Scheduling

2 January 2025

Lecture 9

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

Notify, broadcast	Barrier Synchronization	Readers/Writers
Java synchronized	Scheduler	FIFO
Round Robin	Quantum	

Topics for Today

- Scheduling
  - Simple Algorithms: Priority
  - Prediction Based: SJF, SRTF
  - Advanced: Responsive, Lottery
- Scheduling Case Studies
  - Linux O(1) Scheduler
  - Linux Completely Fair Scheduler (CFS)
- Real Time Scheduling

Today’s concepts

Handling differences in importance: Strict Priority Scheduling

Priority 3	$\longrightarrow$	Job 1	$\longrightarrow$	Job 2	$\longrightarrow$	Job 3
Priority 2	$\longrightarrow$	Job 4				
Priority 1	$\longrightarrow$					
Priority 0	$\longrightarrow$	Job 5	$\longrightarrow$	Job 6	$\longrightarrow$	Job 7

- Execution Plan
  - Always execute highest-priority runnable jobs to completion
  - Each queue can be processed in Round-Robin fashion with some time-quantum
- Problems:
  - Starvation: Lower priority jobs don't get to run because higher priority tasks always running

Handling differences in importance: Strict Priority Scheduling

Priority 3	⟶	Job 1	⟶	Job 2	⟶	Job 3
Priority 2	⟶	Job 4				
Priority 1	⟶					
Priority 0	⟶	Job 5	⟶	Job 6	⟶	Job 7

- Deadlock: Priority Inversion
  - Not strictly a problem with priority scheduling, but happens when low-priority task has lock needed by high-priority task
  - Usually involves third, intermediate priority task that keeps running even though high-priority task should be running
- How to fix problems?
  - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc.

So Far

- Scheduling
  - Simple Algorithms: Priority
  - Prediction Based: SJF, SRTF
  - Advanced: Responsive, Lottery
- Scheduling Case Studies
  - Linux O(1) Scheduler
  - Linux Completely Fair Scheduler (CFS)
- Real Time Scheduling

What if we Knew the Future?

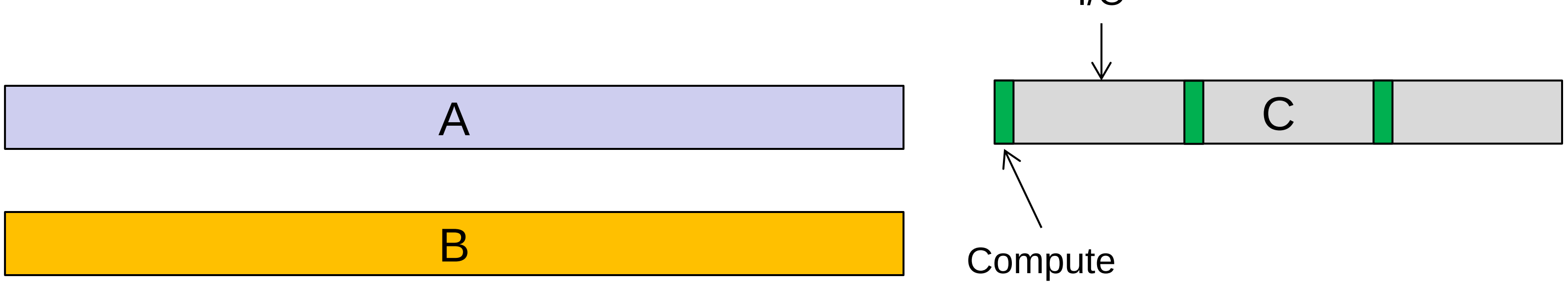
- Could we always mirror best FCFS?
- Shortest Job First (SJF):
  - Run whatever job has the least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF):
  - Preemptive version of SJF: If job arrives and has a shorter time to completion than the remaining time on the current job, immediately preempt CPU
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied either to a whole program or the current CPU burst of each program
  - Idea is to get short jobs out of the system
  - Big effect on short jobs, only small effect on long ones
  - Result is better average response time

Discussion

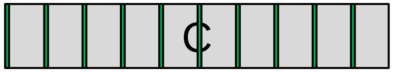
- SJF/SRTF are the best you can do at minimizing average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, focus on SRTF
- Comparison of SRTF with FCFS and RR
  - What if all jobs the same length?
    - SRTF becomes the same as FCFS (i.e. FCFS is best can do if all jobs the same length)
  - What if jobs have varying length?
    - SRTF (and RR): short jobs not stuck behind long ones

Example to illustrate benefits of SRTF

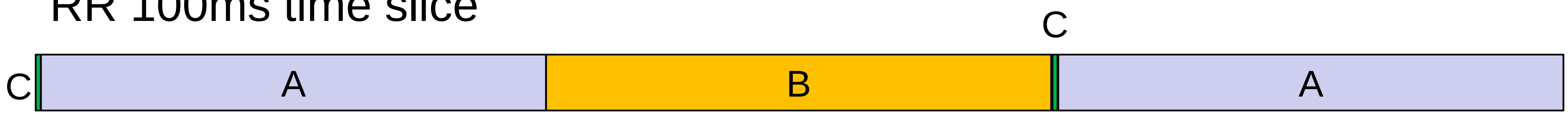
- Three jobs:
  - A and B: Both CPU bound, run for one week
  - C: I/O bound, loop 1ms CPU, 9ms disk I/O
  - If only one at a time, C uses 90% of the disk, A or B could use 100% of the CPU
- With FIFO:
  - Once A or B get in, keep CPU for two weeks
- What about RR or SRTF?
  - Easier to see with a timeline



