PES UNIVERSITY ONLINE

OPERATING SYSTEMS

Multi-Processor Scheduling and Real-Time CPU Scheduling

Chandravva Hebbi
Department of Computer Science

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018
2. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018
3. Some presentation transcripts from A. Frank – P. Weisberg
4. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau

OPERATING SYSTEMS

Multi-Processor Scheduling Real-Time CPU Scheduling

Chandravva Hebbi

Department of Computer Science

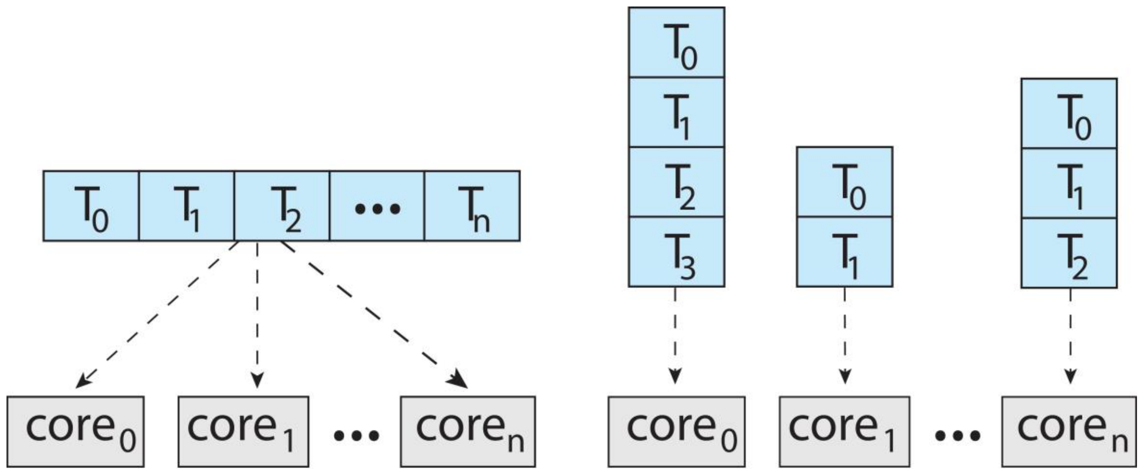
Multiple-Processor Scheduling

- CPU scheduling more complex when multiple CPUs are available, load sharing will be possible
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
  - Currently, most common.
- Virtually all modern operating systems support SMP, including Windows, Linux, and Mac OS X.

- what happens to cache memory when a process has been running on a specific processor?
    - Populated its buffer with data
    - What happens if process migrates?
- **Processor affinity** – process has affinity for processor on which it is currently running
  - **soft affinity:**
    - OS keeps the process with processor, but not guaranteed.
  - **hard affinity**
    - OS does not allow process to migrate between processors.
    - Linux implements soft affinity
    - The **sched_setaffinity()** system call, which supports hard affinity
  - Variations including **processor sets**

Multiple-Processor Scheduling (Cont.)

- Symmetric multiprocessing (SMP) is where each processor is self scheduling.
- All threads may be in a common ready queue (a)
- Each processor may have its own private queue of threads (b)

