CPU SCHEDULING

Ruben Acuña

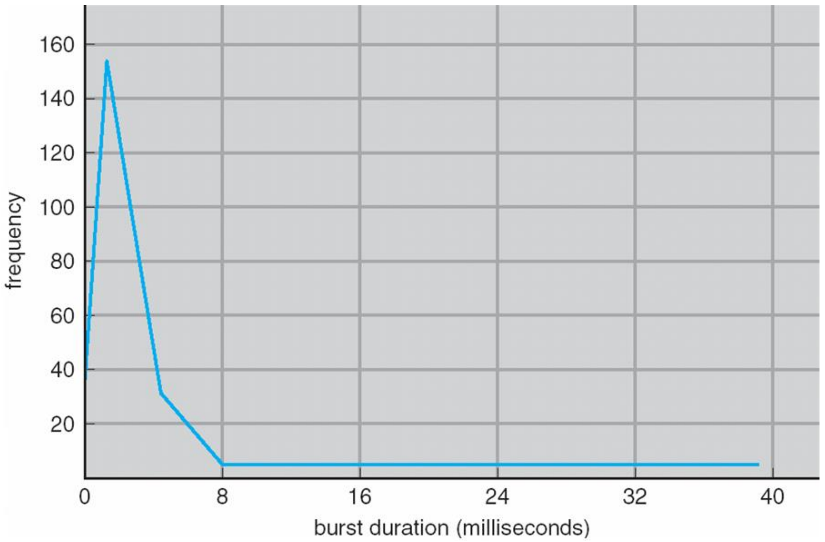
Spring 2018

2

BASIC CONCEPTS (6.1)

BURST PATTERNS

- In general, an active process alternates between performing CPU and I/O work. It may also be queued to perform either of these actions.
  - We will say that processes perform *bursts* of either type of work.
  - CPU bursts follow an exponential pattern.
- From our previous discussions, processes that need CPU will queue in the ready queue, while processes waiting for I/O will exist in a special I/O queue. Queues may not be FIFO
- Our aim with a scheduler algorithm is thus to distribute CPU time.



Also called the interval scheduling problem.

SCHEDULING

- There are four events that can trigger a change in the currently executing process:
    1. "When a process switches from the running state to the waiting state
    2. When a process switches from the running state to the ready state
    3. When a process switches from the waiting state to the ready state
    4. When a process terminates“
- Depending on whether the scheduler is non-preemptive and preemptive, not all of these operations may be permitted.

Not to be confused with streaming algorithms.

LIVE ALGORITHMS

- Until now, you have probably seen only algorithms that terminate.
- You may even have been told that termination is a condition for something to be call an algorithm.
- This is not very general – consider the problem of scheduling processes for execution on the CPU:
  - How can we possibly know all of the processes that will run ahead of time?
- This also means we are going to see two types of algorithms: Those which assume pre-knowledge of the set of jobs (algorithms), and those that do not (live algorithms).
- Since the amount of information available to each algorithm is different, they are fundamentally not comparable.

GREEDY ALGORITHMS

- For a live algorithm, we have no choice but to make a choice based on the information we have available.
- Later on, we'll seen an example of this type of algorithm: a greedy approach to scheduling.
- *In a **greedy algorithm**, we make the general assumption that if we compute the optimal solution to sub-problems, or even individual algorithmic steps, then we will later be able to construct the globally optimal solution to the whole problem. The one point is that no knowledge of the entire problem will impact how the sub-problems will be solved.*
- Often greedy algorithms embed the notation of a heuristic which generates an approximate best result.
- It is uncommon but still possible (!), to generate the optimal solution for the problem.

SCHEDULING CRITERIA (6.2)

SCHEDULING CRITERIA

- How good our scheduling algorithms are can be judged in several ways:
  - *CPU utilization* - of the possible load that could be put on the CPU, how much actually is used.
  - *Throughput* - for some period of time, how many tasks are completed.
  - *Turnaround time* - total completion time for a task.
  - *Waiting time* - time that a task spends in the ready queue.
  - *Response time* - time until a task first gives a result.

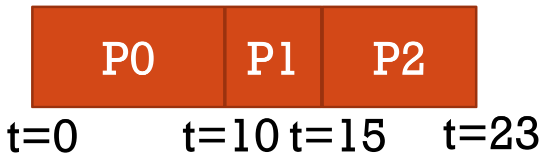
- (An additional implicit criteria could be whether we need a live algorithm.)
- Question: why isn't something like bounded waiting time on this list?

SCHEDULING ALGORITHMS (6.3)

FIRST-COME FIRST-SERVED

Assume that this is next CPU burst time, which will be followed some I/O waiting time.

- Simplest idea: use a FIFO queue.
  - Is this a live algorithm?
- As a simple example, consider a set of ordered pairs of processes and CPU time: (P0, 10), (P1, 5), (P2, 8).



