on the use of communication primitives are needed so as to efficiently solve the restricted scheduling problem
  - How can we control the priority inversion problem
  - The lengths of critical sections might be quite different



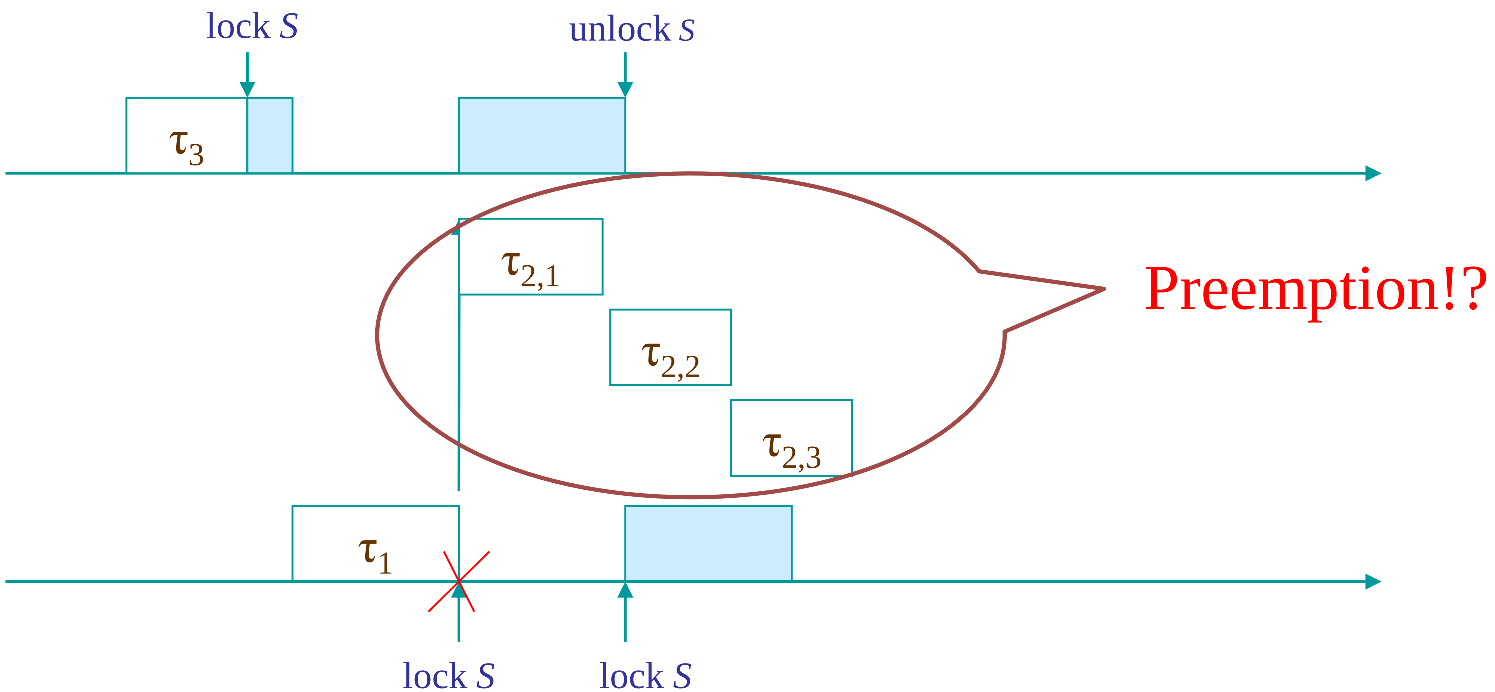
Blocking and Preemption

- Blocking: a higher-priority process is forced to wait for the execution of a lower-priority process
- Preemption: a low-priority process is forced to wait for the execution of a high-priority process



Priority Inversion

- When there are a lot of tasks having priority between that of τ_1 and τ_3, there are a lot of priority inversions

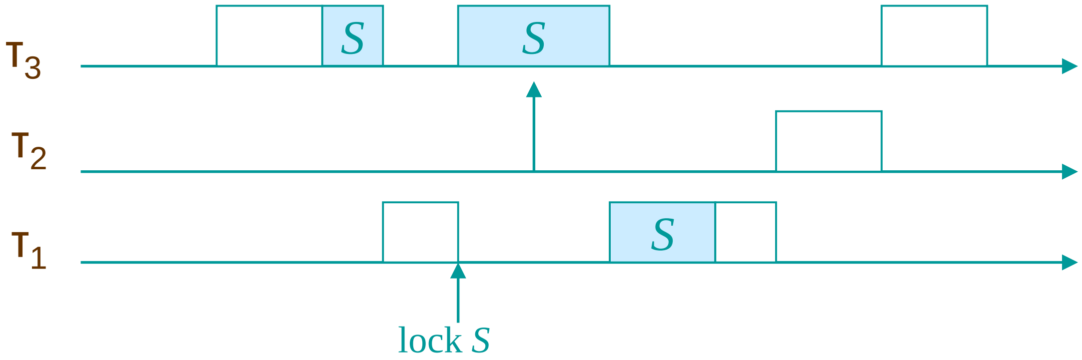


Priority Inheritance Protocol (PIP)