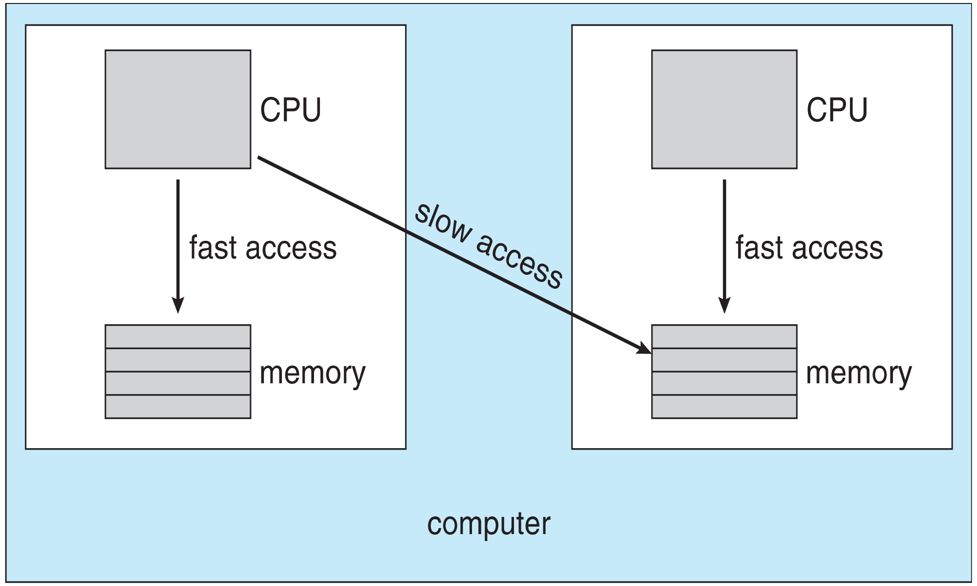


common ready queue
(a)

per-core run queues
(b)

NUMA and CPU Scheduling

- The main-memory architecture of a system can affect processor affinity issues.



If the operating system's CPU scheduler and memory-placement algorithms work together, then a process that is assigned affinity to a particular CPU can be allocated memory on the board where that CPU resides

Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency.
- Load balancing required when processors have their own queue

Approaches to load balancing

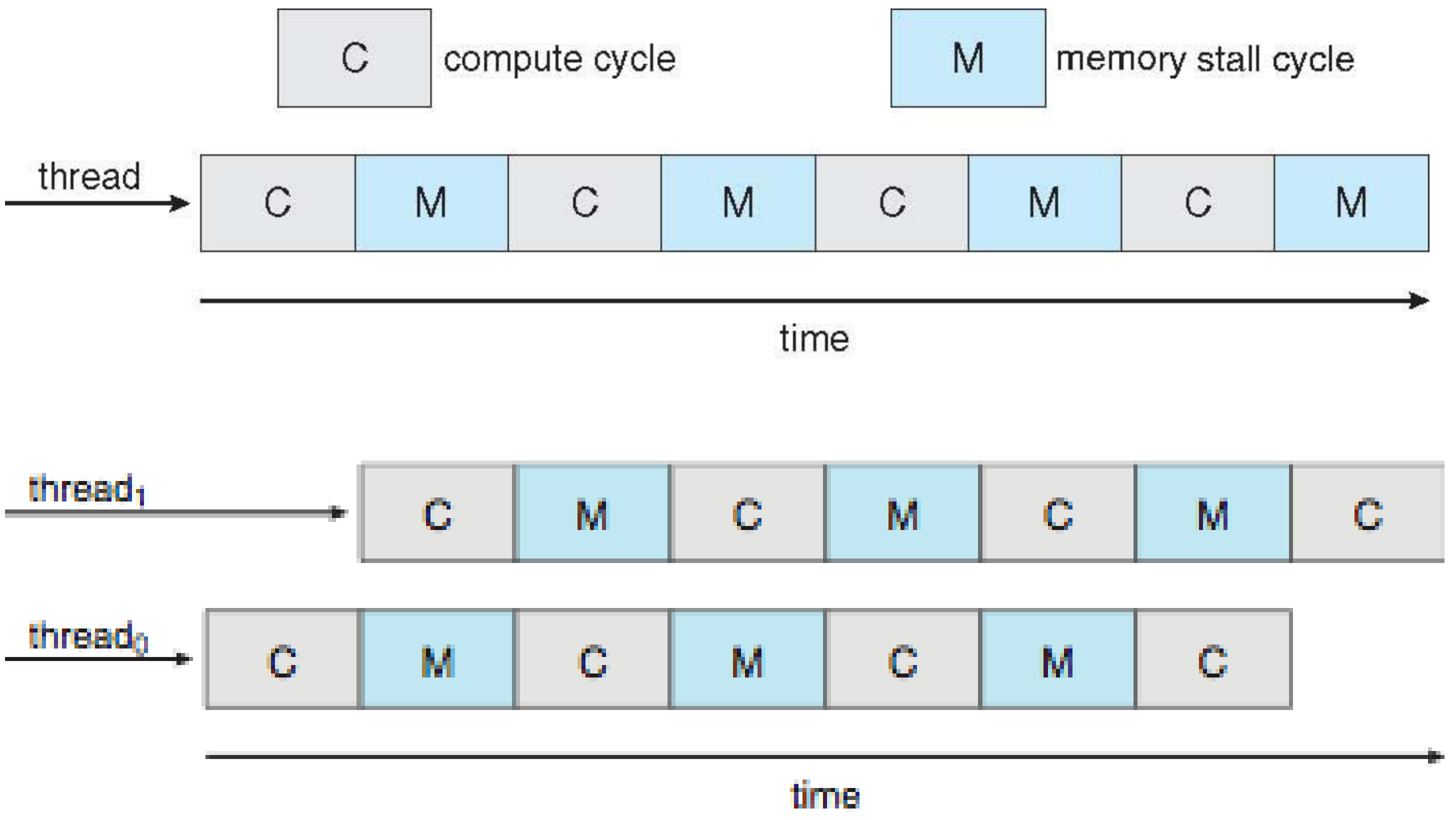
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor
- Load balancing often counteracts the benefits of process affinity

