SE350: Operating Systems

Lecture 8: Scheduling

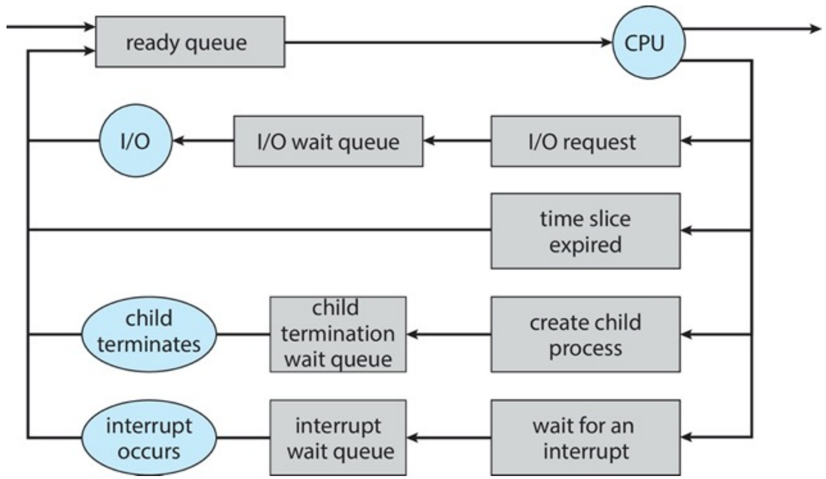
Outline

- Definitions
  - Response time, throughput, scheduling policy, …
- Uniprocessor policies
  - FCFS, SJF/SRTF, Round Robin, …
  - Real-time scheduling
- Multiprocessor policies
  - Oblivious scheduling, gang scheduling, …

Definitions

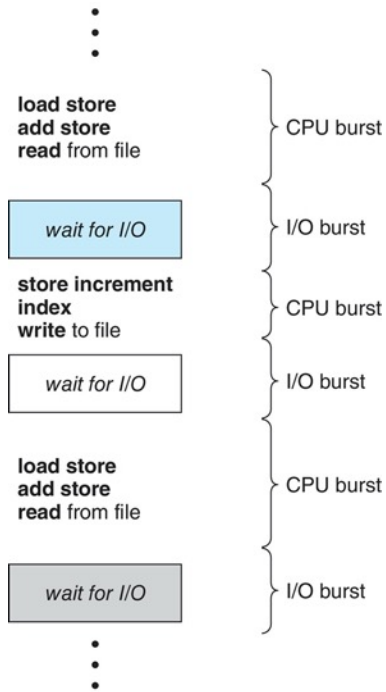
- Task
  - User request (e.g., mouse click, web request, shell command, etc.)
- Workload
  - Set of tasks for system to perform
- Scheduling algorithm
  - Takes workload as input, decides which tasks to do first
- Overhead
  - How much extra work is done by scheduler?
- Preemptive scheduler
  - If we can take resources away from a running task
- Work-conserving
  - Resources are used whenever there is task to run
  - For non-preemptive schedulers, work-conserving is not always better
- Only preemptive, work-conserving schedulers to be considered in this lecture!