- Priority-Driven Scheduling
  - The process which has the highest priority among the ready processes is assigned the processor
- Synchronization
  - Process τ_i must obtain the lock on the semaphore guarding a critical section before τ_i enters the critical section
  - If τ_i obtains the required lock, τ_i enters the corresponding critical section; otherwise, τ_i is blocked and said to be blocked by the process holds the lock on the corresponding semaphore
  - Once τ_i exits a critical section, τ_i unlocks the corresponding semaphore and makes its blocked processes ready
- Priority Inheritance
  - If a process τ_i blocks higher priority processes, τ_i inherits the highest priority of the process blocked by τ_i
  - Priority inheritance is transitive

Properties of PIP

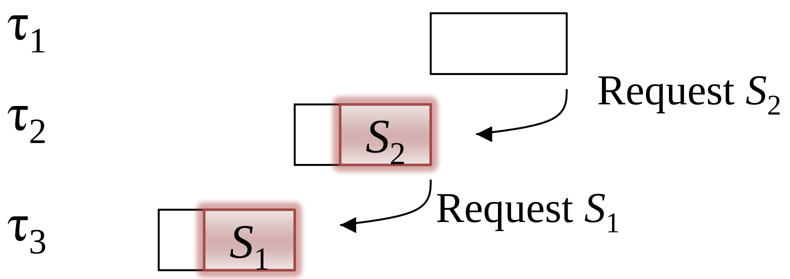
- No priority inversion



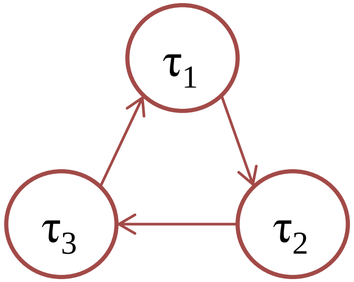
- A semaphore S can be used to cause inheritance blocking to task J only if S is accessed by a task which has a priority lower than that of J and might be accessed by a task which has a priority equal to or higher than that of J.

Concerns of PIP

- A chain of blocking is possible



- A deadlock can be formed



Priority Ceiling Protocol (PCP)

- The priority ceiling of a semaphore is the priority of the highest priority task that may lock the semaphore
- The Basic Priority Inheritance Protocol + Priority Ceiling
- A task *J* may successfully lock a semaphore S if S is available, and the priority of *J* is higher than the highest priority ceiling of all semaphores currently locked by tasks other than *J*
- Priority inheritance is transitive

Properties of PCP

- The priority ceiling protocol prevents transitive blockings
- The Priority ceiling Protocol prevents deadlock
- No job can be blocked for more than one critical section of any lower priority job
- A set of n periodic tasks under the **priority ceiling protocol** can be scheduled by the **rate monotonic algorithm** if the following conditions are satisfied:

$$\forall i, \quad 1 \le i \le n, \quad \sum_{j=1}^{i-1} \frac{c_j}{p_j} + \frac{c_i + B_i}{p_i} \le i\left(2^{1/i} - 1\right)$$