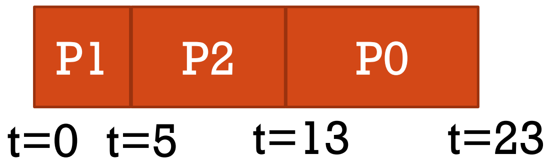
- It seems like FCS would do poorly for some metrics. Let's do some analysis:
  - CPU utilization: Hmm…
  - Throughput: Hmm…
  - Average Completion Time: (10+15+23)/3=16
  - Average Waiting Time: (0+10+15)/3=8.33
  - Response time: Hmm…

Is this optimal?

- Of courses there are issues with this scheme:
  - *Convoy Effect*: assuming a non-preemptive model, we may waste time on a large job when smaller jobs could be cleared and moved directly to an I/O queue.

SHORTEST-JOB-FIRST

- Simple idea: sort the queue.
  - Is this a live algorithm?
- As a simple example, consider a set of ordered pairs of a processes and CPU time: (P0, 10), (P1, 5), (P2, 8).



- Let's do some analysis:
  - Average Completion Time: (5+13+23)/3=13.6
  - Average Waiting Time: (0+5+13)/3=6

Is this optimal?

- Still limited by the liveness…

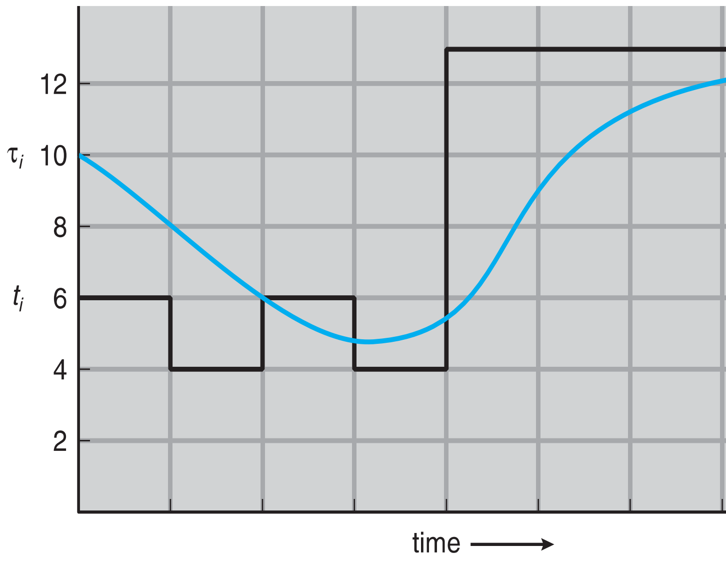
SHORTEST-JOB-FIRST LIVE

- Let's assume we have a set of processes for which we try to guess their next CPU burst time via a moving average.
- Let τ_n represent a guess for CPU time of a process's nth burst. Let t_n be the CPU time for process's nth burst.
- We will say that τ_n may be calculated as a linear combination of last burst time and the last predicted burst time:

$$\alpha \in [0,1]$$

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- What impact does changing α have?

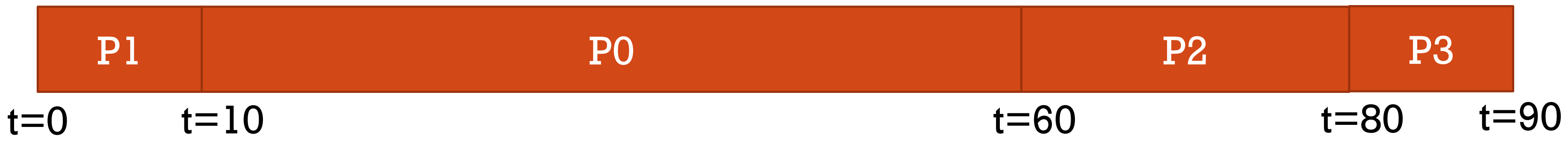


CPU burst (t_i)		6	4	6	4	13	13
"guess" (τ_i)	10	8	6	6	5	9	11

$\alpha = .5 \quad \tau_0 = 10$

PRIORITY SCHEDULING

- Simple idea: sort by priority.
  - How might priority be determined?
- As a simple example, consider a set of ordered triples of a processes, CPU time, and priority: (P0, 50, 3), (P1, 10, 1), (P2, 20, 4), (P3, 10, 5).



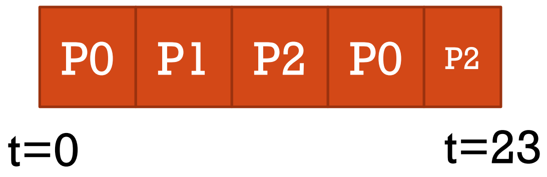
- Let's do some analysis:
  - Average Completion Time: (10+60+80+90)/4=60
  - Average Waiting Time: (0+10+60+80)/4=37.5

Is this stuff "optimal"?