SRTF Example



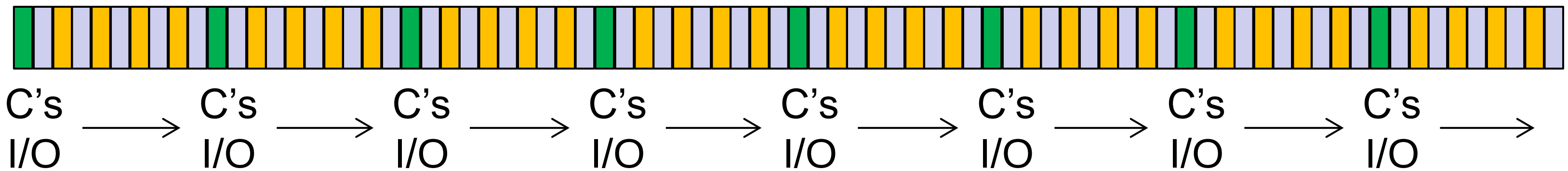
RR 100ms time slice



C starts I/O → C starts I/O →

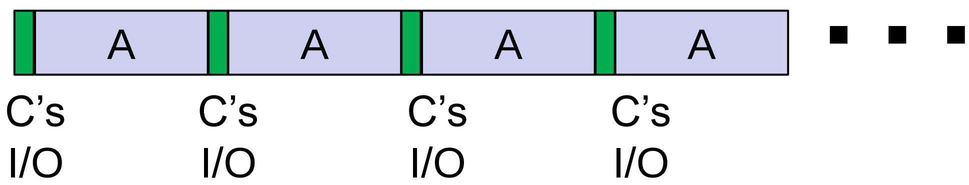
Disk utilization: $\frac{9}{201} \approx 4.5\%$

RR 1ms time slice



Disk utilization: $\approx 90\%$, but lots of context switch overhead!

SRTF



Disk utilization: $\approx 90\%$

SRTF Discussion

- Starvation: SRTF leads to starvation if many small jobs!
  – Large jobs never get to run

Somehow need to predict the future

- How do we do this?
- Some systems ask the user
  – When you submit a job, have to say how long it will take
  – To stop cheating, system kills job if takes too long
  – But: Even non-malicious users have trouble predicting runtime of their jobs

SRTF Discussion

- Bottom line, we can't really know how long job will take
  - However, we can use SRTF as a yardstick to compare to other policies
  - Optimal, so can't do any better

- SRTF Pros & Cons
  - Optimal (average response time) ()
  - Hard to predict future ()
  - Unfair ()

So Far

- Scheduling
  - Simple Algorithms: Priority
  - Prediction Based: SJF, SRTF
  - Advanced: Responsive, Lottery
- Scheduling Case Studies
  - Linux O(1) Scheduler
  - Linux Completely Fair Scheduler (CFS)
- Real Time Scheduling

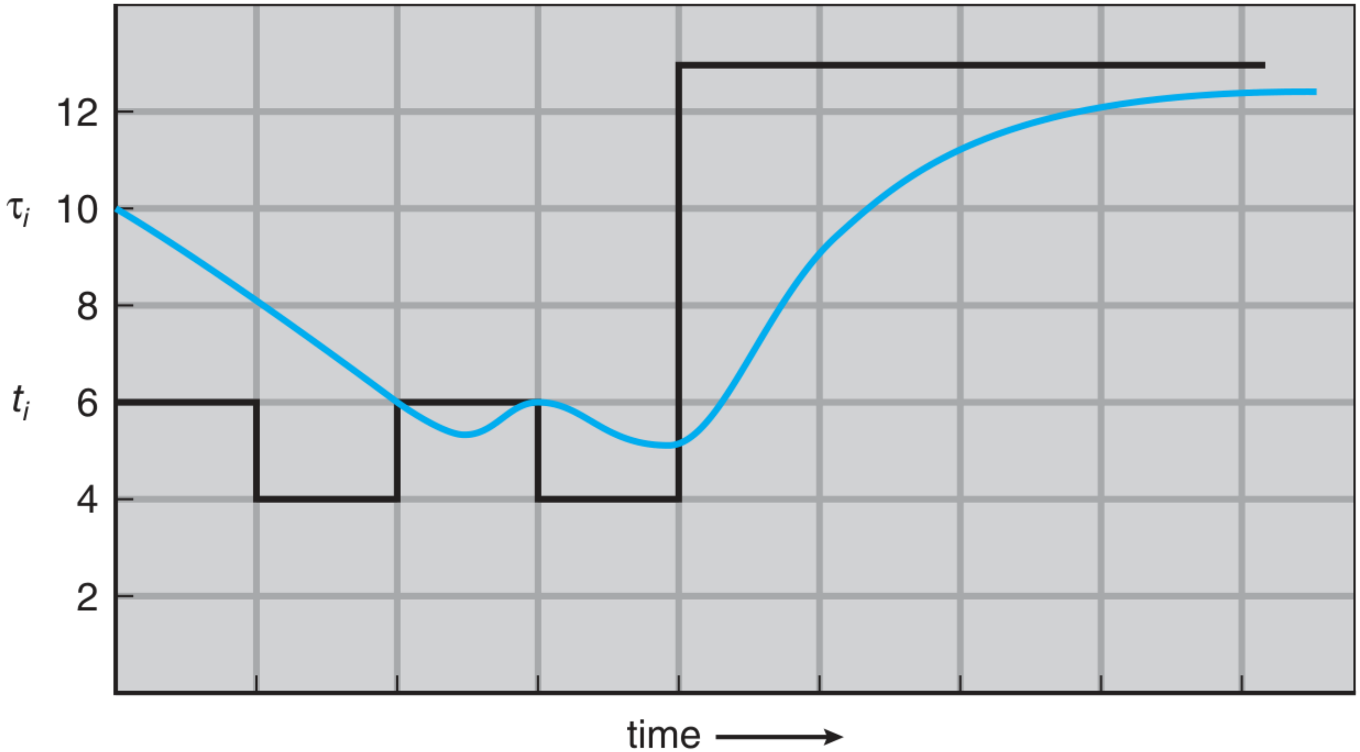
Predicting the Length of the Next CPU Burst

- Adaptive: Changing policy based on past behavior
  - CPU scheduling, in virtual memory, in file systems, etc.
  - Works because programs have predictable behavior
    - If program was I/O bound in past, likely in future
    - If computer behavior were random, wouldn't help
- Example: SRTF with estimated burst length
  - Use an estimator function on previous bursts:
    Let $t_{n-1}, t_{n-2}, t_{n-3}$ etc. be previous CPU burst lengths. Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \dots)$
  - Function f could be one of many different time series estimation schemes (Kalman filters, etc)
  - For instance: Exponential Averaging:

$$\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1} \text{ with } 0 < \alpha \leq 1$$