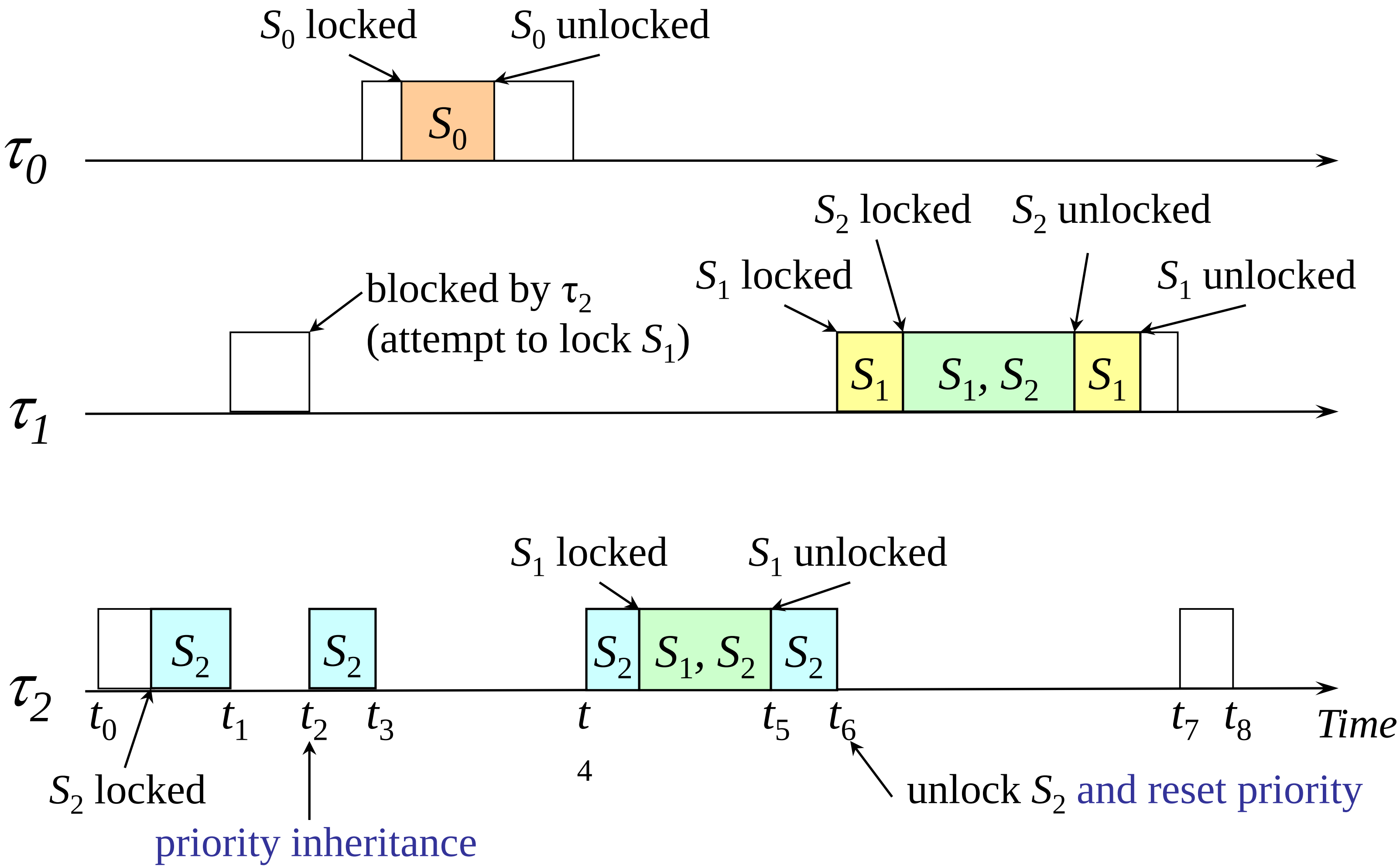
where B_i is the worst-case blocking time for τ_i

