PROCESS SCHEDULING II

CS124 – Operating Systems

Spring 2024, Lecture 12

Real-Time Systems

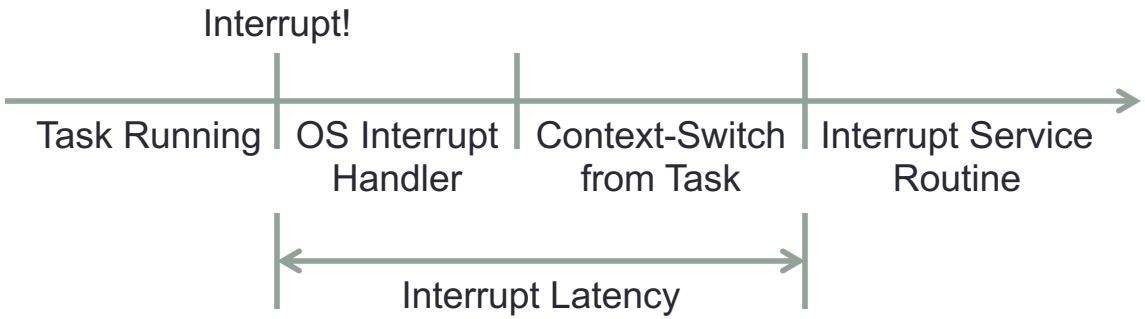
- Increasingly common to have systems with real-time scheduling requirements
- Real-time systems are driven by specific events
  - Often a periodic hardware timer interrupt
  - Can also be other events, e.g. detecting a wheel slipping, or an optical sensor triggering, or a proximity sensor reaching a threshold
- **Event latency** is the amount of time between an event occurring, and when it is actually serviced
  - Usually, real-time systems must keep event latency below a minimum required threshold
  - e.g. antilock braking system has 3-5 ms to respond to wheel-slide
- The real-time system must try to meet its deadlines, regardless of system load
  - Of course, may not always be possible…

Real-Time Systems (2)

- **Hard real-time systems** require tasks to be serviced before their deadlines, otherwise the system has failed
  - e.g. robotic assembly lines, antilock braking systems
- **Soft real-time systems** do not guarantee tasks will be serviced before their deadlines
  - Typically only guarantee that real-time tasks will be higher priority than non-real-time tasks
  - e.g. media players
- Within the operating system, two latencies affect the event latency of the system's response:
  - **Interrupt latency** is the time between an interrupt occurring, and the interrupt service routine beginning to execute
  - **Dispatch latency** is the time the dispatcher takes to switch from one process to another

Interrupt Latency

- Interrupt latency in context:



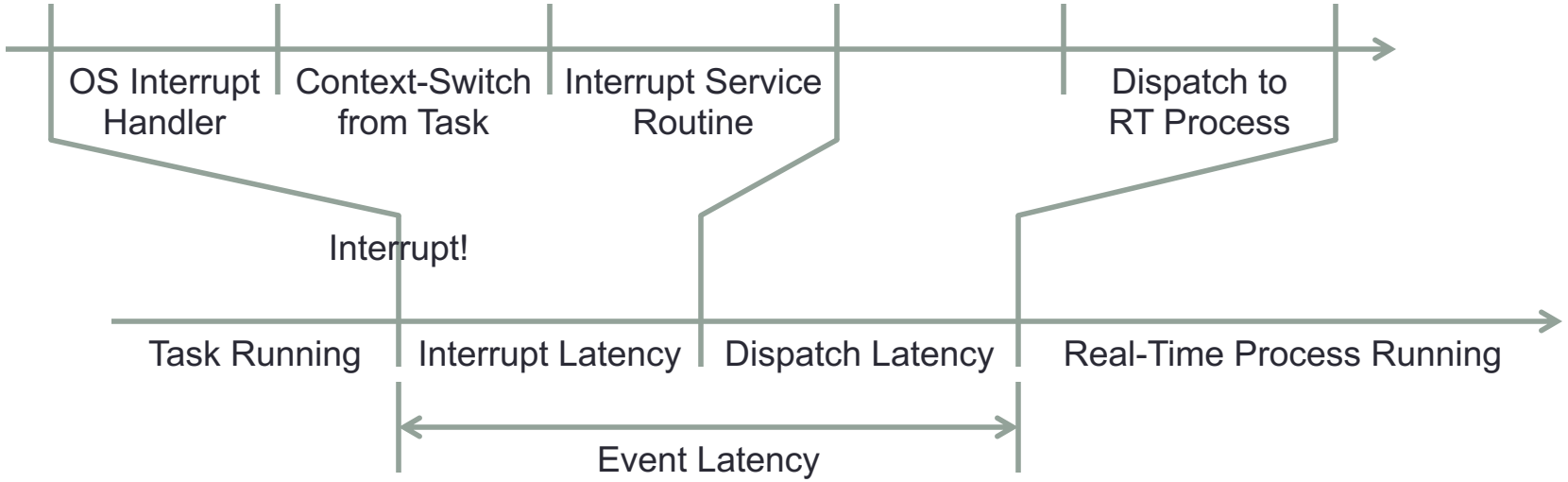
- When an interrupt occurs:
  - CPU must complete the current instruction
    - (If interrupts are turned off, must complete current critical section.)
  - CPU dispatches to the operating system's interrupt handler
  - OS interrupt handler saves the interrupted process' context, and invokes the actual interrupt service routine
- All of this can take some time…