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consumes less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

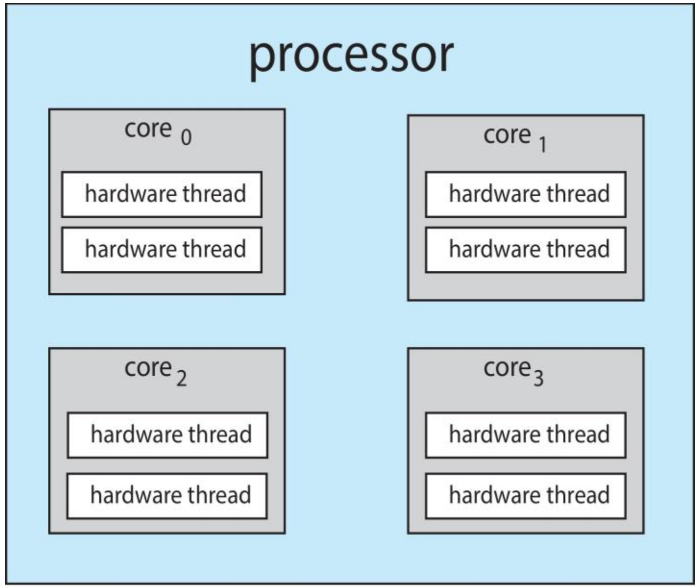
- Memory stall is the situation when a processor accesses memory, it spends a significant amount of time waiting for the data to become available.
- Each core has > 1 hardware threads. If one thread has a memory stall, switch to another thread!

Multithreaded Multicore System

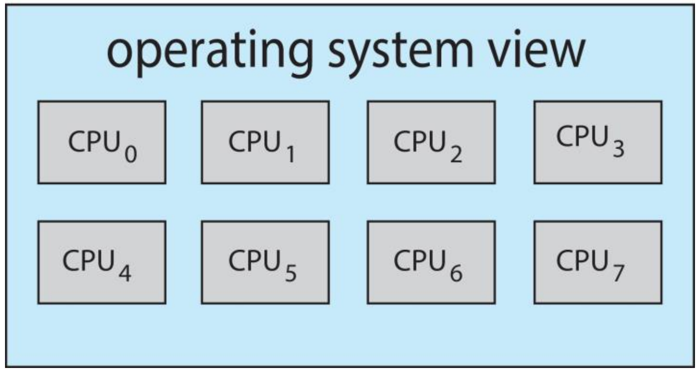


Multithreaded Multicore System (Cont.)

- **Chip-multithreading** (CMT) assigns each core multiple hardware threads. (Intel refers to this as **hyperthreading**.)