Example of PCP

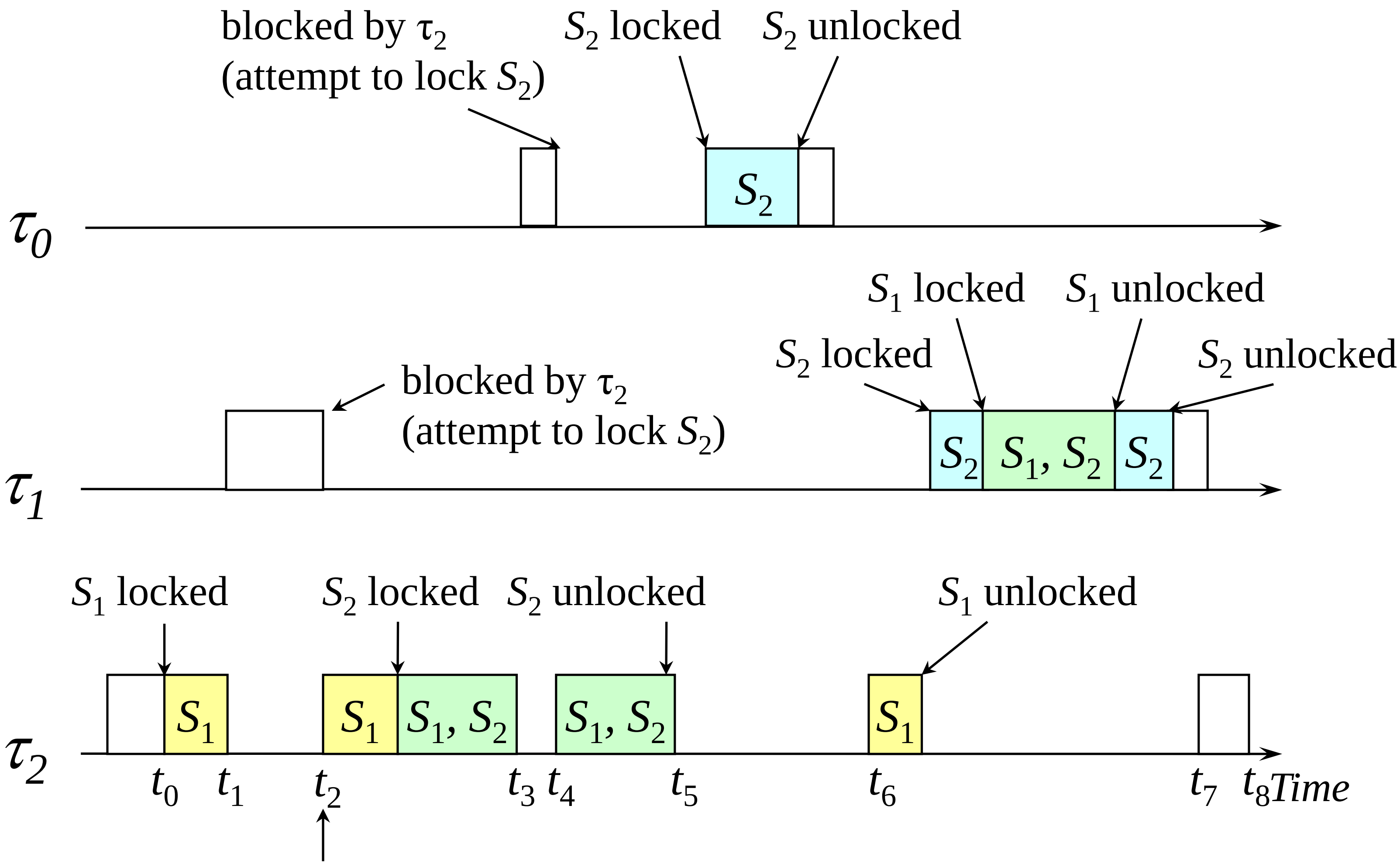
- Consider 4 tasks, t_1, t_2, t_3, and t_4 which have priorities x_1, x_2, x_3, and x_4, respectively, and assume $x_1 > x_2 > x_3 > x_4$ (x_1 is the highest priority). After we profile the programs of the 4 tasks, we have the following information:
  - Task t_1 will lock semaphore S_1 for 3ms.
  - Task t_2 will lock semaphore S_2 for 10ms and lock semaphore S_1 for 13ms.
  - Task t_3 will lock semaphore S_2 for 8ms and lock semaphore S_3 for 15ms.
  - Task t_4 will lock semaphore S_1 for 15ms and lock semaphore S_3 for 23ms.
  - Please derive the priority ceiling of each semaphore. If priority ceiling protocol is used to manage the semaphore locking, please derive the worst-case blocking time of each task.

Answer: Priority ceilings: $S_1 \rightarrow x_1$, $S_2 \rightarrow x_2$, S3 $\rightarrow x_3$.
Worst-case blocking times: $t_1 \rightarrow$ 15ms, $t_2 \rightarrow$ 15ms, $t_3 \rightarrow$ 23ms, $t_4 \rightarrow$ 0ms.

Example: Deadlock Avoidance



Example: Chain Blocking Avoidance



Avoidance blocking occurs!

Rate Monotonic Analysis

Periodic Requirements (1/2)

Task τ_1: C_1=20, P_1=100, U_1=0.2
Task τ_2: C_2=40, P_2=150, U_2=0.267
Task τ_3: C_3=100, P_3=350, U_3=0.286

- Total utilization: $75.3\% \leq 3\left(2^{\frac{1}{3}} - 1\right) = 77.9\%$
- 24.7% of the CPU is usable for lower-priority background computation