Interrupt Latency (2)

- Interrupt latency can be *dramatically* increased by kernel code that disables interrupt handlers
  - Frequently necessary to avoid synchronization issues
- Real-time systems must disable interrupts for as short as possible, to minimize interrupt latency
- Allowing kernel preemption also helps reduce both the length and variance of interrupt latency
  - Allow the kernel to context-switch away from a user process at any time, even when the process is in kernel code

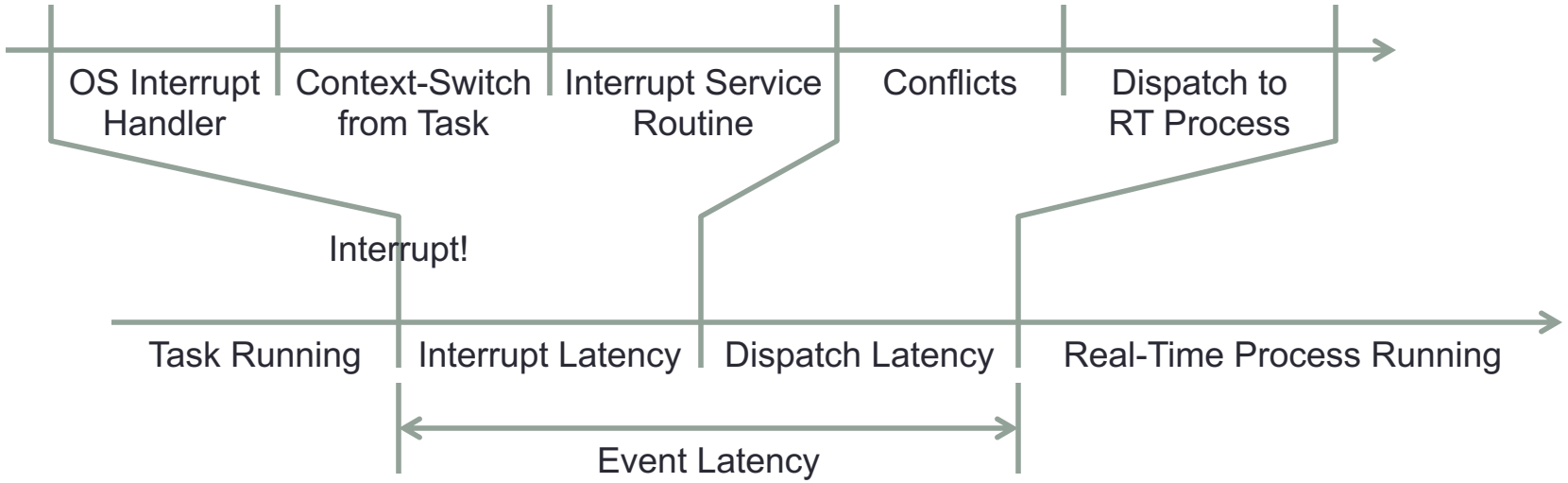
Dispatch Latency

- If the interrupt signals an event for a real-time process, the process can now execute…
  - Need to get the real-time process onto the CPU *as fast as possible*
  - Minimize dispatch latency
- Implication 1: real-time processes must be highest priority of all processes in the system
- Implication 2: the OS scheduler must support preemption of lower-priority processes by higher-priority processes



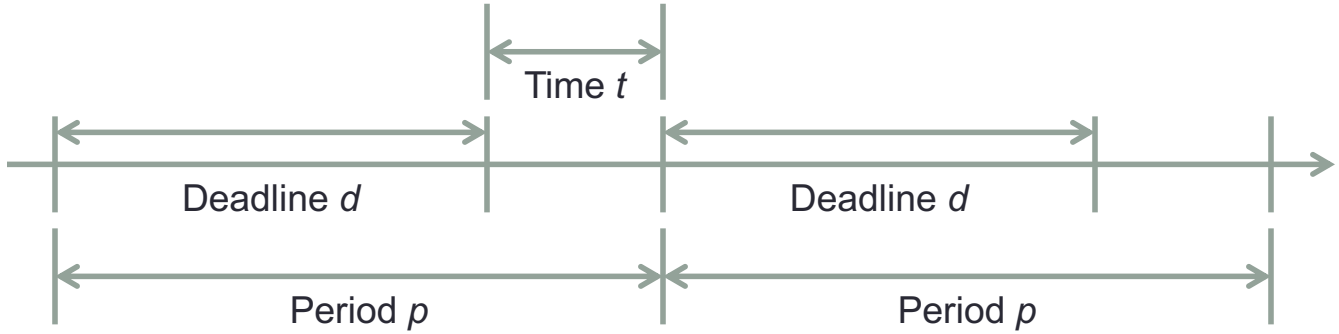
Dispatch Latency (2)

- Scheduler dispatcher must be as efficient as possible
- Also, it's possible that a lower-priority process currently holds other resources that the real-time process needs
- In this case, resource conflicts must be resolved:
  - i.e. cause lower-priority processes to release these resources so the real-time process can use them