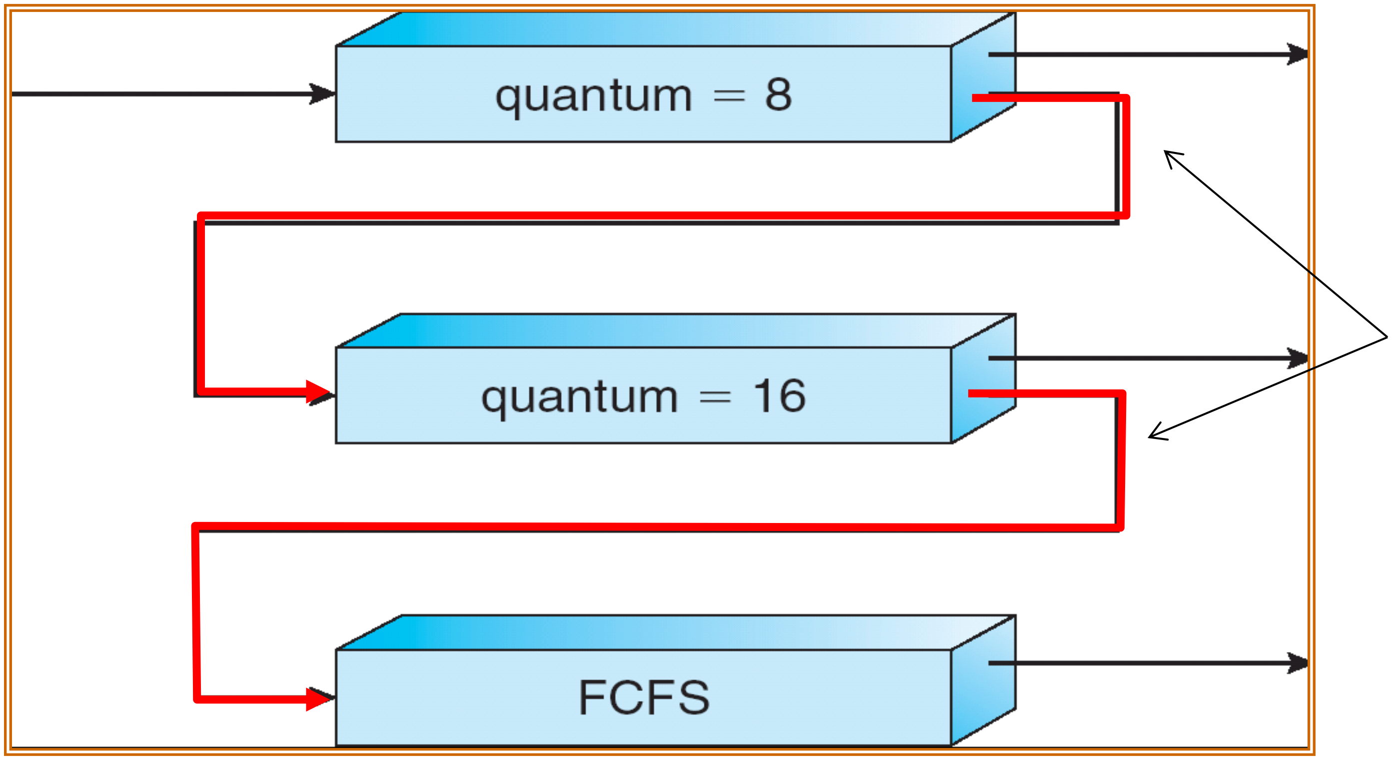
Exponential Average Example

$$\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1} \text{ with } 0 < \alpha \leq 1$$



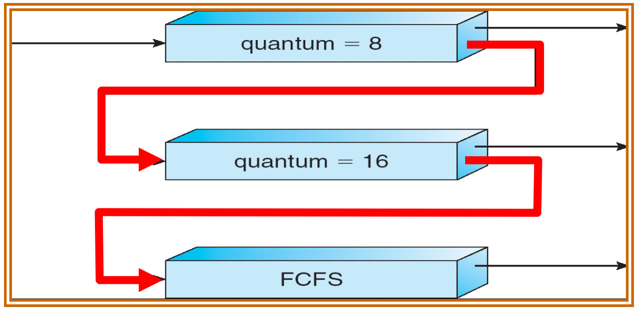
CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Multi-Level Feedback Scheduling



Long-Running Compute Tasks Demoted to Low Priority

Multi-Level Feedback Scheduling

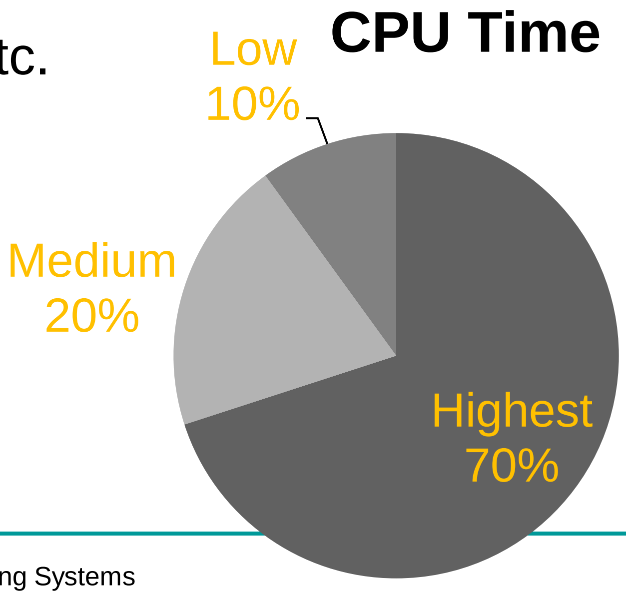


Long-Running Compute Tasks Demoted to Low Priority

- A method for exploiting past behavior
  - First used in CTSS (MIT 1961)
  - Multiple queues, each with different priority
    - Higher priority queues often considered “foreground” tasks
- Each queue has its own scheduling algorithm
  - Ex. foreground – RR, background – FCFS
  - Sometimes multiple RR priorities with quantum increasing exponentially (highest: 1ms, next: 2ms, next: 4ms, etc)
- Adjust each job’s priority as follows (details vary)
  - Job starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn’t expire, push up one level (or to top)