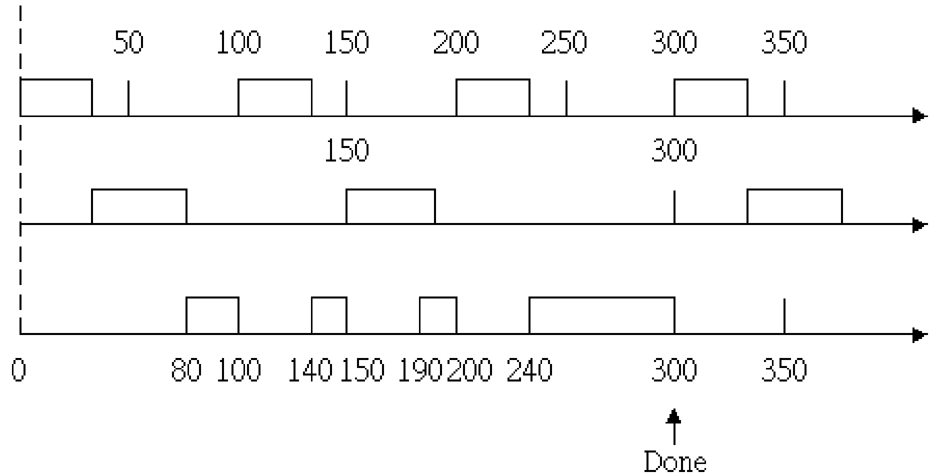
Periodic Requirements (2/2)

Task τ_1: C_1=40, P_1=100, U_1=0.4

Task τ_2: C_2=40, P_2=150, U_2=0.267

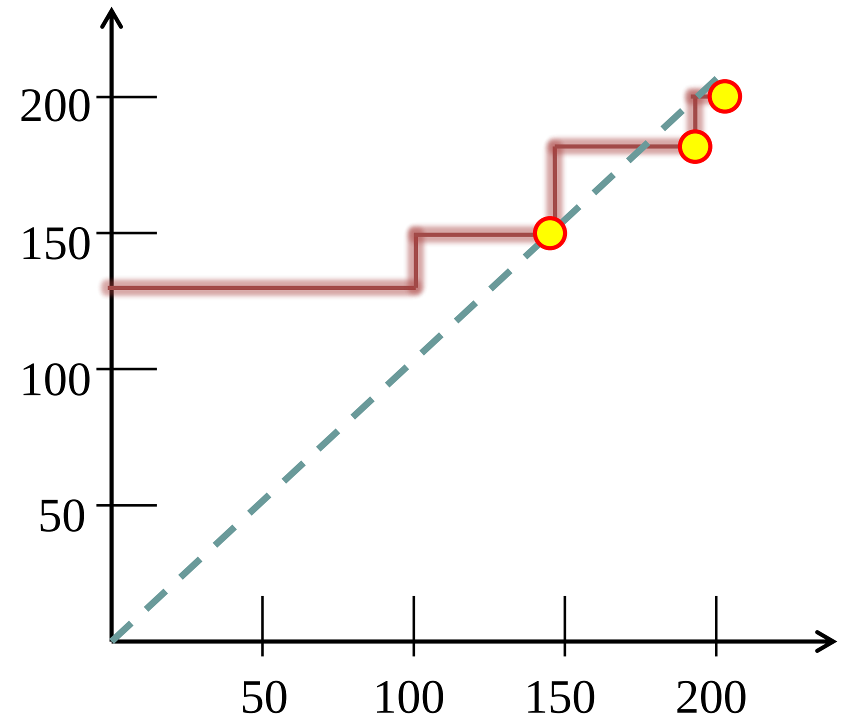
Task τ_3: C_3=100, P_3=350, U_3=0.286

- The utilization of the first two tasks: $66.7\% \le 2\left(2^{\frac{1}{2}} - 1\right) = 82.8\%$
- The total utilization: $95.3\% > 3\left(2^{\frac{1}{3}} - 1\right) = 77.9\%$



Rate Monotonic Analysis (RMA)

- A RMA Example:
  - $\tau_1(20,100)$, $\tau_2(30,150)$, $\tau_3(80, 210)$, $\tau_4(100,400)$
  - τ_1
    - $c_1 <= 100$
  - τ_2
    - $c_1 + c_2 <= 100$ or
    - $2c_1 + c_2 <= 150$
  - τ_3
    - $c_1 + c_2 + c_3 <= 100$ or
    - $2c_1 + c_2 + c_3 <= 150$ or
    - $2c_1 + 2c_2 + c_3 <= 200$ or
    - $3c_1 + 2c_2 + c_3 <= 210$
  - τ_4
    - $c_1 + c_2 + c_3 + c_4 <= 100$ or
    - $2c_1 + c_2 + c_3 + c_4 <= 150$ or
    - …



RMA with Blocking Consideration (1/2)