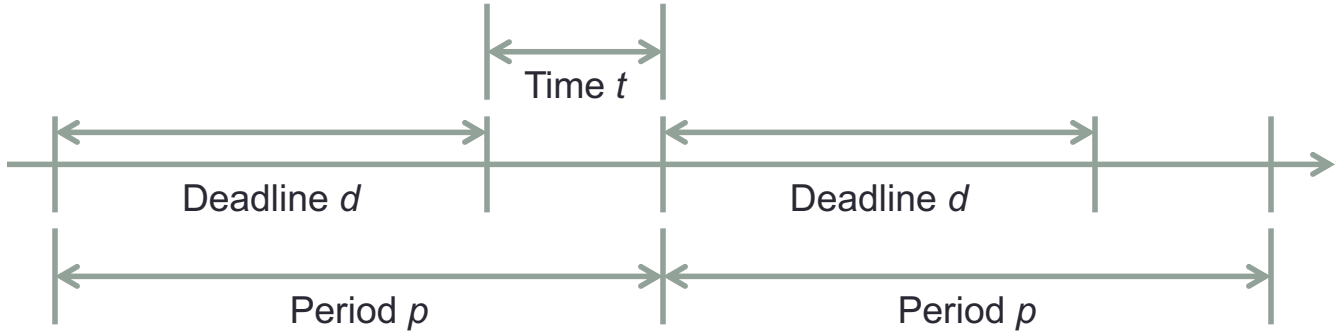
Real-Time Process Scheduling

- Real-time scheduling algorithms often make assumptions about real-time processes
- Real-time processes are often **periodic**: need the CPU at constant intervals
  - Must execute on a regular period *p*
  - Each period, execution takes a fixed time *t* to complete ($t < p$)
- Given these values (and event latency), can determine a deadline *d* for the process to receive the CPU
  - If real-time process doesn't receive the CPU within the deadline, it will not complete on time



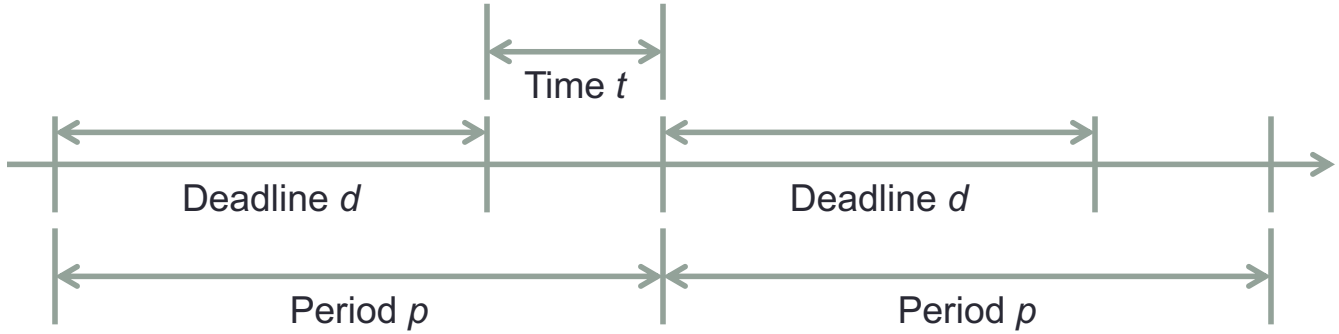
Real-Time Process Scheduling (2)

- Some OSes require real-time processes to state their scheduling requirements to the OS scheduler
- The OS can then make a decision about whether it can schedule the real-time process successfully
  - Called an **admission-control** algorithm
  - If the OS cannot schedule the process based on its requirements, the scheduling request is rejected
  - Otherwise, the OS admits the process, guaranteeing that it can satisfy the process' scheduling requirements
  - (Primarily a feature of hard real-time operating systems…)



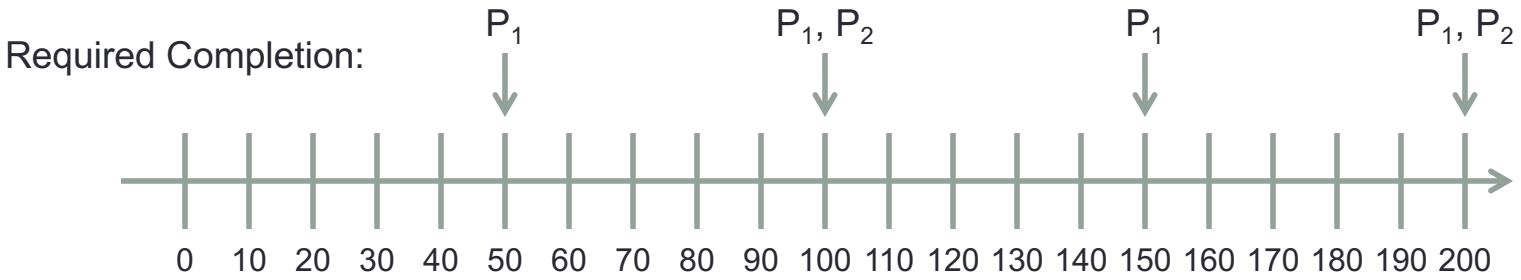
Rate-Monotonic Scheduling

- Real-time processes can be scheduled with **rate-monotonic scheduling**
- Tasks are assigned a priority inversely based on their period
  - The shorter the period, the higher the priority
- Higher-priority tasks preempt lower-priority ones
- This algorithm is optimal in terms of static priorities
  - If a set of real-time processes can't be scheduled by this algorithm, no algorithm that assigns static priorities can schedule them