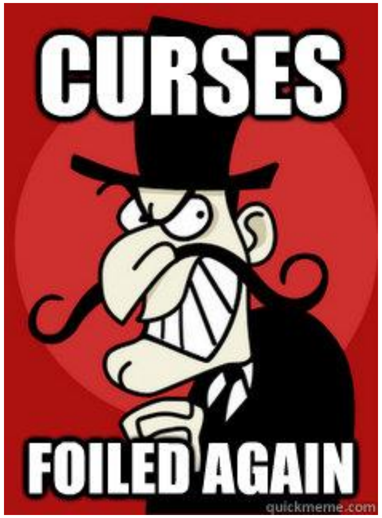
Scheduling Details

- Result approximates SRTF:
  - CPU bound jobs drop like a rock
  - Short-running I/O bound jobs stay near top
- Scheduling must be done between the queues
  - Fixed priority scheduling:
    - Serve all from highest priority, then next priority, etc.
  - Time slice:
    - Each queue gets a certain amount of CPU time:



Scheduling Details

- Countermeasure: User action that can foil intent of the OS designer
  - For multilevel feedback, put in a bunch of meaningless I/O to keep job's priority high
  - If everyone did this, it wouldn't work!



- Example of Othello program:
  - Playing against competitor, so key was to do computing at higher priority than competitors.
    - Put in `printf` → ran much faster!

Image sources: quickmeme.com, amazon.com

Scheduling Fairness

- What about fairness?
  - Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc):
    - Long running jobs may never get CPU
    - In Multics, shut down machine, found 10-year-old job
  - Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run

Tradeoff: Fairness gained by hurting average response time!

How to implement fairness?

Could give each queue some fraction of the CPU

- What if one long-running job and 100 short-running ones?
- Like express lanes in a supermarket—sometimes express lanes get so long, get better service by going into one of the other lines

Could increase priority of jobs that don't get service

- What is done in UNIX
- This is ad hoc— at what rate should you increase priorities?
- As system gets overloaded, no job gets CPU time, so everyone increases in priority $\Rightarrow$ Interactive jobs suffer

Lottery Scheduling

Basics

- Give each job some number of lottery tickets
- On each time slice, randomly pick a winning ticket
- On average, CPU time is proportional to number of tickets given to each job

How to assign tickets?

- To approximate SRTF, short running jobs get more, long running jobs get fewer
- To avoid starvation, every job gets at least one ticket (everyone makes progress)

Lottery Scheduling

- Advantage over strict priority scheduling: behaves gracefully as load changes
  - Adding or deleting a job affects all jobs proportionally, independent of how many tickets each job possesses

Lottery Scheduling Example

- Assume short jobs get 10 tickets, long jobs get 1 ticket
- What if too many short jobs to give reasonable response time?
  - In UNIX, if load average is 100, hard to make progress
  - One approach: log some user out

# Short Jobs/ # Long Jobs	% of CPU each Short Job Gets	% of CPU each Long Job Gets
1/1	91%	9%
0/2	N/A	50%
2/0	50%	N/A
10/1	9.9%	0.99%
1/10	50%	5%

So Far

- Scheduling
  - Simple Algorithms: Priority
  - Prediction Based: SJF, SRTF
  - Advanced: Responsive, Lottery
- Scheduling Case Studies
  - Linux O(1) Scheduler
  - Linux Completely Fair Scheduler (CFS)
- Real Time Scheduling

How to Evaluate a Scheduling algorithm?

Deterministic modeling

- Takes a predetermined workload and compute the performance of each algorithm for that workload