- On a quad-core system with 2 hardware threads per core, the operating system sees 8 logical processors.



Multithreaded Multicore System (Cont.)

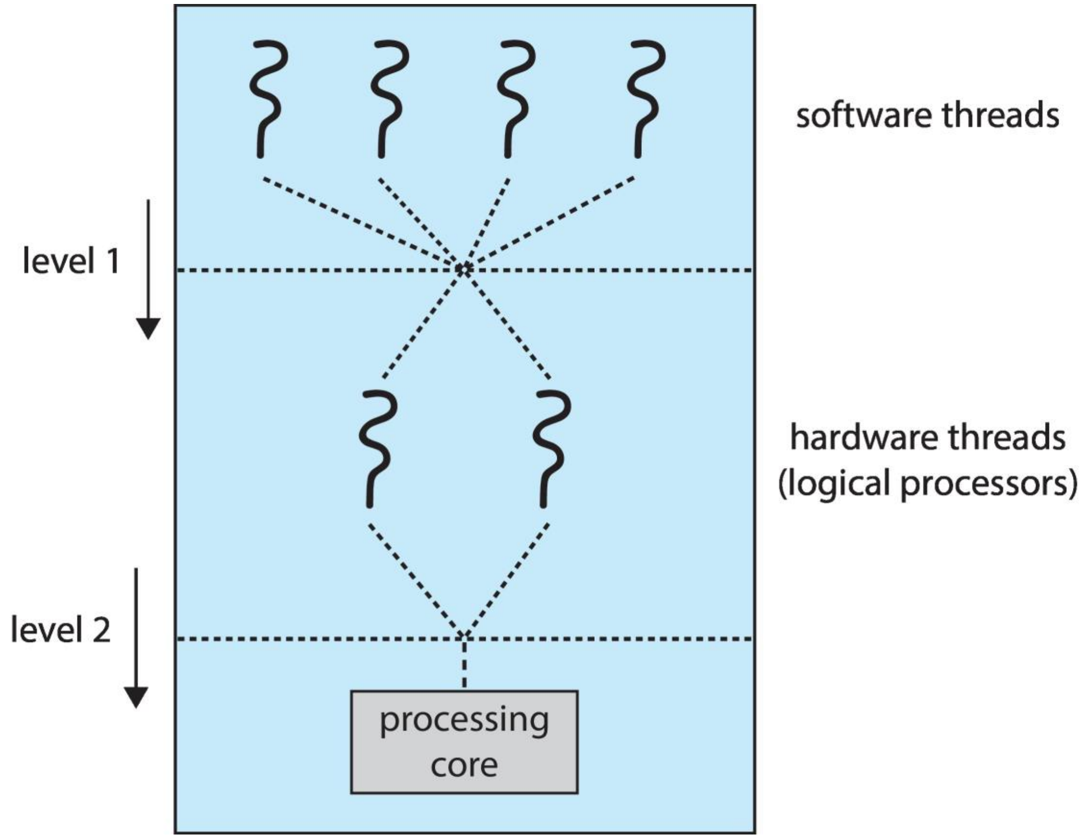
Two ways to multithread a processing core:

- **Coarse grained** multithreading
  - A thread executes on a processor until a long-latency event such as a memory stall occurs.
  - Because of the long latency processor switches to another thread.
  - Cost of switching is high. Why?
- **fine-grained** multithreading
  - Switching between the threads at finer level of granularity
  - Cost of switching low.

Multithreaded Multicore System (Cont.)