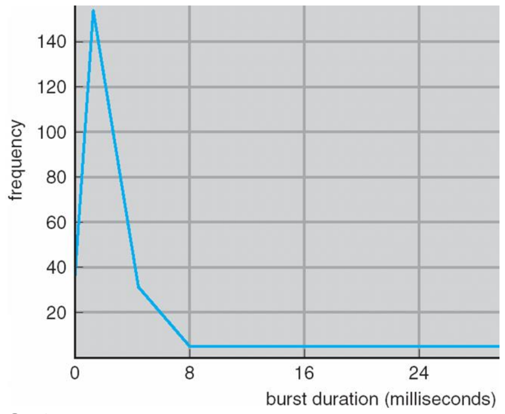
- Of courses there are issues with this scheme:
  - Bounded Waiting Time: what happens if new high priority processes keep getting added? How can we fix it?

ROUND ROBIN SCHEDULING

- Idea: make each process take turns using a specific amount of CPU time.
- We will say that CPU time is divided into segments called *time quantum* that represent the time a process may be active before being preempted.
- As an example, consider the processes and CPU usage: (P0, 10), (P1, 5), (P2, 8). Say we pick a time quantum of 5.



Maybe this is useful…?



- Let's do some analysis:
  - Average Completion Time: (20+10+23)/3=17.6
  - Average Waiting Time: (10+5+15)/3=10
- Main task for us is to optimize the time quantum selection:
  - What happens to performance if we make it high? Why?
  - What happens to performance if we make it low? Why?

Image from Operating System Concepts.

MULTILEVEL QUEUE SCHEDULING

- Idea: processes can be divided into categories that correspond to how their completion/waiting times should work. For example, *foreground* tasks need to be addressed immediately, while *background* tasks can wait.
  - Clearly, these tasks have different performance expectations, so using the same scheduler is a bad idea.
- Thus, we have the notation of a *multilevel queue*.
  - Overall we need some way to allocate time among levels.
  - Each queue may be managed by a different algorithm.
- What mechanisms can we use for these two scheduling concerns?

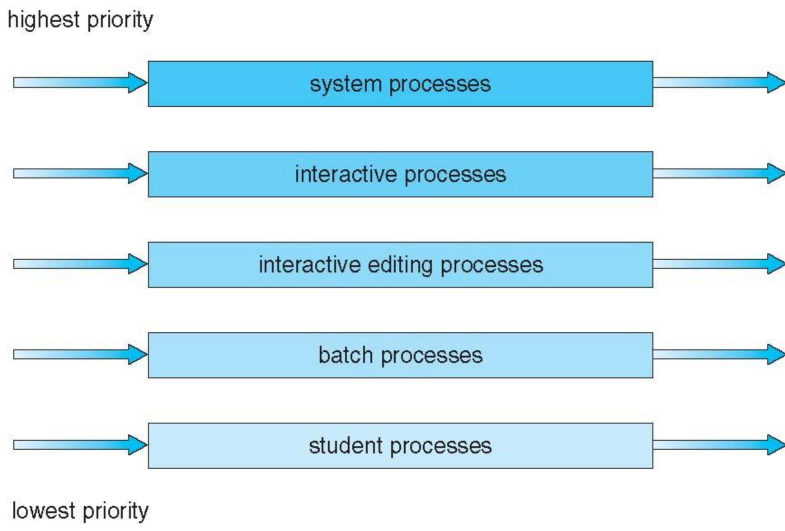


Image from Operating System Concepts.

MULTILEVEL FEEDBACK QUEUE SCHEDULING

- To elaborate, we can also consider a *multilevel feedback queue* were processes may be migrated across levels.
  - Processes in a high-priority queue must complete before processes in a lower queue can execute.

- Why exactly would this be more useful than a regular multi-level queue?
- What exactly is this scheme optimizing for?

- Difference from multilevel:
  - Need mechanism to move to higher-priority queue.
  - Need mechanism to move to lower-priority queue.

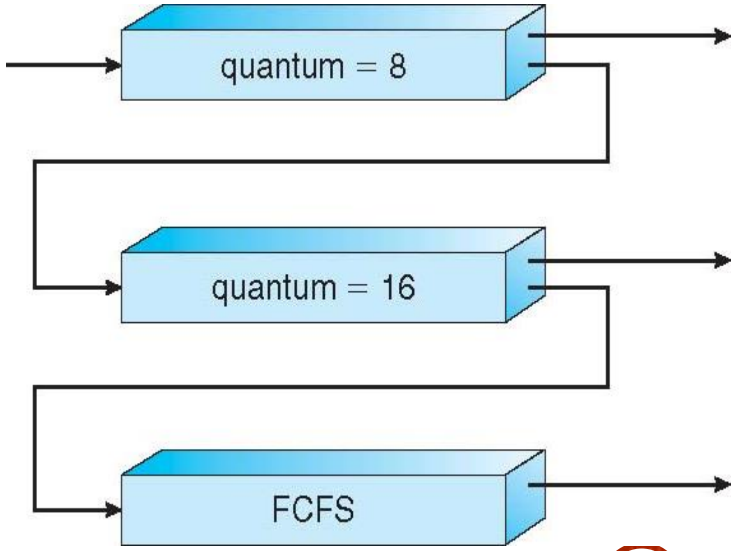
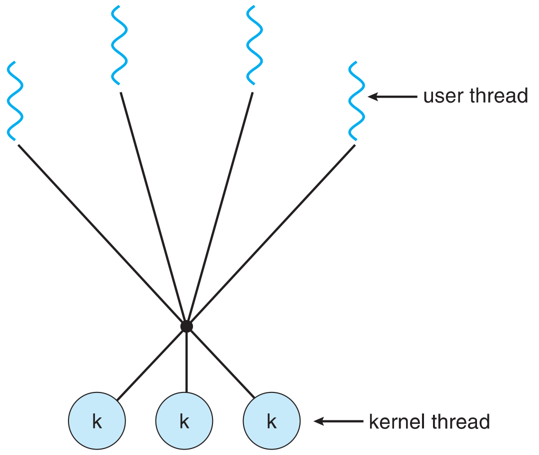


Image from Operating System Concepts.