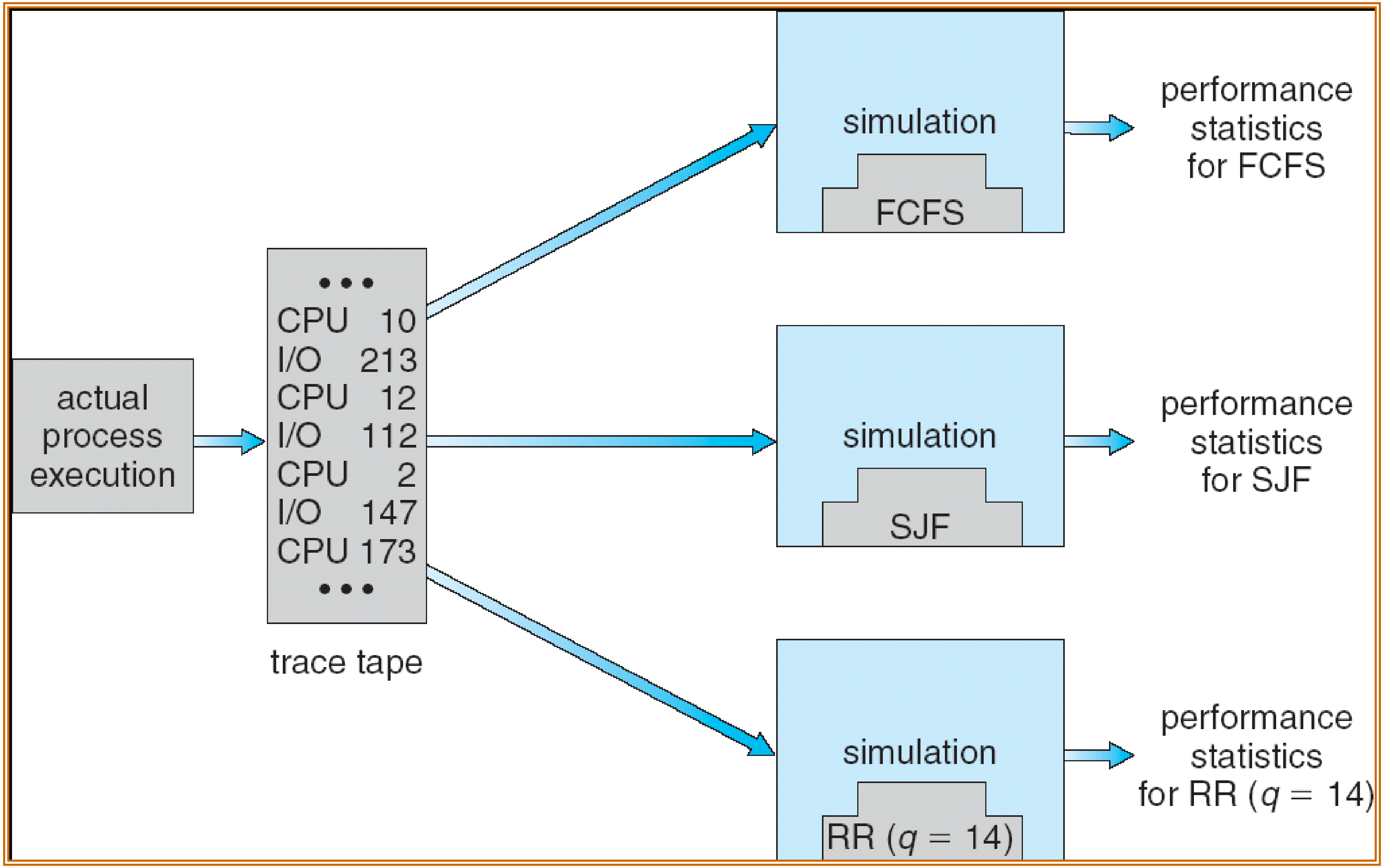
Queueing models

- Mathematical approach for handling stochastic workloads

Implementation/Simulation:

- Build system which allows actual algorithms to be run against actual data
- Most flexible/general.

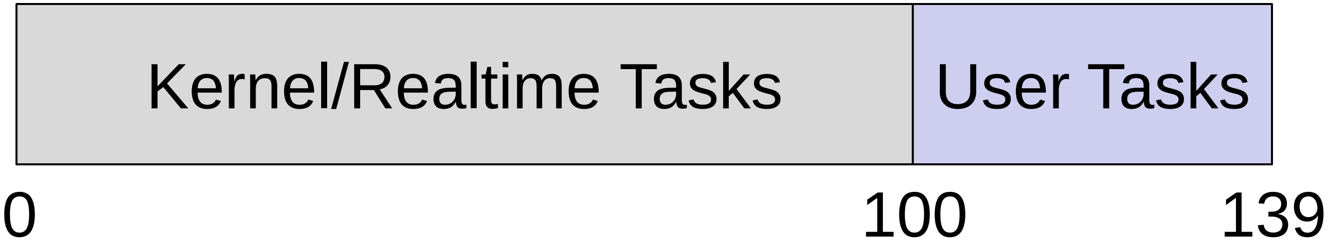
Implementation/Simulation



Case Study: Linux O(1) Scheduler

- Priority-based scheduler: 140 priorities
  - 40 for “user tasks” (set by “nice”), 100 for “Realtime/Kernel”
  - Lower priority value $\Rightarrow$ higher priority (for nice values)
  - Highest priority value $\Rightarrow$ Lower priority (for realtime values)
- All algorithms O(1)
  - Timeslices/priorities/interactivity credits all computed when job finishes time slice
  - 140-bit bit mask indicates presence or absence of job at given priority level

O(1) Priorities



Case Study: Linux O(1) Scheduler

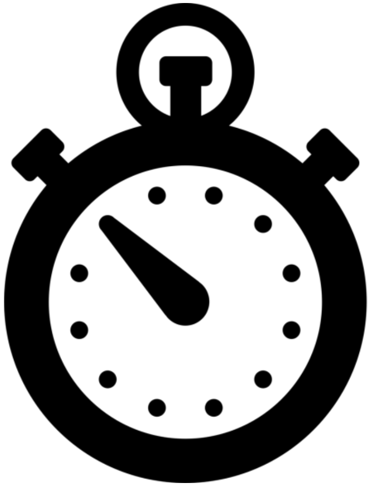
- Two separate priority queues: “active” and “expired”
  - All tasks in the active queue use up their timeslices and get placed on the expired queue, after which queues swapped
- Timeslice depends on priority – linearly mapped onto timeslice range
  - Like a multi-level queue (one queue per priority) with different timeslice at each level
  - Execution split into “Timeslice Granularity” chunks – round robin through priority

O(1) Heuristics

- User-task priority adjusted ± 5 based on heuristics
  - `p->sleep_avg = sleep_time – run_time`
  - Higher `sleep_avg` $\Rightarrow$ more I/O bound the task, more reward (and vice versa)
- Interactive Credit (IC)
  - Earned when a task sleeps for a “long” time
  - Spend when a task runs for a “long” time
  - IC is used to provide history-based continuity to avoid changing interactivity for temporary changes in behavior
- However, “interactive tasks” get special dispensation
  - To try to maintain interactivity
  - Placed back into active queue, unless some other task has been starved for too long…

O(1) and Real-Time Tasks

- Always preempt non-Real-Time tasks
- No dynamic adjustment of priorities
- Scheduling schemes:
  - `SCHED_FIFO:` preempts other tasks, no timeslice limit
  - `SCHED_RR:` preempts normal tasks, RR scheduling amongst tasks of same priority



Linux Completely Fair Scheduler (CFS)

- First appeared in 2.6.23 (Oct 2007), modified in 2.6.38 (Nov 2010)

- “CFS doesn't track sleeping time and doesn't use heuristics to identify interactive tasks—it just makes sure every process gets a fair share of CPU within a set amount of time given the number of runnable processes on the CPU.”

Linux Completely Fair Scheduler (CFS)

- Inspired by Networking "Fair Queueing"
  - Each process given their fair share of resources
  - Models an "ideal multitasking processor" in which N processes execute simultaneously as if they truly got $1/N$ of the processor
    - Tries to give each process an equal fraction of the processor

- Priorities reflected by weights such that increasing a task's priority by 1 always gives the same fractional increase in CPU time – regardless of current priority