Recall: CPU Scheduling

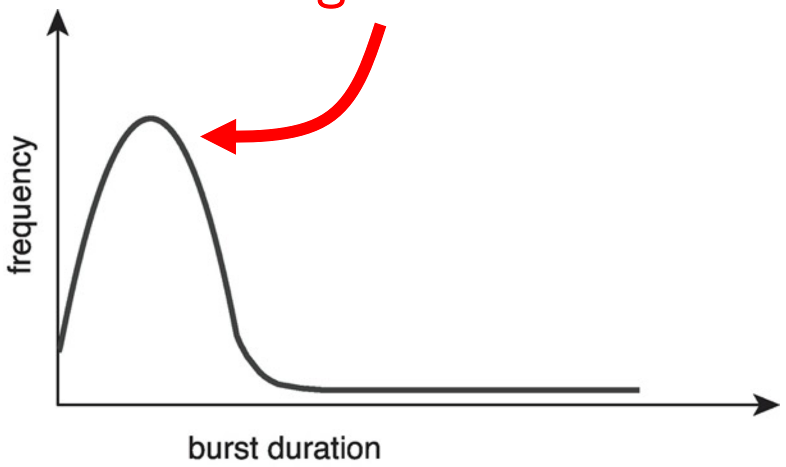


- Earlier, we talked about life-cycle of threads
  - Threads work their way from ready to running to various waiting queues
- Question: How does OS decide which thread to dequeue?
  - Obvious queue to worry about is ready queue
  - Others can be scheduled as well, however
- Scheduling: Deciding which thread gets resource from moment to moment

Execution Model



Weighted toward small bursts



- Programs alternate between bursts of CPU and I/O
  - Use CPU for some period, then do I/O, then use CPU again, etc.
- CPU scheduling is about choosing thread which gets CPU for its next CPU burst
- With preemption, thread may be forced to give up CPU before finishing its burst

CPU Scheduling Assumptions

- There are many implicit assumptions for CPU scheduling
  - One program per user
  - One thread per program
  - Programs are independent

- These may not hold in all systems, but they simplify the problem

- High-level goal is to divide CPU time to optimize some desired properties

CPU Scheduling Policy Goals/Criteria

- Minimize average response time
  - Minimize elapsed time to do an operation (or task)
  - Response time is what users see
    - Time to echo a keystroke in editor
    - Time to compile a program
    - Real-time tasks must meet *deadlines* imposed by "environment"

CPU Scheduling Policy Goals/Criteria (cont.)

- Maximize throughput
  - Maximize operations (or tasks) per time unit (e.g., second)
  - Throughput related to response time, but not identical
    - Minimizing response time could lead to more context switching which will than hurt throughput (more on this later!)
  - Two parts to maximizing throughput
    - Minimize overhead (e.g., context-switching)
    - Efficient use of resources (e.g., CPU, disk, memory, etc.)

CPU Scheduling Policy Goals/Criteria (cont.)

- Achieve fairness
  - Share CPU time among *users* in some *equitable* way
  - What does equitable mean?
    - Equal share of CPU time?
      - What if some tasks don't need their full share?
    - Minimize variance in worst case performance?
      - What if some tasks were running when no one else was running?
  - Who are users? Actual users or programs?
    - If A runs one thread and B runs five, B could get five times as much CPU time on many OS's
  - Fairness is not minimizing average response time
    - Improving average response time could make system less fair (more on this later!)

Outline

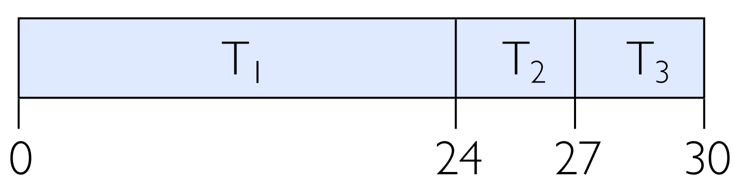
- Definitions
  - Response time, throughput, scheduling policy, …
- Uniprocessor policies
  - FCFS, SJF/SRTF, Round Robin, …
  - Real-time scheduling
- Multiprocessor policies
  - Oblivious scheduling, gang scheduling, …

First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS)
  - Also "First In, First Out" (FIFO)
  - In early systems, FCFS meant one program scheduled until done (including its I/O activities)
  - Now, it means that program keeps CPU until the end of its CPU burst
- Example:

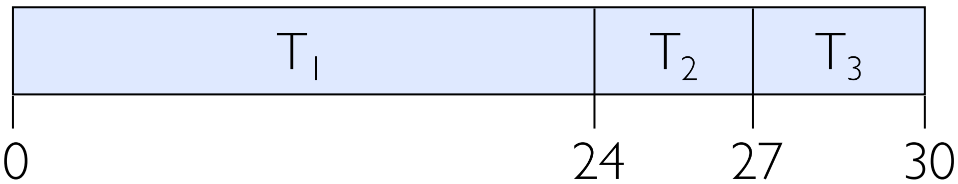
Thread	CPU Burst Time
T_1	24
T_2	3
T_3	3

  - Suppose threads arrive in order: T_1, T_2, T_3
    The Gantt Chart for FCFS scheduling is



FCFS Scheduling (cont.)