- A RMA Example with blocking time:
  - $\tau_1(20,100)$, $\tau_2(30,150)$, $\tau_3(80, 210)$, $\tau_4(100,400)$
  - τ_1: $(S_1, 5)$
  - τ_2: $(S_2, 15)$
  - τ_3: $(S_1, 10)$, $(S_3, 5)$
  - τ_4: $(S_2, 5)$, $(S_3, 20)$
- What is the priority ceiling of each semaphore?
  - S_1: τ_1, S_2: τ_2, S_3: τ_3
- When PCP is adopted (block once), what is the blocking time of each task?
  - τ_1: 10, τ_2: 10, τ_3: 20, τ_4: 0

RMA with Blocking Consideration (2/2)

- A RMA Example with blocking time:
  - For each task, we have to consider the execution time, period, and blocking time
  - $\tau_1(20,100,10)$, $\tau_2(30,150,10)$, $\tau_3(80, 210,20)$, $\tau_4(100,400,0)$
  - τ_1
    - $b_1 + c_1 <= 100$
  - τ_2
    - $b_2 + c_1 + c_2 <= 100$ or
    - $b_2 + 2c_1 + c_2 <= 150$
  - τ_3
    - $b_3 + c_1 + c_2 + c_3 <= 100$ or
    - $b_3 + 2c_1 + c_2 + c_3 <= 150$ or
    - $b_3 + 2c_1 + 2c_2 + c_3 <= 200$ or
    - $b_3 + 3c_1 + 2c_2 + c_3 <= 210$
  - τ_4
    - …

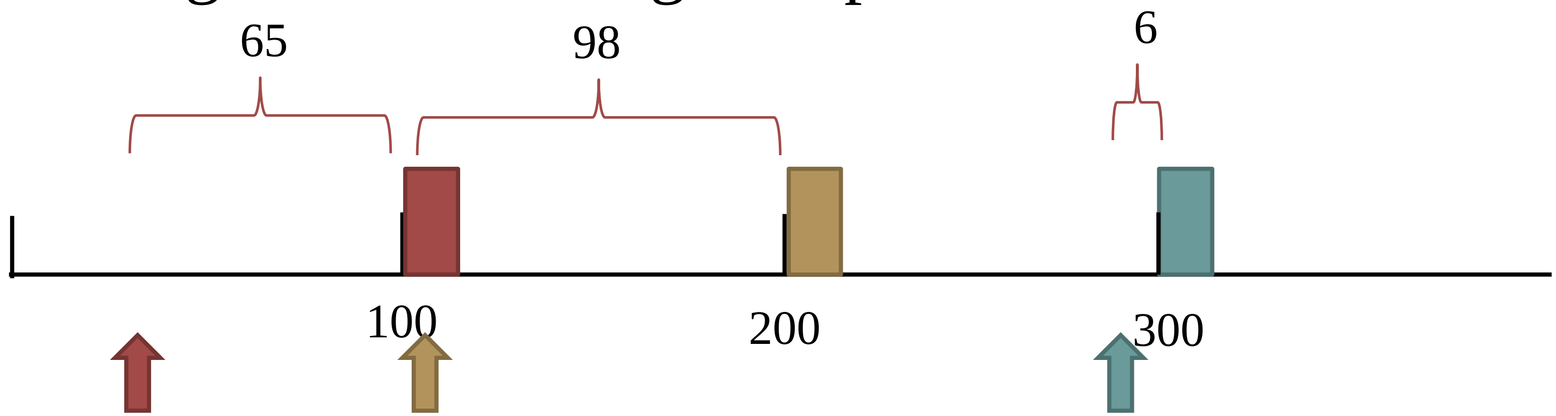
Aperiodic Servers

Observation of Aperiodic Tasks

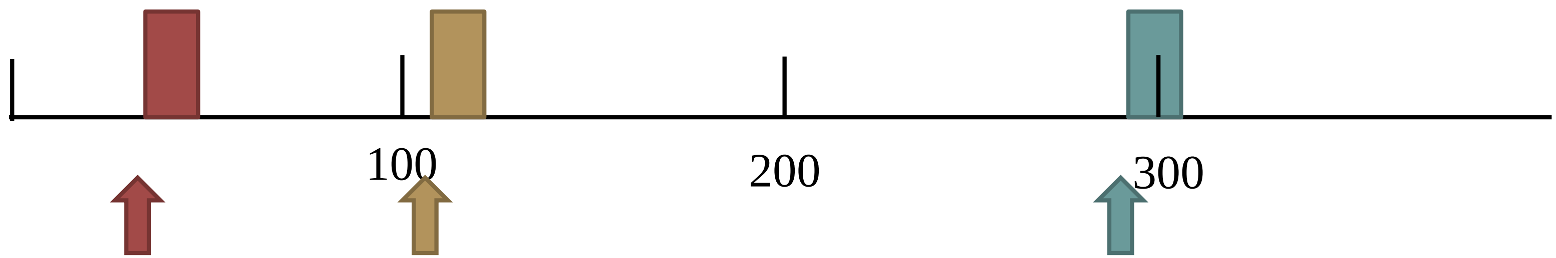
- Aperiodic tasks run at irregular intervals
- Aperiodic deadlines
  - Hard deadline: minimum inter-arrival time
  - Soft deadline: best average response time
- Services such as
  - User requests
  - Device interrupts
  - …