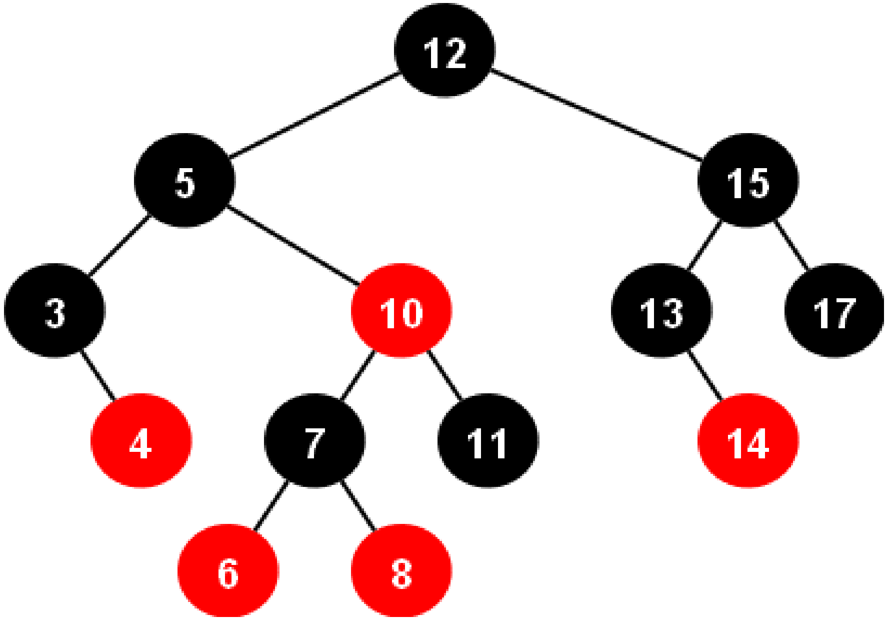
Linux Completely Fair Scheduler (CFS)

- Idea: Track amount of "virtual time" received by each process when it is executing
  - Take real execution time, scale by weighting factor
    - Lower priority $\Rightarrow$ real time divided by greater weight
    - Actually – multiply by $\frac{sum\ of\ all\ weights}{current\ weight}$
  - Keep virtual time advancing at same rate

- Targeted latency ($\boldsymbol{T_L}$): period of time after which all processes get to run at least a little
  - Each process runs with quantum $\left(W_p / \sum W_i\right) \times T_L$
  - Never smaller than "minimum granularity"

Linux Completely Fair Scheduler (CFS)

- Use of Red-Black tree to hold all runnable processes as sorted on `vruntime` variable
  - $O(\log n)$ time to perform insertions/deletions
    - Cache the item at far left (item with earliest `vruntime`)
  - When ready to schedule, grab job with the smallest `vruntime` (which will be item at the far left).



CFS Examples

- Targeted latency $= 20ms$, Minimum Granularity $= 1ms$

Two CPU bound tasks with same priorities

- Both switch with $10ms$

Two CPU bound tasks separated by `nice` value of 5

- One task gets $5ms$, another gets $15ms$

40 tasks:

- Each gets $1ms$ (no longer totally fair)

CFS Examples

One CPU bound task, one interactive task same priority

- While interactive task sleeps, CPU bound task runs and increments `vruntime`
- When interactive task wakes up, runs immediately, since it is behind on `vruntime`

CFS Update

- Group scheduling facilities (2.6.38)
  - Can give fair fractions to groups (like a user or other mechanism for grouping processes)
  - So, two users, one starts 1 process, other starts 40, each will get 50% of CPU

So Far

- Scheduling
  - Simple Algorithms: Priority
  - Prediction Based: SJF, SRTF
  - Advanced: Responsive, Lottery
- Scheduling Case Studies
  - Linux O(1) Scheduler
  - Linux Completely Fair Scheduler (CFS)
- Real Time Scheduling

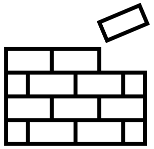
Real-Time Scheduling (RTS)

- Efficiency is important but predictability is essential:
  - We need to be able to predict with confidence the worst case response times for systems
  - In RTS, performance guarantees are:
    - Task- and/or class centric
    - Often ensured a priori
- In conventional systems, performance is:
  - System oriented and often throughput oriented
  - Post-processing (… wait and see …)

- Real-time is about enforcing predictability and is not the same at fast computing!!!

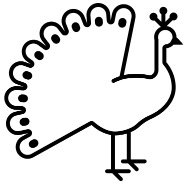
Real-Time Scheduling (RTS)

Hard Real-Time



- Attempt to meet all deadlines
- EDF (Earliest Deadline First), LLF (Least Laxity First), RMS (Rate-Monotonic Scheduling), DM (Deadline Monotonic Scheduling)

Soft Real-Time

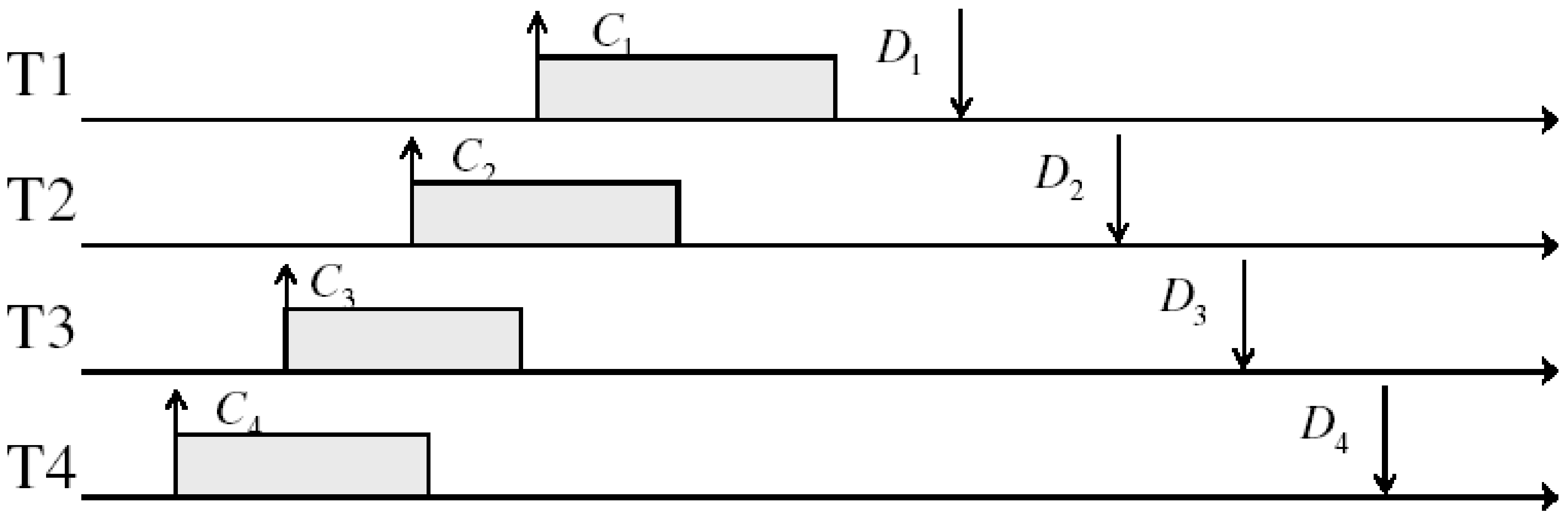


- Attempt to meet deadlines with high probability
- Minimize miss ratio / maximize completion ratio (firm real-time)
- Important for multimedia applications
- CBS (Constant Bandwidth Server)