THREAD SCHEDULING (6.4)

CONTENTION MECHANISMS



- Initially, we need to consider the issue of mapping between user-level threads and the underlying LWPs.
  - If we have a n:m or n:1 model, then we must schedule (map) user-threads to kernel threads within a process: *Process-Contention Scope* (PCS).
  - If we have a 1:1 mode, then we don't need to map anything, since threads already are active in the kernel and can be scheduled onto a CPU: *System-Contention Scope* (SCS).
- When using pthreads, we can easily pick SCS or PCS when setting up thread attributes.
  - See left example.

```
…
pthread_attr_init(&attr);

//fetching scope
if (pthread_attr_getscope(&attr, &scope) != 0)
    fprintf(stderr, "Unable to get scheduling");
else {
    if (scope == PTHREAD_SCOPE_PROCESS)
        printf("PTHREAD_SCOPE_PROCESS");
    else if (scope == PTHREAD_SCOPE_SYSTEM)
        printf("PTHREAD_SCOPE_SYSTEM");
    else
        fprintf(stderr, "Illegal scope value.\n");
}

//setting scope
pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

for (i = 0; i < NUM_THREADS; i++)
    pthread_create(&tid[i], &attr, runner, NULL);

…
```

Image from Operating System Concepts.

MULTIPLE-PROCESSOR SCHEDULING (6.5)

APPROACHES AND AFFINITY

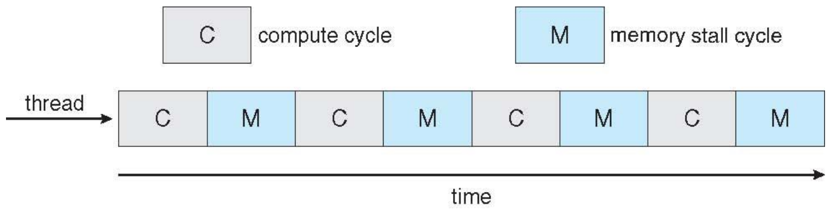
- When we have multiple CPUs in a system, we have a more complicated architecture for assigning work:
  - In *Asymmetric Multi-Processing* (AMP), a single CPU handles task assignment.
  - In *Symmetric Multi-Processing* (SMP), all CPUs are peers in task and resource acquisition.
  - Which of these is (probably) more reliable?

- By default, the scheduler is simply trying to match ready tasks to idle CPUs – no other resource is considered.
- This can be problematic because it is more expensive to move a process between different CPUs than it is to keep it on one. Even if the move only occurs non-preemptively. Why is this?
- By setting *processor affinity*, we can force a process to run on a particular set of CPUs.

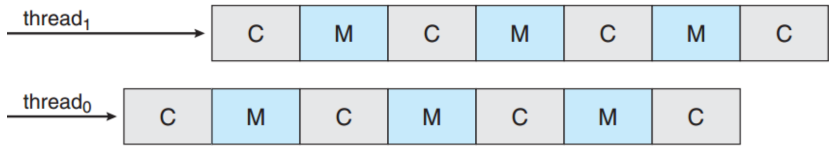
LOAD BALANCING AND MULTITHREAD CORES

- On a SMP architecture, we need to do *load balancing* to make sure all CPUs are active. We don't want to let CPUs idle when there is work that could possibly be done.
  - In general, we care about load balancing when instead of using one global ready queue, there is a individual queue for each processor. Why?
  - When fixing balance, CPUs may either do *push migration* or *pull migration* of processes.

- Consider the execution of a program on a physical core. Sooner or late, that program will request memory outside of the cache and cause a *memory stall*. This means idleness.