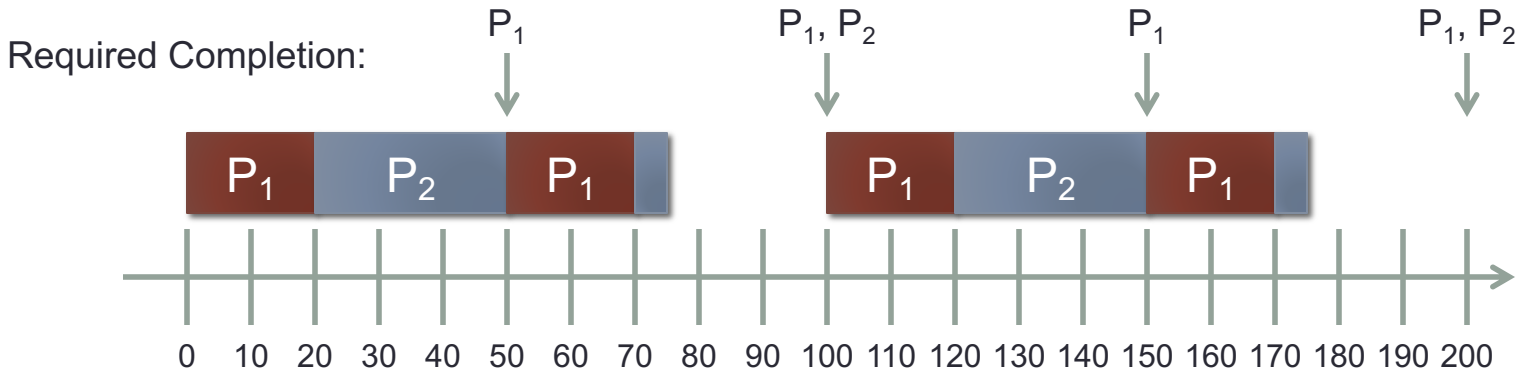
Rate-Monotonic Scheduling (2)

- Lower-priority processes may be preempted by higher-priority ones
  - As with fixed-priority preemptive scheduling, priority inversion can become an issue (and the solutions are the same)
- **Is a set of real-time processes actually schedulable?**
  - Processes must execute on a specific period *p*
  - Processes complete a CPU burst of length *t* during each period
- Example: two real-time processes
  - P_1 has a period of 50 clocks, CPU burst of 20 clocks
  - P_2 has a period of 100 clocks, CPU burst of 35 clocks



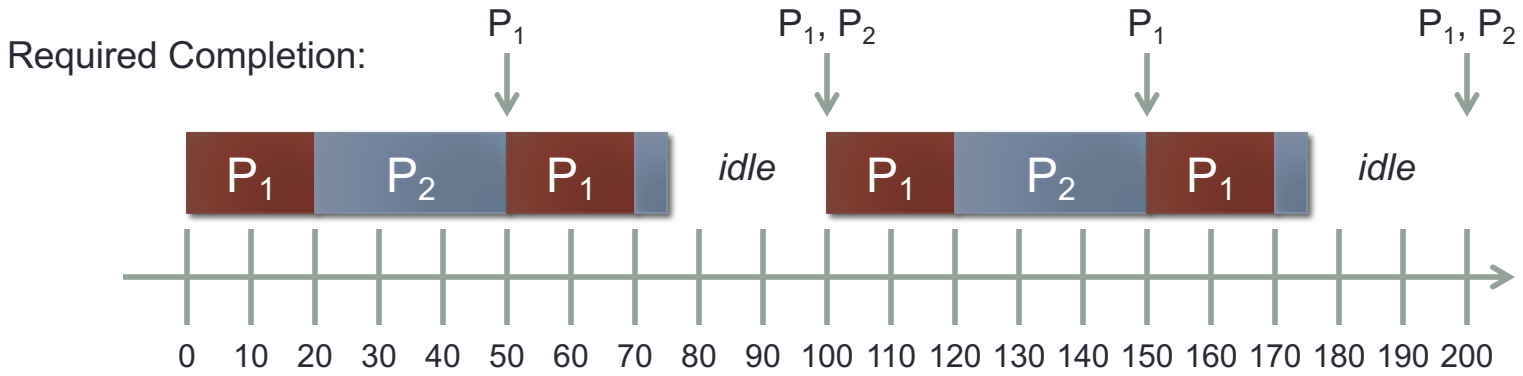
Rate-Monotonic Scheduling (2)

- Example: two real-time processes
  - P_1 has a period of 50 clocks, CPU burst of 20 clocks
  - P_2 has a period of 100 clocks, CPU burst of 35 clocks
- P_1 and P_2 can begin executing at the same time…
  - P_1 has the higher priority, so it takes the CPU first
  - P_1 completes its processing, and then P_2 starts…
- Part way through P_2's CPU burst, P_1 must execute again
  - Preempts P_2, and completes
  - P_2 regains the CPU and completes its processing



Rate-Monotonic Scheduling (3)