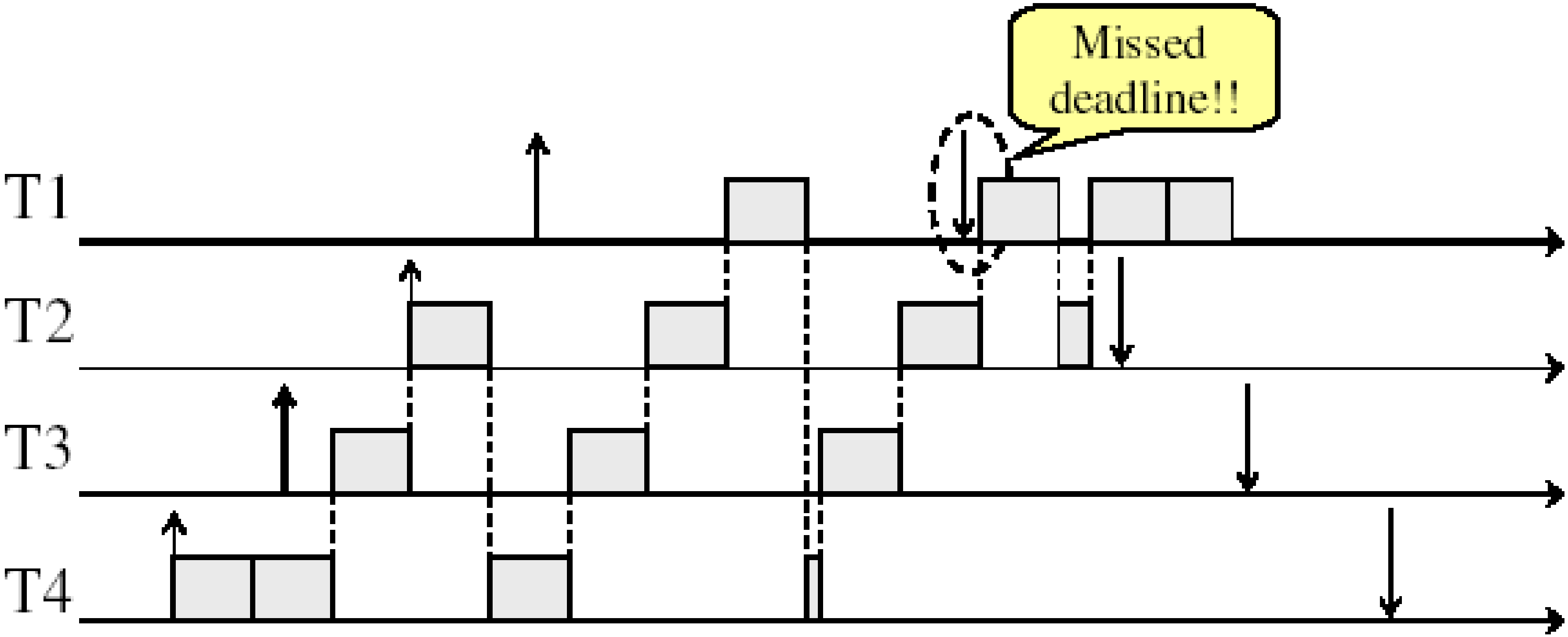
Example: Workload Characteristics

- Tasks are preemptable, independent with arbitrary arrival (=release) times
- Times have deadlines (D) and known computation times (C)

Example Setup:

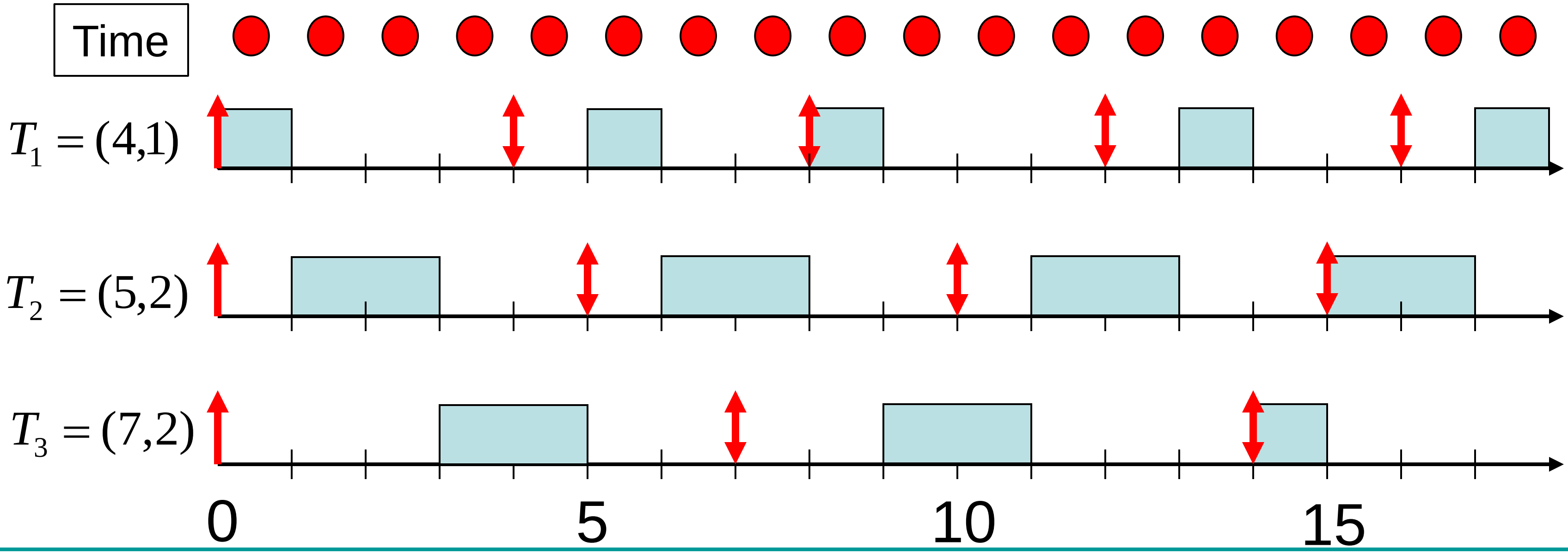


Example: Using Round Robin



Earliest Deadline First (EDF)