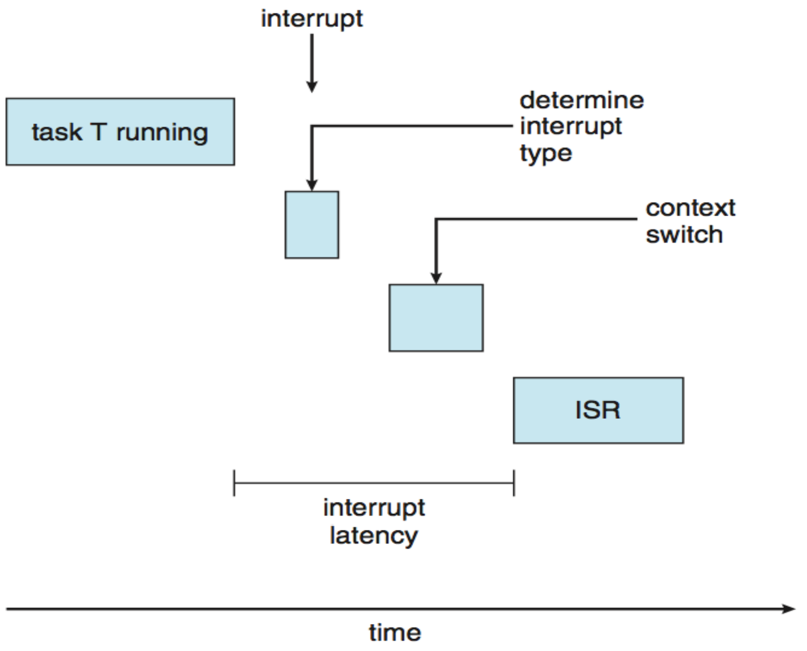
- To address this we can multi-thread the core so that program CPU bursts are interleaved and complementary.



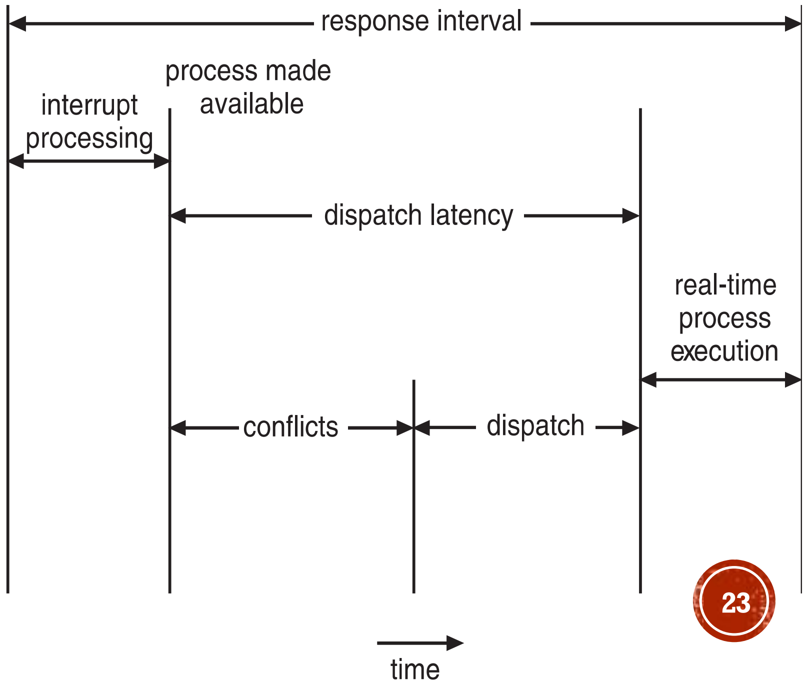
Images from Operating System Concepts.

REAL-TIME CPU SCHEDULING (6.6)

Latency Requirements



- A scheduler that cannot provide a time bound on execution of a specific (critical) process is a *soft real-time system*. A system with a bound is a *hard real-time system*.
  - Any examples of these?
- In a hard system, we need bounds on latency, which is caused by the action of triggering a process.
- The latencies prior to starting a process can be separated into interrupt latency and dispatch latency.



PERIODIC/PRIORITY-BASED SCHEDULING

- Let's redefine our scheduling problem to be a little more regular:
  - For each process, we will say there is a burst time (t), deadline (d), and period (p). For real-time hard scheduling we need to hit the deadline!
  - Let a process's *rate* be $\frac{1}{p}$.
  - Naturally, $0 \leq t \leq d \leq p$. Why?
- What type of process does this model?
- Is it useful/accurate for our real-time discussion?
- As the period (not burst time!) increases, does that mean that the process requires more or less CPU time?

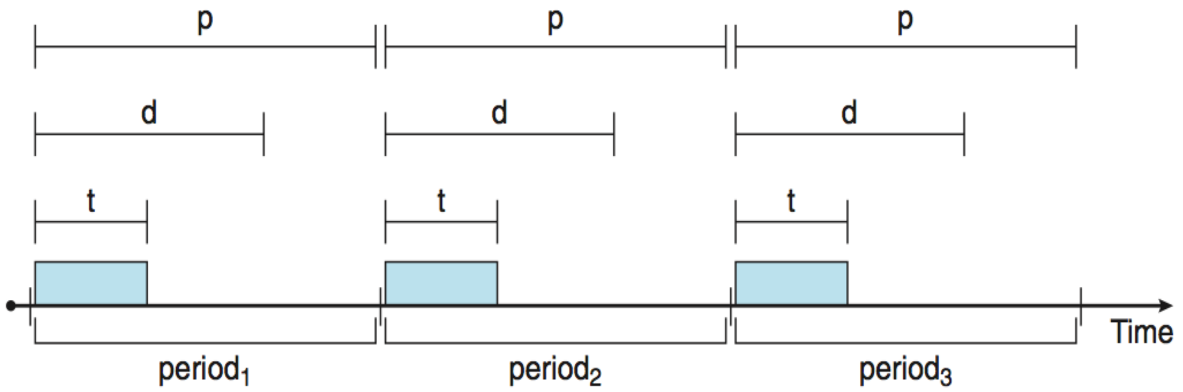
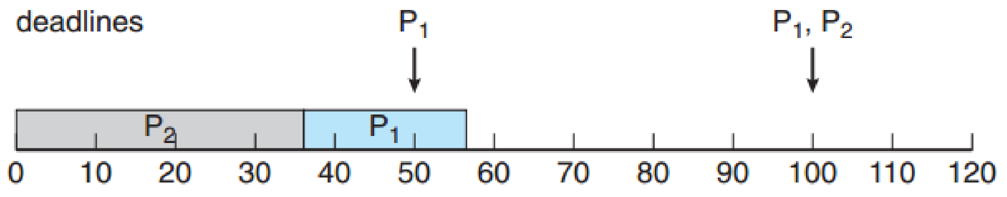