Scheduling Aperiodic Tasks

- Polling Server~ Average Response Time = 50 units

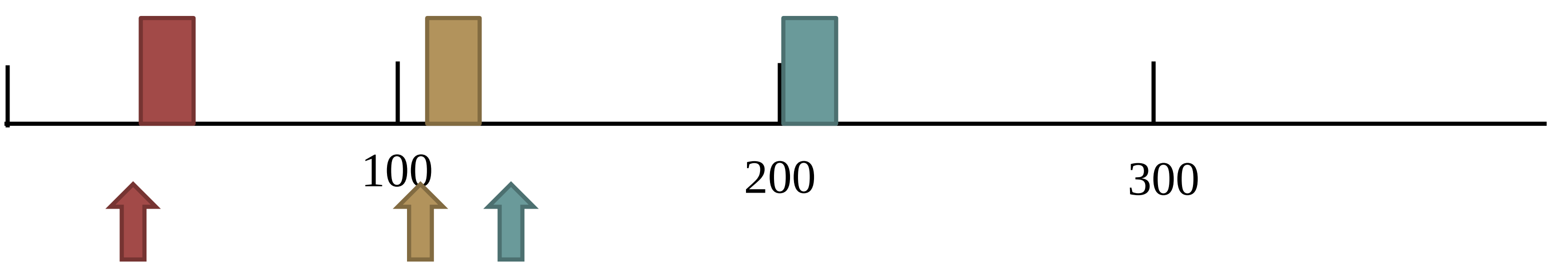


- Interrupt Server ~ Average Response Time = 1 unit



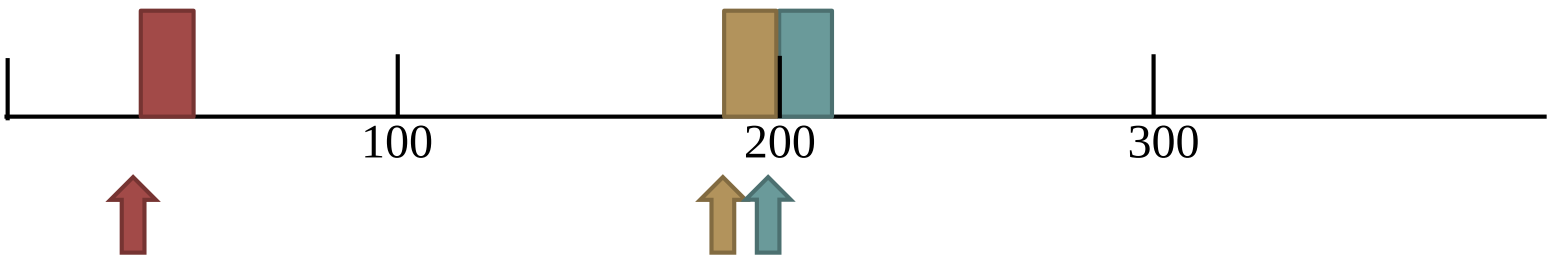
Deferrable Server

- Polling Server: the average response time is long
- Interrupt Server: the computing time of aperiodic tasks is difficult to limited
- Deferrable Server
  - In each period, a deferrable server has a execution budget
  - When execution budget is used up, server execution drops to a lower (background) priority