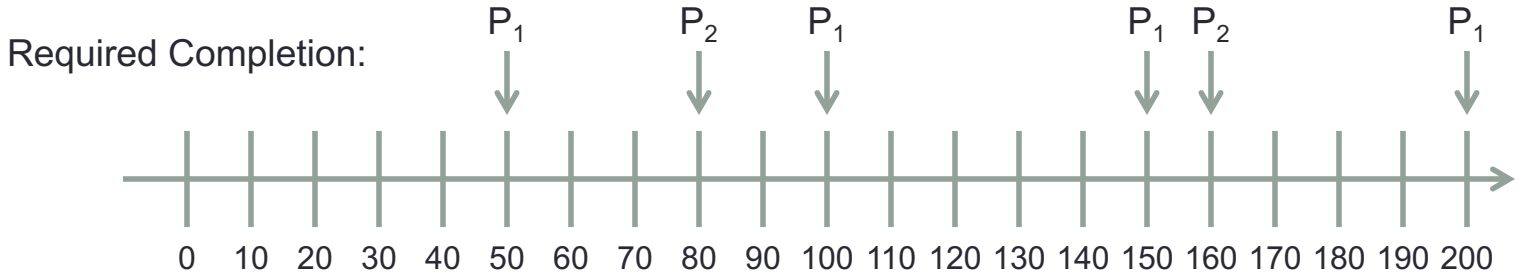
- With these processes, clearly have some time left over…
  - P_1 has a period of 50 clocks, CPU burst of 20 clocks
  - P_2 has a period of 100 clocks, CPU burst of 35 clocks
- **CPU utilization** of P1 = time / period = 20 / 50 = 0.4
- CPU utilization of P2 = 35 / 100 = 0.35
- Total CPU utilization = 0.75
- *Can we use CPU utilization to tell if a set of real-time processes can be scheduled?*



Rate-Monotonic Scheduling (4)

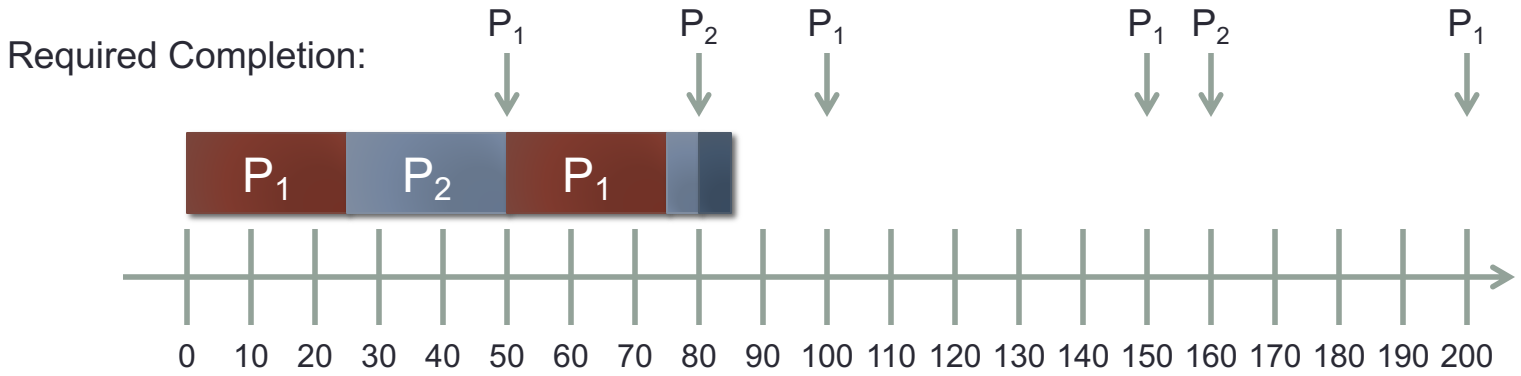
- Another example:
  - P_1 has a period of 50 clocks, CPU burst of 25 clocks
  - P_2 has a period of 80 clocks, CPU burst of 35 clocks
- CPU utilization of P_1 = time / period = 25 / 50 = 0.5
- CPU utilization of P_2 = time / period = 35 / 80 = ~0.44
- Total CPU utilization = ~0.94

- *But can we actually schedule these processes with rate-monotonic scheduling?*



Rate-Monotonic Scheduling (5)

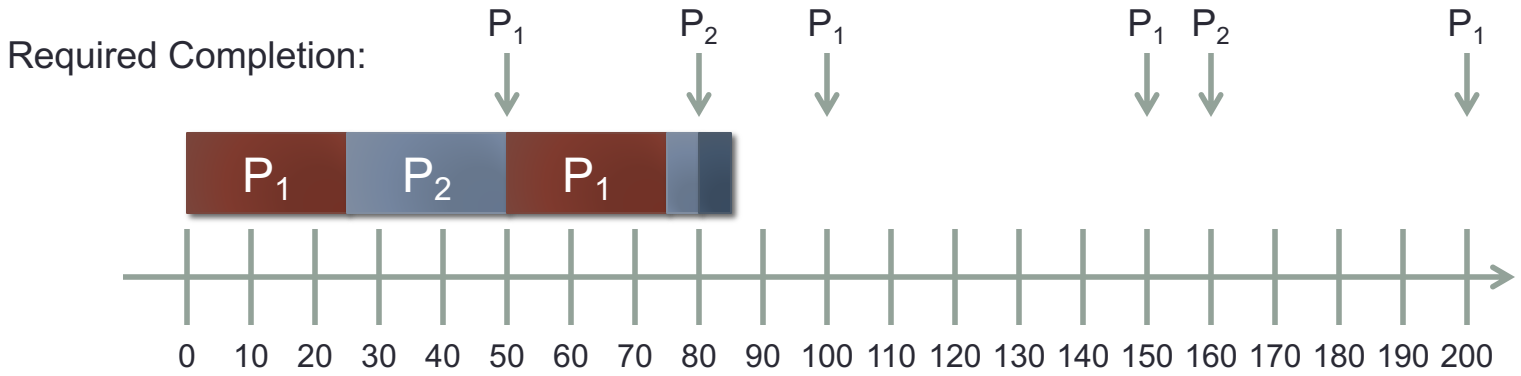
- Again, simulate rate-monotonic scheduling:
- P_1 has higher priority, executes first, and completes
- P_2 begins to execute…
- …but is preempted by P_1 before it completes…
- When P_2 regains CPU, it misses its deadline by 5 clocks
- Even though total CPU utilization is < 100%, still not good enough!