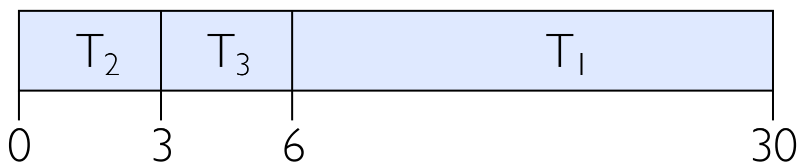
- Example continued:



  - Waiting time for T_1 is **0**, for T_2 is **24**, and for T_3 is **27**
  - Average waiting time is **(0 + 24 + 27)/3 = 17**
  - Average response time is **(24 + 27 + 30)/3 = 27**
- Convoy effect: Short threads get stuck behind long ones
  - At supermarket, you with milk get stuck behind cart full of small items

FCFS Scheduling (cont.)

- If threads arrive in order: T_2 , T_3 , T_1, then we have



  - Waiting time for T1 is **6**, for T2 is **0**, and for T3 is **3**
  - Average waiting time is **(6 + 0 + 3)/3 = 3**
  - Average response time is **(3 + 6 + 30)/3 = 13**
  - Average waiting time is much better (before it was **17**)
  - Average response time is better (before it was **27**)
- Pros and cons of FCFS
  - Simple (+)
  - Short tasks get stuck behind long ones (-)

Round Robin (RR) Scheduling

- FCFS is potentially bad for short tasks!
  - Depends on submit order
  - If you are first in line at supermarket with milk, you don't care who is behind you, on the other hand…
- Round Robin
  - Each thread gets small unit of CPU time, called *time quantum* (usually 10-100 milliseconds)
  - Once quantum expires, thread is preempted and added to end of ready queue
  - N threads in ready queue and time quantum is $q \Rightarrow$
    - Each thread gets $1/N$ of CPU time in chunks of at most q time units
    - No thread waits more than $(N-1)q$ time units

Photo: Pinterest.ca

Example: RR with Time Quantum of 20

- Example:

Thread	Burst Time
T_1	53
T_2	8
T_3	68
T_4	24

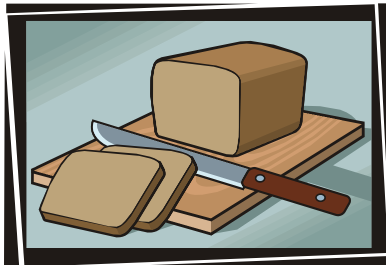
  - The Gantt chart is

T_1	T_2	T_3	T_4	T_1	T_3	T_4	T_1	T_3	T_3

0 20 28 48 68 88 108 112 125 145 153

  - Waiting time for $T_1 = (68 - 20) + (112 - 88) = 72$
    $T_2 = (20 - 0) = 20$
    $T_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$
    $T_4 = (48 - 0) + (108 - 68) = 88$
  - Average waiting time is $(72 + 20 + 85 + 88) / 4 = 66\frac{1}{4}$
  - Average response time is $(125 + 28 + 153 + 112) / 4 = 104\frac{1}{2}$

