CSC 4320/6320: Operating Systems

Chapter 05: CPU Scheduling - III

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Outline

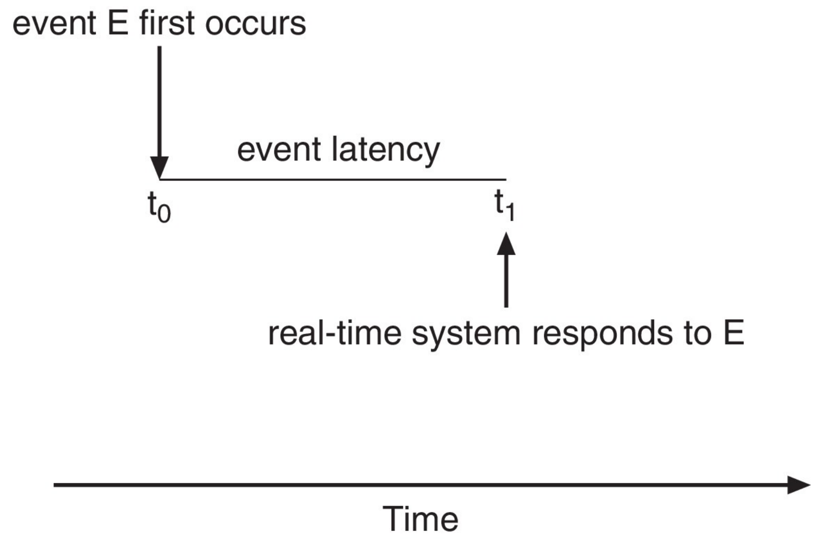
- Basic Concepts
- Scheduling Criteria
- Scheduling Algorithms
- Thread Scheduling
- Multi-Processor Scheduling
- Real-Time CPU Scheduling
- Operating Systems Examples
- Algorithm Evaluation

Objectives

- Describe various CPU scheduling algorithms
- Assess CPU scheduling algorithms based on scheduling criteria
- Explain the issues related to multiprocessor and multicore scheduling
- Describe various real-time scheduling algorithms

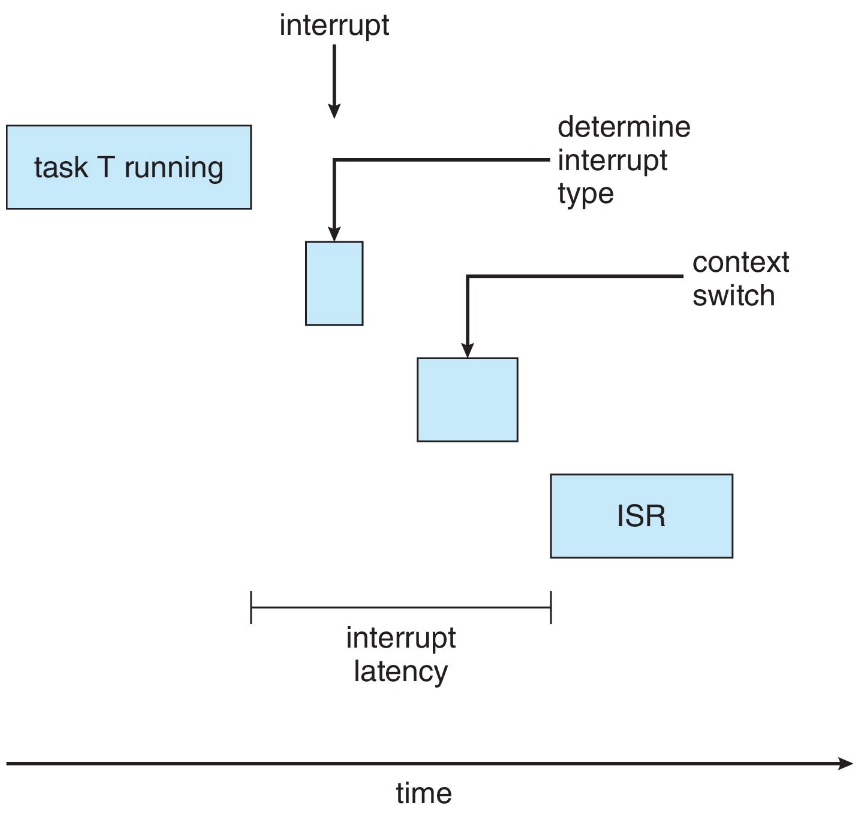
Review: Real-Time CPU Scheduling

- Event latency – the amount of time that elapses from when an event occurs to when it is serviced.
- Two types of latencies affect performance
  1. **Interrupt latency** – time from arrival of interrupt to start of routine that services interrupt
  2. **Dispatch latency** – time for scheduler dispatcher to take current process off CPU and switch to another