Rate-Monotonic Scheduling (6)

- For scheduling *N* real-time processes, CPU utilization can be no more than $N(2^{1/N} - 1)$
  - e.g. for two processes, CPU utilization must be ≤ 0.828
- As N → ∞, maximum CPU utilization → 0.69

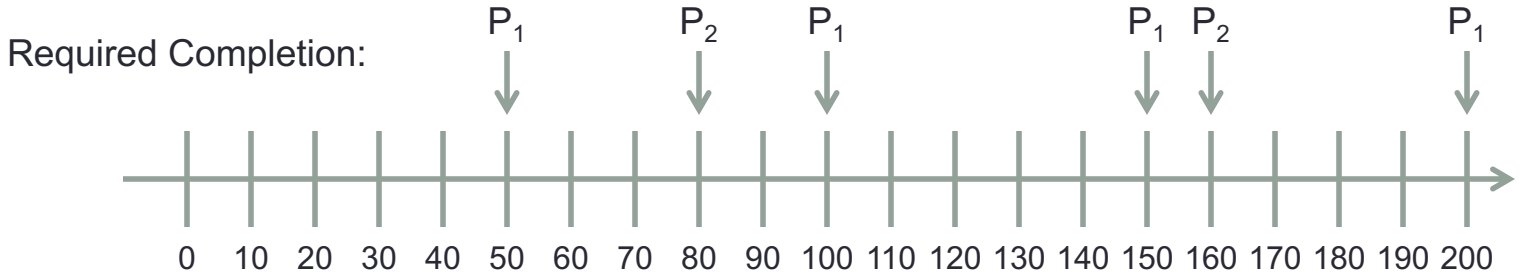
- For details on the above constraint, see:
  - Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment by Liu and Layland (1973)



Earliest Deadline First Scheduling

- **Earliest deadline first** (EDF) scheduling algorithm dynamically assigns priorities to real-time processes
  - The earlier a process' deadline, the higher its priority
  - Processes must state their deadlines to the scheduler

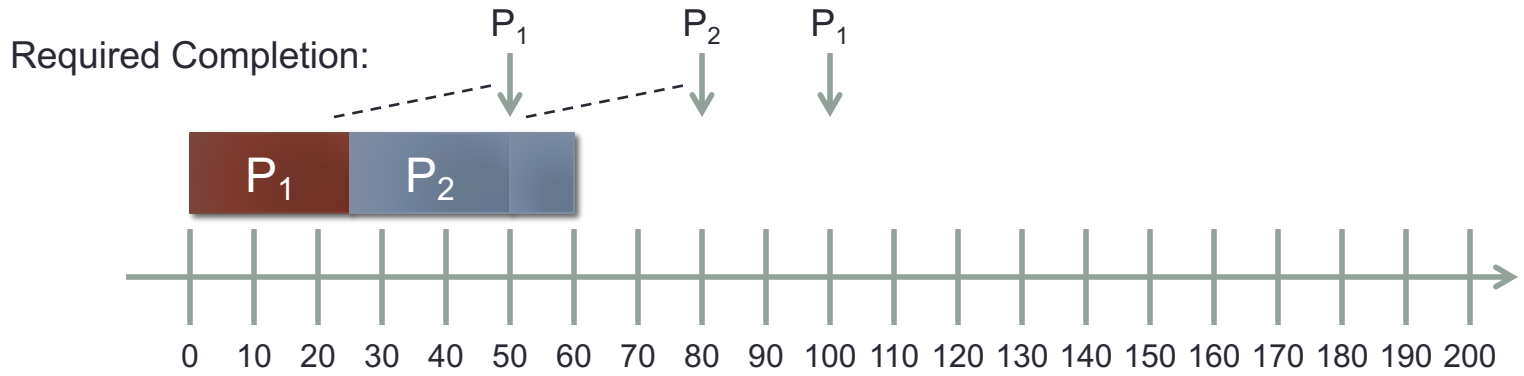
- Our previous example:
  - P_1 has a period of 50 clocks, CPU burst of 25 clocks
  - P_2 has a period of 80 clocks, CPU burst of 35 clocks



Earliest Deadline First Scheduling (2)

- With earliest deadline first scheduling:
- P_1 has earliest deadline at 50, so it runs first, completes
- P_2 begins running…
- This time when P_1's next deadline at 100 comes up, P_2's deadline at 80 is still earlier, so it runs to completion