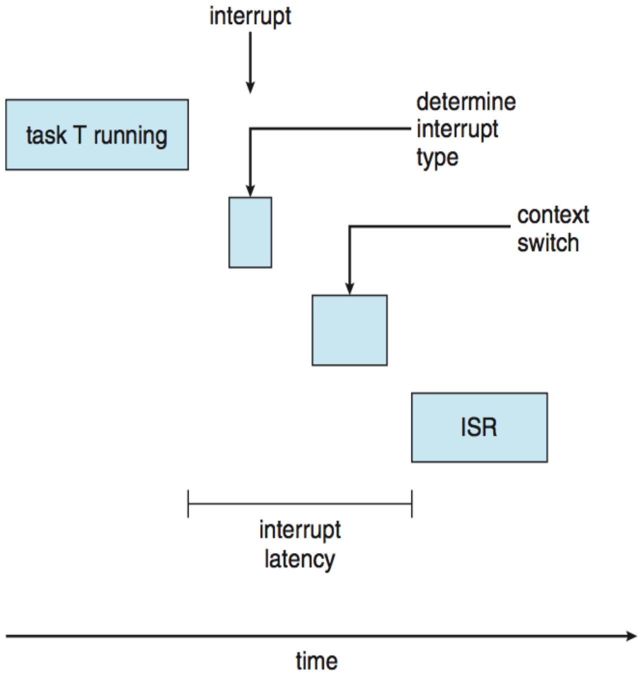
Multithreaded multicore processor needs two levels of scheduling:

1. The operating system deciding which software thread to run on a logical CPU
2. How each core decides which hardware thread to run on the physical core.



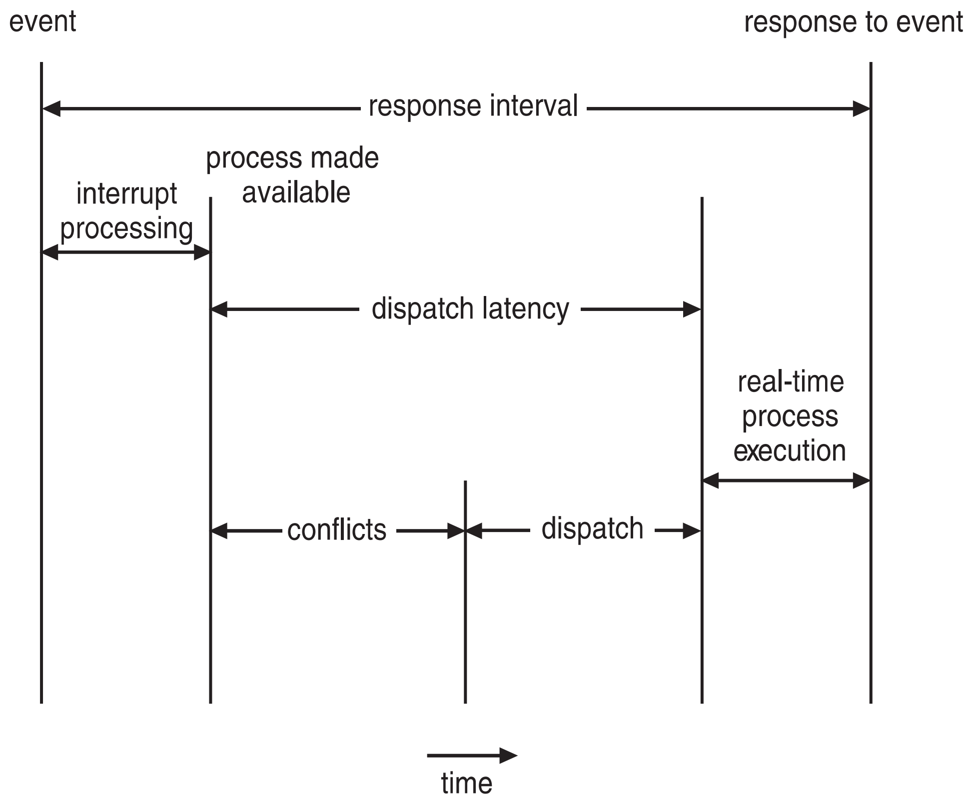
Real-Time CPU Scheduling

- Can present obvious challenges
- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline.
- Different events have different latencies.
- 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 to take current process off CPU and switch to another