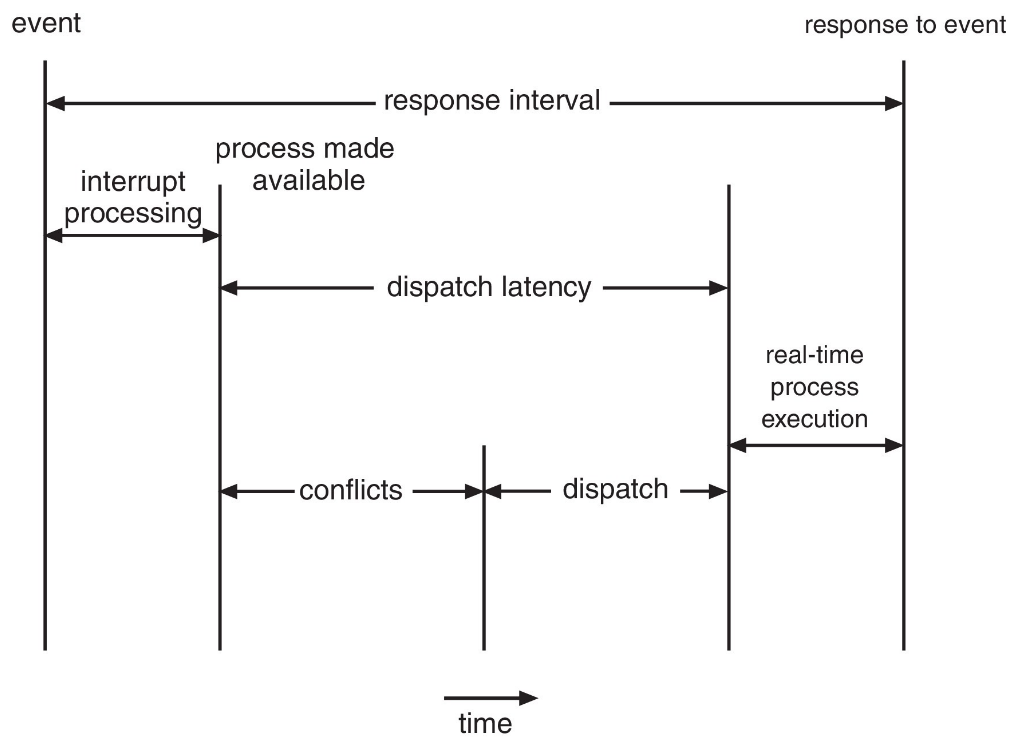
Sensor detects a fire → Delay in ISR execution → Fire alarm task is scheduled (includes conflicts resolution) → Delay in execution

Review: Interrupt Latency



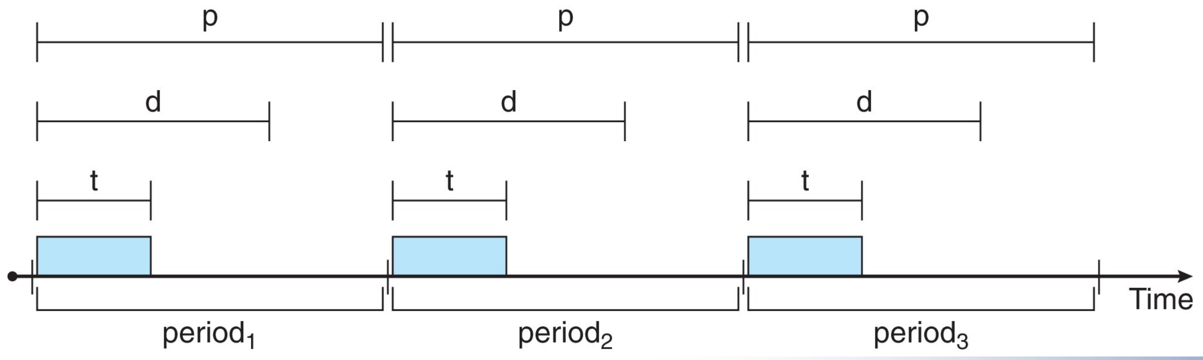
Review: Dispatch Latency

- Conflict phase of dispatch latency:
  1. Preemption of any process running in kernel mode
  2. Release by low-priority process of resources needed by high-priority processes