Image from Operating System Concepts.

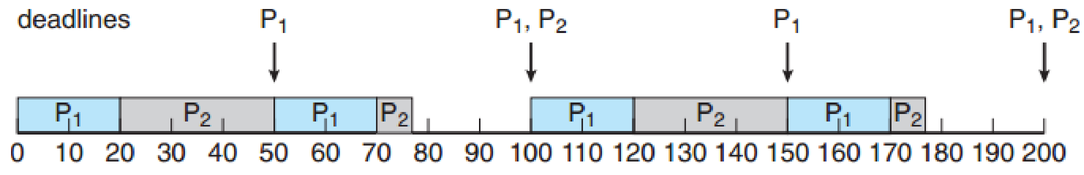
RATE-MONOTONIC SCHEDULING

Last entry from tuple is removed, a separate deadline isn't needed.

- As an example, consider the processes: (P1, 50, 20), (P2, 100, 35). The tuples have the form (process, period, time).
- Is this load possible? Let the CPU utilization of a process be $P_i = \frac{t_i}{p_i}$. $CPU(P1) = .4, CPU(P2) = .35$. Conclusion?
- So what happens if we let P2 have higher priority?



- How about letting P1 have higher priority?

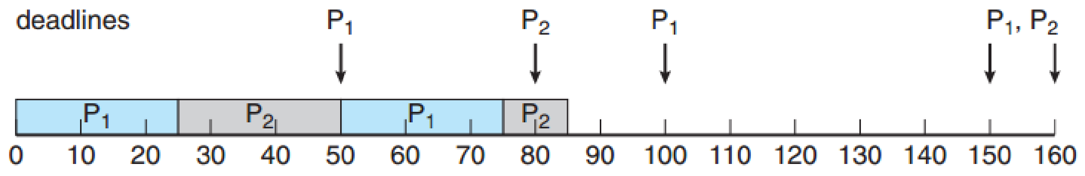


- Can we propose a general rule for priorities?

Images from Operating System Concepts.

RATE-MONOTONIC SCHEDULING

- As another example, consider the processes: (P1, 50, 25), (P2, 80, 35). P1 will have higher priority. CPU utilization for this would be .94.

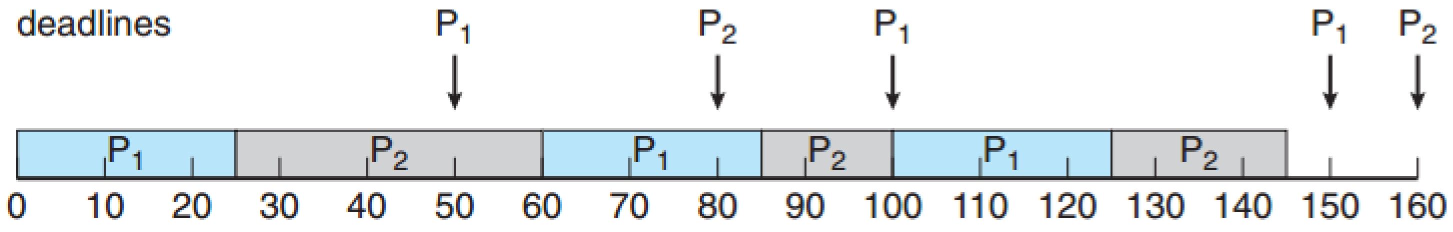


- Why is having a summed CPU utilization less than 1 not enough to guarantee a scheduling solution?
- Provided we know the number of processes that are running (N), then we can compute the actual "max CPU load" as: $N(2^{\frac{1}{N}} - 1)$.

Image from Operating System Concepts.

EARLIEST-DEADLINE-FIRST SCHEDULING

- For this method, processes with an earlier deadline are assigned higher priority.
- As an example, consider the processes: (P1, 50, 25, 50), (P2, 80, 35, 80).



- Advantages of this method are that we don't require processes to be periodic or have a constant burst time. What is the drawback?

Image from Operating System Concepts.