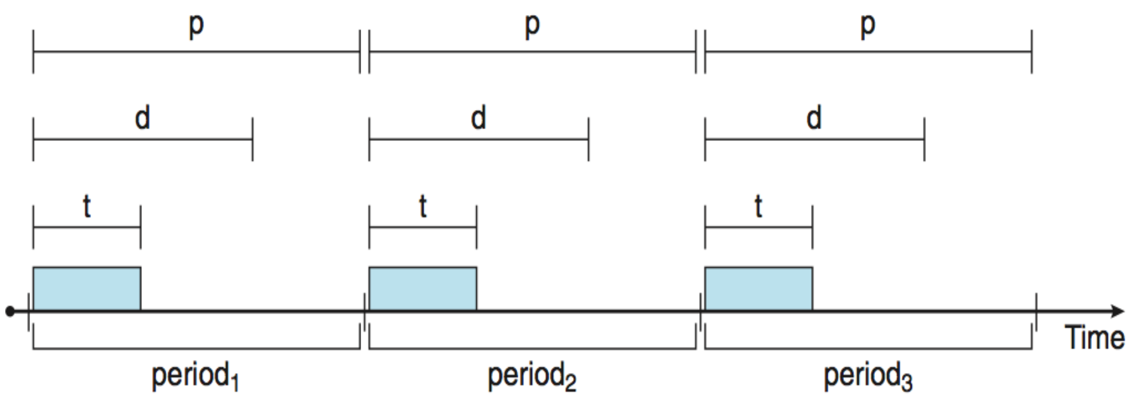
Real-Time CPU Scheduling (Cont.)

- 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



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 $1/p$
- Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements.



Rate Monotonic Scheduling

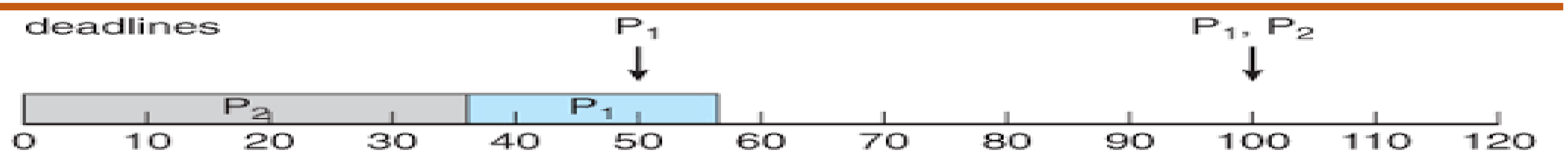
- Priority based algorithm that belongs to the static priority scheduling category for Real Time Operating Systems.
- A priority is assigned based on the inverse of its period
- Shorter periods = higher priority;
- Longer periods = lower priority

Example

P1 and P2 are 50 and 100, respectively—that is, p_1 = 50 and p_2 = 100.

The processing times are t_1 = 20 for P1 and t_2 = 35 for P2. The deadline for each process requires that it complete its CPU burst by the start of its next period.

Missed Deadlines with Rate Monotonic Scheduling



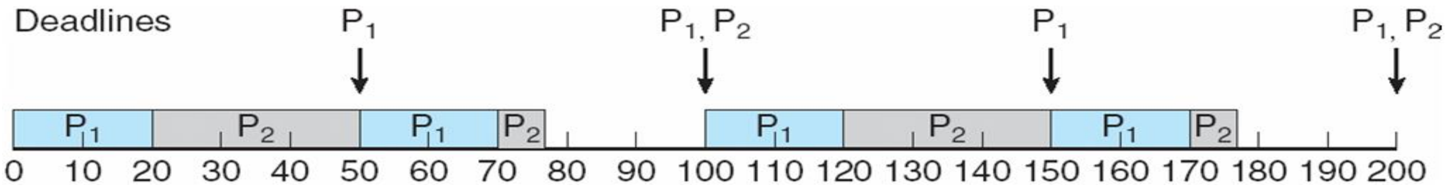
- CPU utilization-ti/pi
- P1=20/50=.40 P2=35/100=.35 total utilization=.75 which is 75%