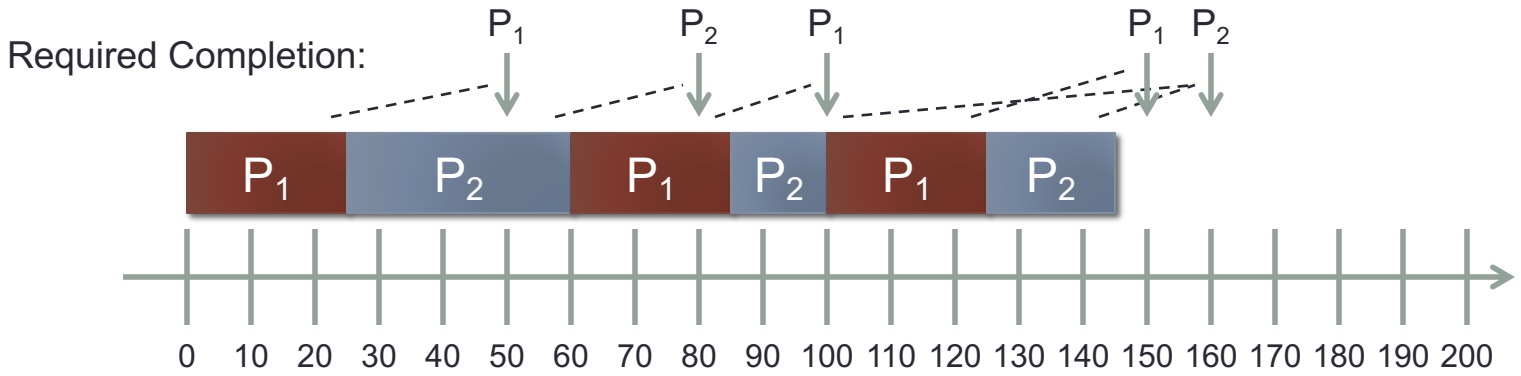
- Both processes complete in time for their first deadline



Earliest Deadline First Scheduling (3)

- P_1 starts running again for its second deadline at 100, and it completes before that deadline
- P_2 begins running for its second deadline at 160…
- …but is preempted by P_1 due to its 3rd deadline at 150
  - P_1 completes in time for its third deadline
  - P_2 resumes execution, and also completes in time for 2nd deadline

- Both processes complete in time for their deadlines



Earliest Deadline First Scheduling (4)

- Earliest deadline first scheduling has fewer requirements than rate-monotonic scheduling:
  - Doesn't actually require periodic processes, or constant CPU burst times
  - Only requires processes to state their deadlines
- EDF scheduling is theoretically optimal:
  - If a set of processes has at most 100% CPU utilization, EDF can schedule that set of processes such that it meets its deadlines
- Of course, event latency imposes an overhead…
  - Prevents EDF scheduling from achieving 100% CPU utilization

Linux 2.4 Scheduler

- Linux has used a wide range of scheduling algorithms
- Linux 2.4 scheduler: time is divided into epochs
- At the start of each epoch, scheduler assigns a priority to every process based on its behavior
  - Real-time processes have an absolute priority assigned to them, and are highest priority
  - Interactive processes have a dynamic priority assigned to them based on behavior in the previous epoch
  - Batch processes are given the lowest priority
- Each process' priority is used to compute a time quantum
  - Different processes can have different size quantums
  - (Higher-priority processes generally get larger timeslices)
  - When a process has completely used its quantum, it's preempted and another process runs

Linux 2.4 Scheduler (2)

- When scheduler is invoked, it iterates over all processes
  - Computes a “goodness,” based on priority and other considerations
    - e.g. processes can receive a bonus if they last ran on the same CPU that the scheduler is running on (encourages CPU affinity)
  - The best process is given the CPU
- Higher priority processes preempt lower priority ones
  - If a higher-priority process becomes runnable, it takes the CPU from a lower-priority process that currently holds the CPU
- The current epoch ends when all runnable processes have consumed their entire time quantum
- A process can be scheduled on the CPU multiple times within an epoch
  - e.g. it yields or blocks before its quantum is consumed, and then becomes runnable before the epoch completes

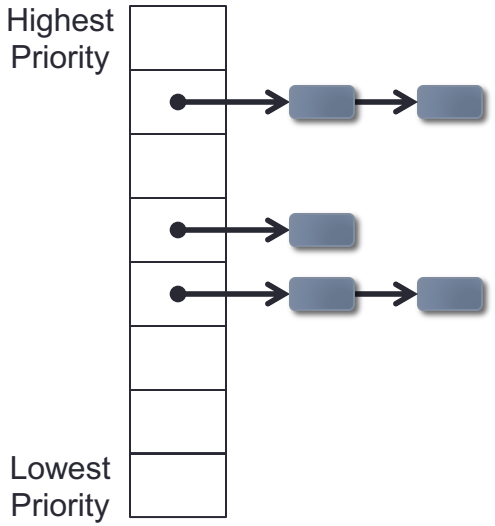
Linux 2.4 Scheduler (3)

- At start of next epoch, the scheduler iterates through all processes again, computing a new priority for each one
- Several O(N) computations in the scheduler made it scale terribly to large numbers of processes
  - Assigning a dynamic priority to each process at start of epochs
  - Choosing a process from the run queue

- Linux 2.6 scheduler was written to eliminate as many inefficiencies in the 2.4 scheduler as possible
- Captured the basic principles of the 2.4 scheduler, but a faster implementation