- Tasks periodic with period P and computation C in each period: (P, C)
- Preemptive priority-based dynamic scheduling
- Each task is assigned a (current) priority based on how close the absolute deadline is.
- The scheduler always schedules the active task with the closest absolute deadline.



EDF: Schedulability Test

Theorem (Utilization-based Schedulability Test):

A task set $T_1, T_2, \ldots, T_n$ with $D_i = P_i$ is schedulable by the Earliest Deadline First (EDF) scheduling algorithm if

$$\sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq 1$$

Exact schedulability test (necessary + sufficient)

Proof: [Liu and Layland, 1973]

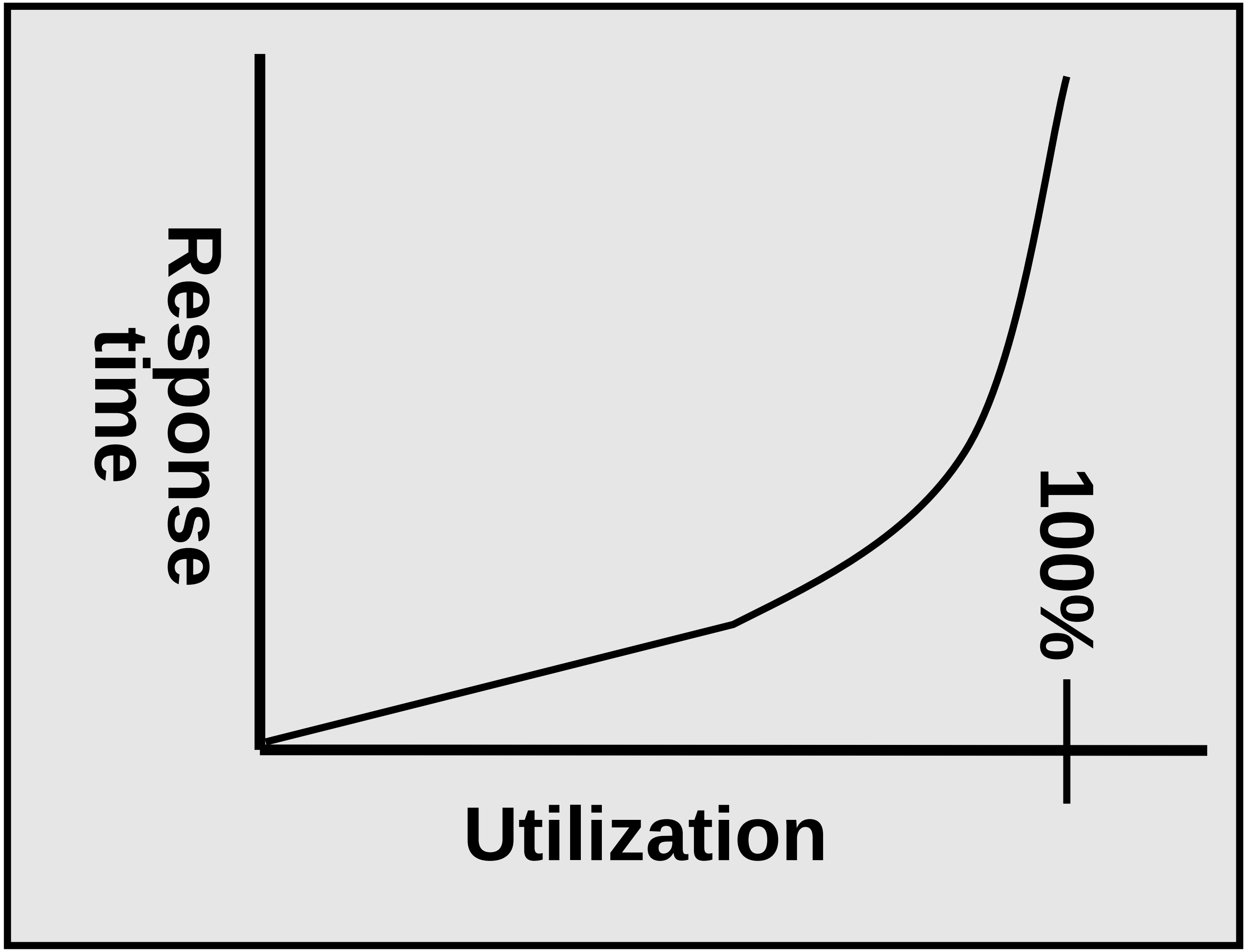
A Final Word On Scheduling

- When do the details of the scheduling policy and fairness really matter? When there aren't enough resources to go around

- When should you buy a faster computer (or network link, etc)?
  - One approach: When it will pay for itself in improved response time
  - Assuming you're paying for worse response time in reduced productivity, customer angst, etc.

- Maybe you should buy a faster X when X is utilized 100%, but usually response time goes to ∞ as utilization goes to 100%

The Problem



And so…

- An interesting implication of the curve:
  – Most scheduling algorithms work well in the “linear” portion of the load curve, fail otherwise
  – Argues for buying a faster X when hit “knee” of curve

Conclusion

- Scheduling
  - Simple Algorithms: Priority
  - Prediction Based: SJF, SRTF
  - Advanced: Responsive, Lottery
- Scheduling Case Studies
  - Linux O(1) Scheduler
  - Linux Completely Fair Scheduler (CFS)
- Real Time Scheduling