Review: Priority-based Scheduling

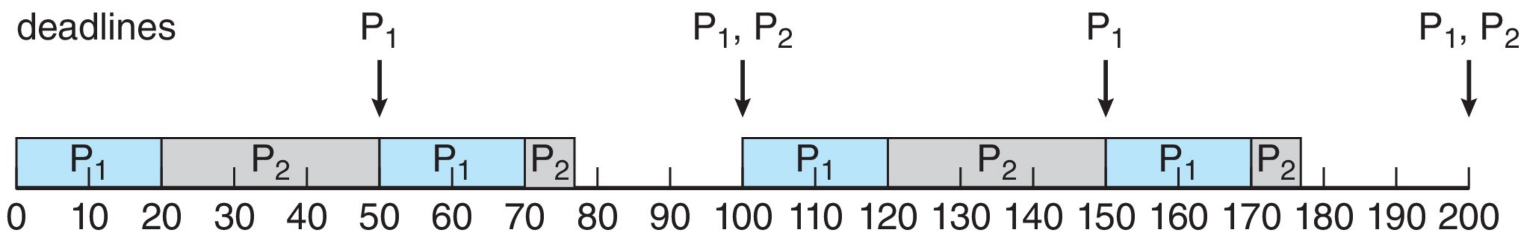
- For real-time scheduling, scheduler must support preemptive, priority-based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones require CPU at constant intervals.
  - Has processing time *t*, deadline *d*, period *p*
  - $0 \le t \le d \le p$
  - **Rate** of periodic task is $1/p$



Rate Monotonic Scheduling

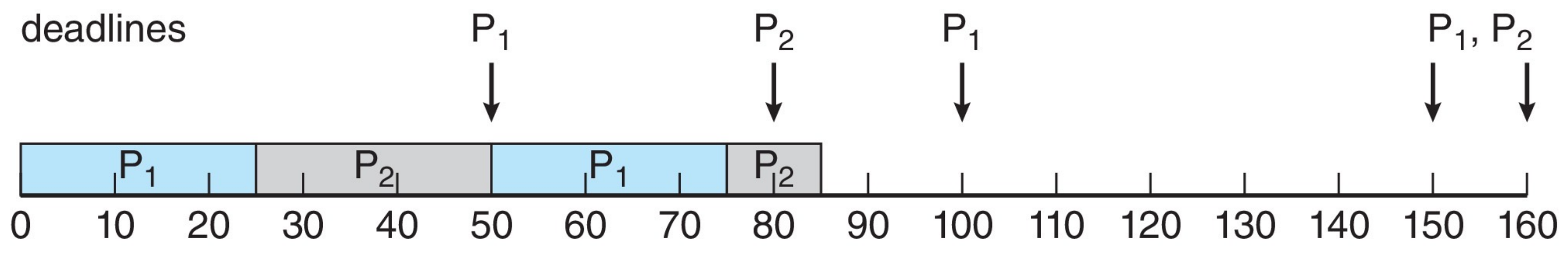
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority
- P_1 is assigned a higher priority than P_2.

Processes: P1 and P2
Periods: p1 = 50 and p2 = 100.
Processing times: t1 = 20 and t2 = 35
Deadline for each process requires that it complete its CPU burst by the start of its next period.



Missed Deadlines with Rate Monotonic Scheduling

- Process P_2 misses finishing its deadline at time 80
- Assume that process P1 has a period of p1 = 50 and a CPU burst of t1 = 25. For P2, the corresponding values are p2 = 80 and t2 = 35.
- Figure

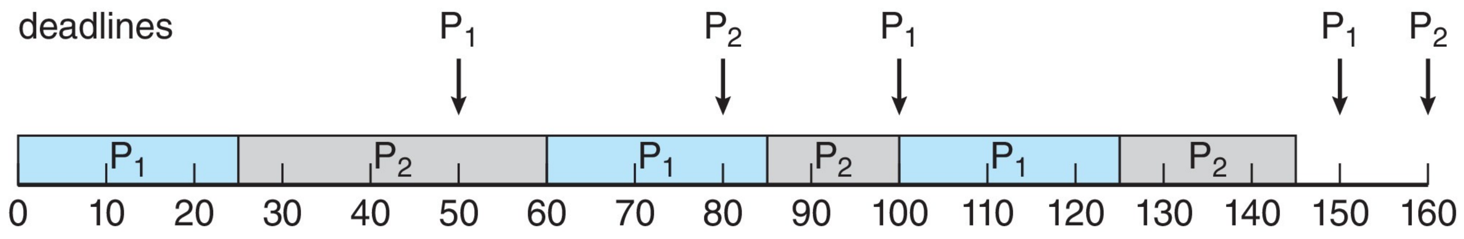


The total CPU utilization of the two processes is (25/50) + (35/80) = 0.94

Limitation: CPU utilization is bounded, and it is not always possible to maximize CPU resources fully. The worst-case CPU utilization for scheduling N processes is $U \leq N\,(2^{1/N} - 1)$ (to make sure all deadlines are met)

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned according to deadlines:
  - The earlier the deadline, the higher the priority
  - The later the deadline, the lower the priority
  - Theoretically optimal, CPU utilization is 100%
- Figure



P1 has values of p1 = 50 and t1 = 25 and
that P2 has values of p2 = 80 and t2 = 35.