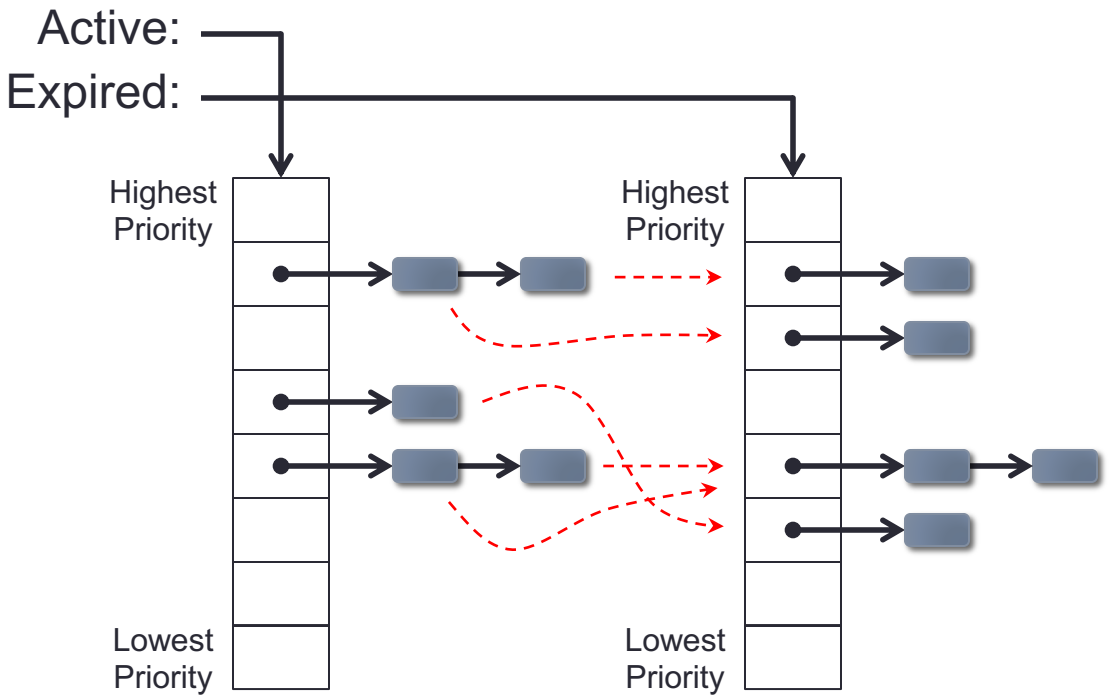
Linux 2.6 O(1) Scheduler

- Linux O(1) scheduler still includes the notion of epochs, but only informally
- Processes are maintained in a priority array structure
  - Each priority level has a queue of processes
- A bitmap records which priority levels have runnable processes
- Finding the highest-priority process to run is a constant-time operation
  - Find index of lowest 1-bit in bitmap
  - Use that index to access the priority array



Linux 2.6 O(1) Scheduler (2)

- Linux O(1) scheduler maintains two priority arrays:
  - Active array contains processes with remaining time
  - Expired array holds processes that have used up their quantum
- When an active process uses its entire quantum, it is moved to the expired array
  - When it is moved, a new priority is given to the process
- When the active priority array is empty, the epoch is over
  - O(1) scheduler swaps the active and expired pointers and starts over again



Linux 2.6 O(1) Scheduler (3)

- The O(1) scheduler is very cleverly written to be efficient
  - Same basic mechanism as the 2.4 scheduler, but much faster
  - Used until Linux kernel 2.6.23
- Unfortunately, the O(1) scheduler implementation was very complicated and difficult to maintain

- After version 2.6.23, Linux switched to the Completely Fair Scheduler (CFS)
  - Originally written by Ingo Molnar

Linux Completely Fair Scheduler

- Linux Completely Fair Scheduler (CFS):
  - Instead of maintaining processes in various queues, the CFS simply ensures that each process gets its “fair share” of the CPU
  - What constitutes a “fair share” is affected by the process’ priority; e.g. high-priority processes get a larger share, etc.
- Scheduler maintains a **virtual run time** for each process:
  - Records how long each process has run on the CPU
  - This virtual clock is inversely scaled by the process’ priority: the clock runs slower for high-priority processes, faster for low-priority
- All ready processes are maintained in a red-black tree, ordered by increasing virtual run times
  - O(log N) time to insert a process into the tree
- The leftmost process has run for shortest amount of time, and therefore has the greatest need for the CPU

Linux Completely Fair Scheduler (2)

- To facilitate rapid identification of leftmost process, a separate variable records this process
  - O(1) lookup for next process to run
- The CFS scheduler chooses a time quantum based on:
  - The **targeted latency** of the system: a time interval in which every ready process should receive the CPU at least once
  - The total number of processes in the system: the targeted latency has minimum and default values, but can be dynamically increased if a system is under heavy load
- Using these values and a process' virtual run time, the scheduler determines when to preempt each process

Linux Completely Fair Scheduler (3)

- General behaviors:
  - Processes that block or yield frequently will have lower virtual runtimes than those that don't
  - When these processes become ready to execute, they will receive the CPU very quickly

- No idea how long Linux will stick with the Completely Fair Scheduler
  - CFS has been default scheduler since 2007, so it seems to be here to stay
  - Very interesting to study Linux schedulers: very different from the most widely used approaches to process scheduling

Other Scheduling Algorithms

- Linux Completely Fair Scheduler is similar to another algorithm called **stride scheduling**
  - Unfortunately, not enough time to discuss this algorithm
  - See: Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management by Waldspurger (1995)

Next Time

- Implementation of user processes and system calls