Case 1

- P2 is given higher priority than P1.
- P1 completes at 55, it misses the deadline

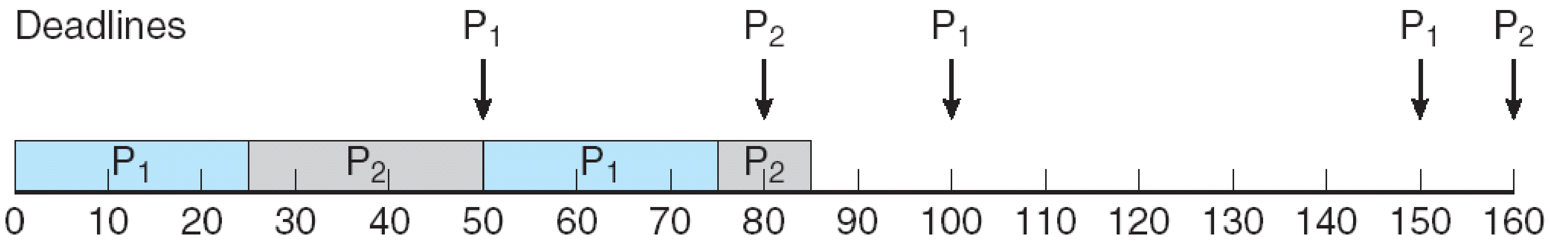
Case 2

- P1 is given highest priority as P1 time is smaller than P2.
- P1 and P2 complete by 75.
- CPU is idle till 100



Missed Deadlines with Rate Monotonic Scheduling

Consider Processes P1 and P2 with P1=50, t1=25 and P2=80,t2=35



Worst case CPU utilization for N processes is: $N(2^{1/N}-1)$

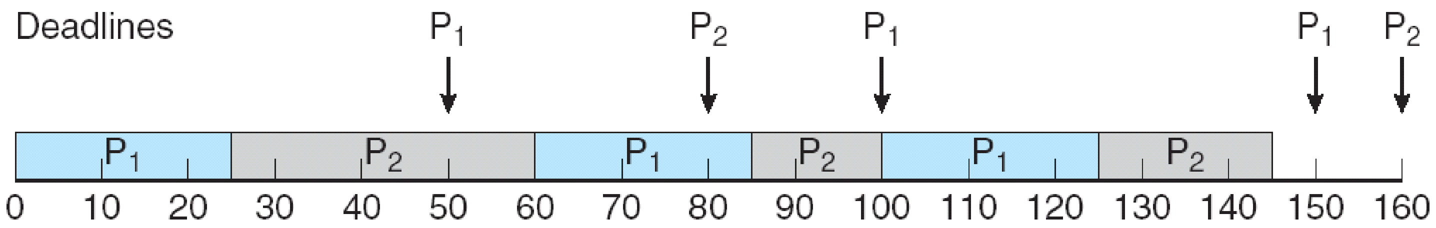
CPU utilization falls as the number of process approaches to infinity

Earliest Deadline First Scheduling (EDF)

- Priorities are assigned dynamically according to deadlines:
  - the earlier the deadline, the higher the priority;
  - the later the deadline, the lower the priority
- Process must announce its deadline when it becomes runnable

Example

- Consider Processes P1 and P2 with P1=50, t1=25 and P2=80,t2=35



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.

As an 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.

What happens if new process D request for 30 shares

POSIX Real-Time Scheduling

- The POSIX.1b standard
- API provides functions for managing real-time threads
- Defines two scheduling classes for real-time threads:

1. SCHED_FIFO - threads are scheduled using a FCFS strategy with a FIFO queue. There is no time-slicing for threads of equal priority
2. SCHED_RR - similar to SCHED_FIFO except time-slicing occurs for threads of equal priority

- Defines two functions for getting and setting scheduling policy:

1. **pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)**
2. **pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)**

THANK YOU

Chandravva Hebbi
Department of Computer Science Engineering
chandravvahebbi @pes.edu