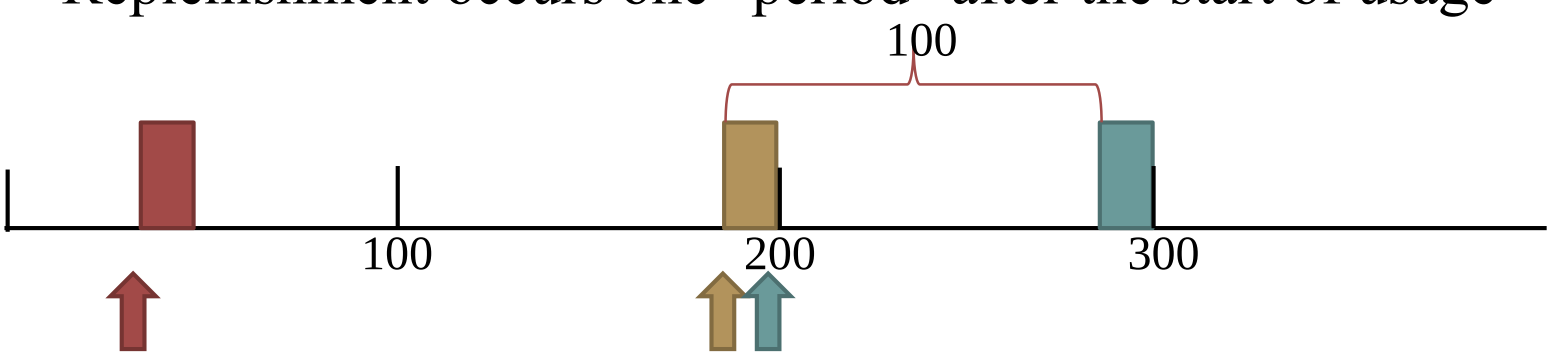


Sporadic Server

- Deferrable Server might consume two times of the execution budget in short time



- Sporadic Server
  - Replenishment occurs one "period" after the start of usage

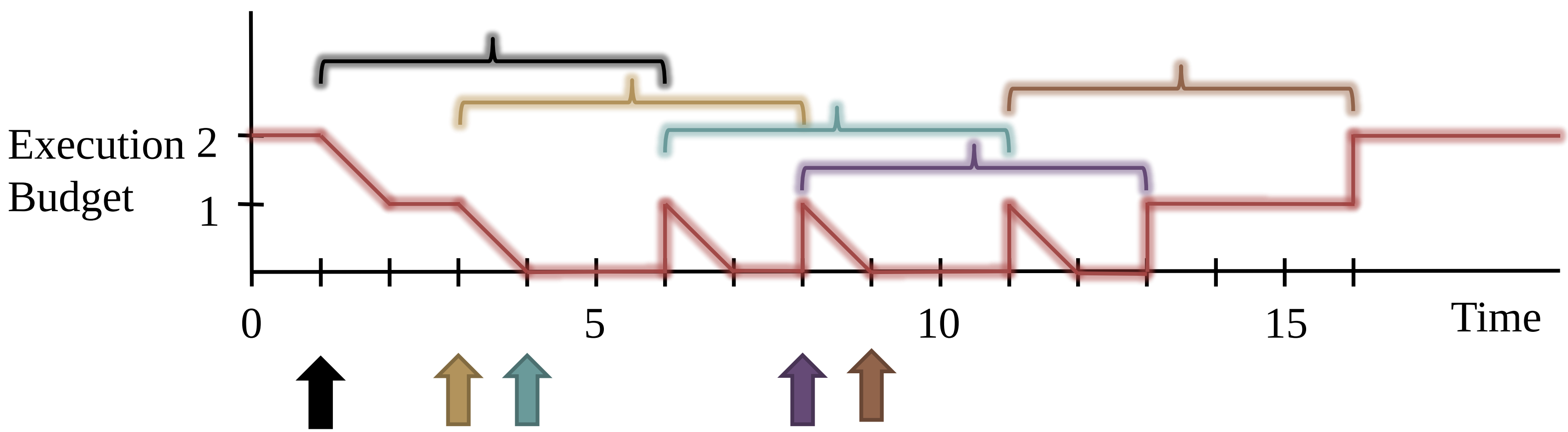


Properties of Sporadic Server

- A sporadic server differs from a deferrable server in its replenishment policy:
  - A 100 ms deferrable server replenishes its execution budget every 100 ms, no matter when the execution budget is used
  - The affect of a sporadic server on lower priority tasks is no worse than a periodic task with the same period and execution time

An Example of Sporadic Server

- A sporadic server has a replenishment period 5 and an execution budget 2
- Each event consumes the execution 1
- Events arrive at 1, 3, 4, 8, 9



Properties of Sporadic Server

- For a sporadic server has a replenishment period X and an execution budget Y
  - Given a set of sporadic tasks, If
    - Each of the aperiodic tasks has its minimum inter-arrival time no less than X
    - The total execution of the task set is no more than Y
  - All sporadic tasks can meet the deadline constraints
- When a system consists of periodic tasks and sporadic servers
  - A sporadic server with replenishment period X and an execution budget Y can be consider as a periodic task with a period X and an execution time Y
  - The system can then use analysis scheme of RM or EDF

Department of Computer Science and Information Engineering, Chang Gung University