- EDF scheduling does not require that processes be periodic
- a process announce its deadline to the scheduler when it becomes runnable

Proportional Share Scheduling

- *T* shares are allocated among all processes in the system
- An application receives *N* shares where *N* < *T*
- This ensures each application will receive ***N*** / *T* of the total processor time

Example: assume that a total of T = 100 shares is to be divided among three processes, A, B, and C. A is assigned 50 shares, B is assigned 15 shares, and C is assigned 20 shares.

- A will have 50 percent of total processor time,
- B will have 15 percent
- C will have 20 percent.

If a new process D requested 30 shares, the admission controller would deny D entry into the system.

Questions

The rate of a periodic task in a hard real-time system is ____, where p is a period and t is the processing time.

A) [✓ 1/p]
B) p/t
C) 1/t
D) pt

Which of the following is true of the rate-monotonic scheduling algorithm?

A) The task with the shortest period will have the lowest priority.
B) Fine-It uses a dynamic priority policy.
C) [✓ CPU utilization is bounded when using this algorithm.]
D) It is non-preemptive.

Questions

Which of the following is true of earliest-deadline-first (EDF) scheduling algorithm?

A) When a process becomes runnable, it must announce its deadline requirements to the system. ✓

B) Deadlines are assigned as following: the earlier the deadline, the lower the priority; the later the deadline, the higher the priority.

C) Priorities are fixed; that is, they cannot be adjusted when a new process starts running.

D) It assigns priorities statically according to deadline.

CSC 4320/6320: Operating Systems

Chapter 06: Synchronization Tools

Spring 2025

Outline

- Background
- The Critical-Section Problem
- Peterson's Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors
- Liveness
- Evaluation

Objectives

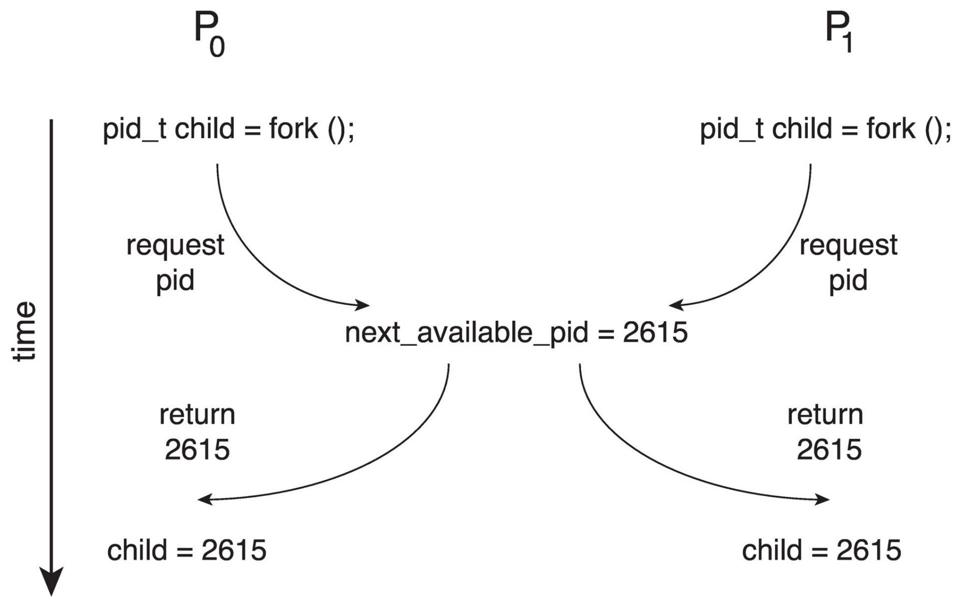
- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer, which lead to race condition.

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Question

A race condition ____.

A) results when several threads try to access the same data concurrently

B) results when several threads try to access and modify the same data concurrently ✓

C) will result only if the outcome of execution does not depend on the order in which instructions are executed

D) None of the above

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process ***P**$_i$*

```
while (true) {
        entry section
            critical section
        exit section
            remainder section
}
```

Critical-Section Problem (Cont.)

Requirements for solution to critical-section problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Interrupt-based Solution

- Entry section: disable interrupts
- Exit section: enable interrupts
- Will this solve the problem?
  - What if the critical section is code that runs for an hour?
  - Can some processes starve – never enter their critical section.
  - What if there are two CPUs?

Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Software Solution 1

- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share one variable:
  - `int turn;`
- The variable `turn` indicates whose turn it is to enter the critical section
- initially, the value of `turn` is set to *i*

Algorithm for Process P_i

```
while (true){

   while (turn = = j);

   /* critical section */

   turn = j;

   /* remainder section */

}
```