PROPORTIONAL SHARE SCHEDULING

- Last idea is straightforward: for some interval, divide processing time into T shares.
- When a new task runs, then it will request a constant N shares of time for the duration of it's execution. This guarantees a lower bound on process resource availability.
- Some gateway service will have to ensure the availability of shares prior to granting a task's execution.

- When would we use a scheduler like this?

PTHREADS SCHEDULING

- When using pthreads, we can easily pick FIFO or RR when setting up thread attributes.
  - See left example.

```
...
// get the current scheduling policy
if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
    fprintf(stderr, "Unable to get policy.");
else {
    if (policy == SCHED_OTHER) printf("SCHED_OTHER");
    else if (policy == SCHED_RR) printf("SCHED_RR");
    else if (policy == SCHED_FIFO) printf("SCHED_FIFO");
}

// set the scheduling policy - FIFO, RR, or OTHER
if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
    fprintf(stderr, "Unable to set policy.");
 ...
```

OPERATING-SYSTEM EXAMPLES (6.7)

Skipping 6.7.2 (Example: Windows Scheduling)

LINUX

- Since 2.6.x, Linux uses the Completely Fair Scheduler (CFS).
- Works somewhat like a multilevel queue where tasks are sorted into *scheduling classes* (e.g., default, real-time).
  - The next task is based on the highest priority class and the highest priority task within that class.
  - When executed, tasks receive a dynamic time quantum based on their "*nice*ness" and a *targeted latency*. Nice values range from -20 to 19, where lower values receive more CPU time (priority).
    - Can targeted latency be a constant?
  - Priority is indirectly computed from *virtual run time* (vruntime). Vruntime decays over time – lower priority decays faster.
    - Runnable tasks are stored in a balanced BST, keyed by vruntime. Smallest vruntime will be executed next.

LINUX

- Consider submitting two tasks to the scheduler: an I/O bound one and a CPU bound one. Say they have the same vruntime but the CPU one has higher priority (maybe it is interactive).
- Although the CPU task will be executed first, its vruntime will increase and then the I/O bound task will be able to run.

- As mentioned previously, we can manually pick scheduling in pthreads (SCHED_RR, SCHED_FIFO) for real-time tasks. Threads scheduled in this way will end up in the real-time class of tasks and have a higher priority than normal "nice" threads.

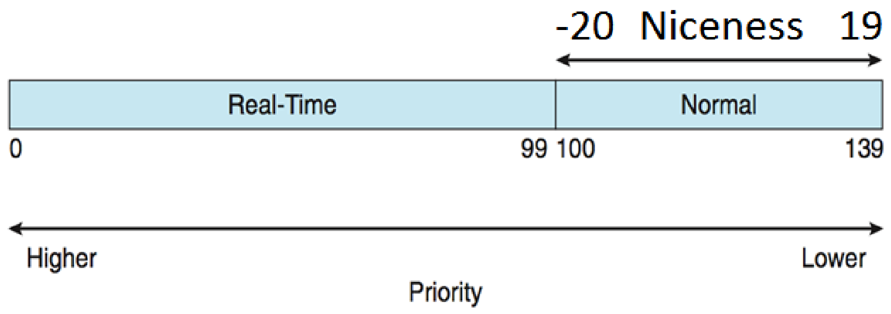
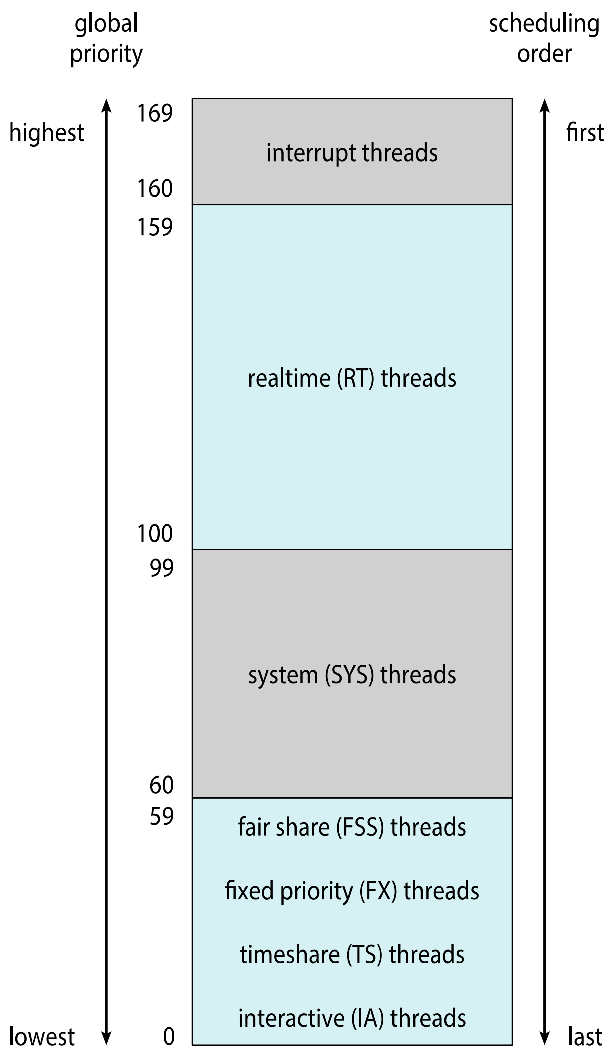


Image from Operating System Concepts.

SOLARIS

- Again a class based scheduler: time sharing (TS), interactive (IA), real time (RT), system (SYS), fair share (FSS), fixed priority (FP).
- Most threads go into TS, which is set up like a feedback queue. Higher priority processes are given smaller time slices… why?
- On the right, is a projection of these classes into a global space.
  - If multiple threads have the same priority, then they are served RR.



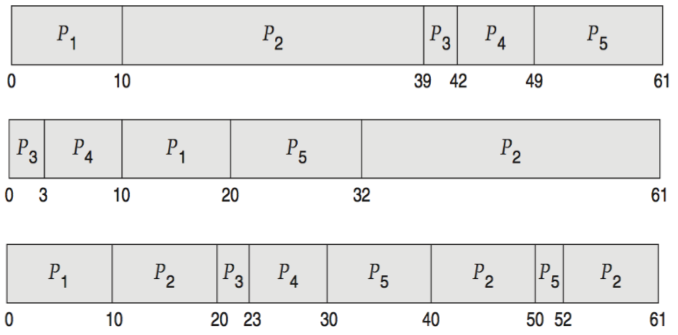
Visual size of regions does not correspond to how many threads will be there!

Image from Operating System Concepts.

ALGORITHM EVALUATION (6.8)

DETERMINISTIC EVALUATIONS AND QUEUEING MODELS

- The simplest way to evaluate an algorithm is to feed it some set of predetermined processes with burst times, priority information, etc. The book calls this *deterministic modeling*.
  - Example: FCS, SFJ, RR (top to bottom at right).



P_1 P_2 P_3 P_4 P_5
0 10 39 42 49 61
P_3 P_4 P_1 P_5 P_2
0 3 10 20 32 61
P_1 P_2 P_3 P_4 P_5 P_2 P_5 P_2
0 10 20 23 30 40 50 52 61

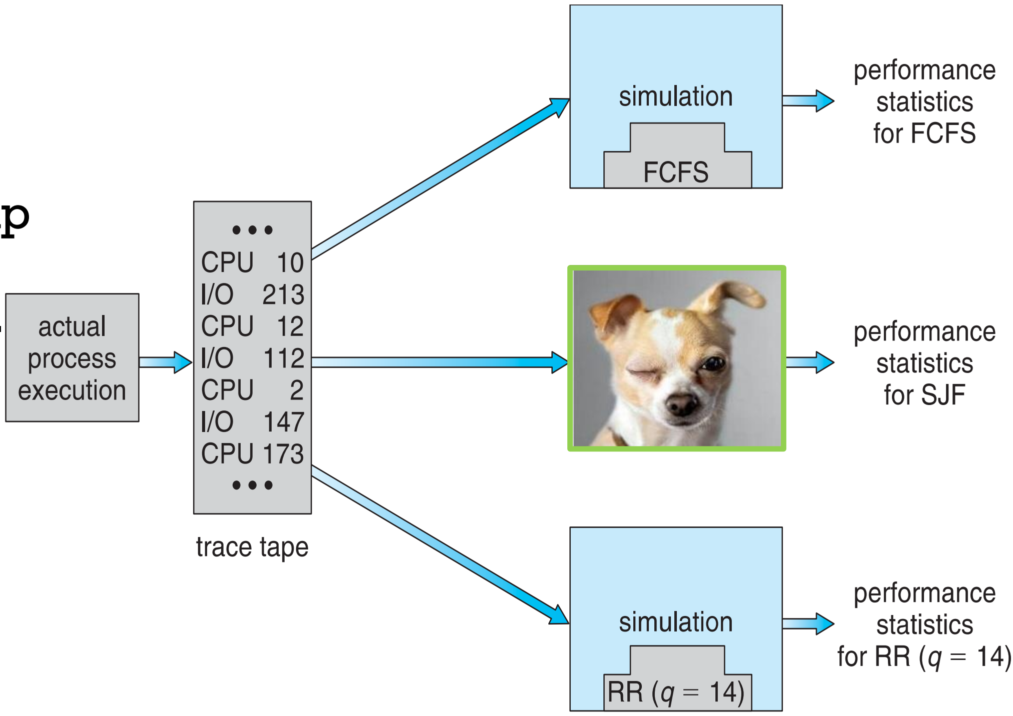
- Then, we can compute whatever property we care about (waiting time, completion time, etc.)

- Sometimes we can take the approach of making some assumption of the distribution of data as it passes through a ready/waiting queue.

Image from Operating System Concepts.

SIMULATION

- We could also consider setting up an system to do some event simulation of a real environment.
- The events can come from random generation, certain distributions, etc.
- Ideally, we would trace the execution of an actual machine and use that as a benchmark.



- The textbook has a comment, for whatever reason, that talks about not manipulating the scheduler of a production environment…

Image from Operating System Concepts.