Round-Robin Discussion



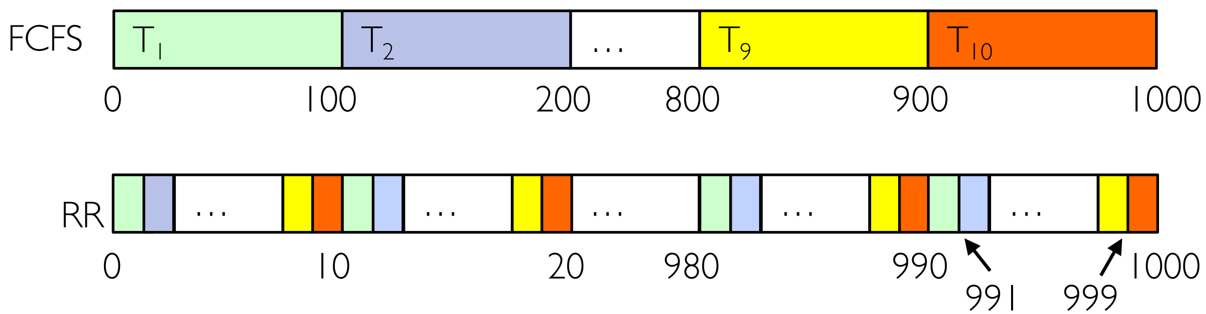
- Pros and cons of RR
  - Better for short tasks, Fair (+)
  - Context-switching time adds up for long tasks (-)
- How does *performance* change with time quantum?
  - What if it's too long?
    - Response time suffers!
  - What if it's too short?
    - Throughput suffers!
  - What if it's infinite (∞)?
    - RR $\Rightarrow$ FCFS
  - Time quantum must be long compared to context switching time, otherwise overhead will be too high

Round-Robin Discussion (cont.)

- Actual choices of time quantum
  - Initially, UNIX time quantum was one second
    - Worked ok when UNIX was used by one or two users
    - What if you use text editor while there are three compilations going on?
      - It takes 3 seconds to echo each keystroke!
  - Need to balance short-task performance and long-task throughput
    - Typical time quantum today is between 10ms – 100ms
    - Typical context-switching overhead is 0.1ms – 1ms
    - Roughly 1% overhead due to context-switching

FCFS vs. RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Suppose there are 10 tasks, each take 100s of CPU time, RR quantum is 1s



- Completion times

Task #	FCFS	RR
1	100	991
2	200	992
...	...	...
9	900	999
10	1000	1000

FCFS vs. RR (cont.)

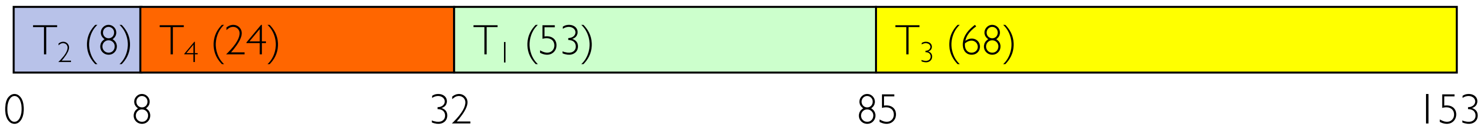
- Completion times

Task #	FCFS	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  - Bad when all jobs have the same length
- Also, cache must be shared between all tasks with RR but can be devoted to each task with FIFO
  - Total time for RR is longer even for zero-cost context switching!

Earlier Example: RR vs. FCFS, Effect of Different Time Quanta

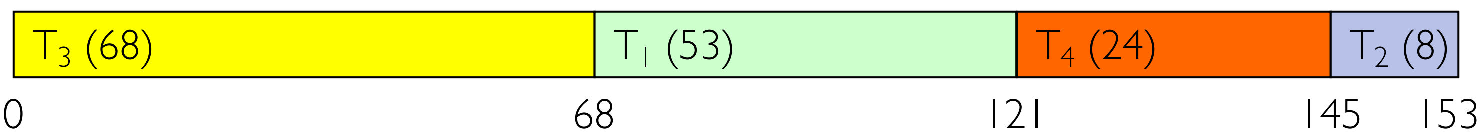
Best FCFS



	Quantum	T1	T2	T3	T5	Average
Waiting Time	Best FCFS	32	0	85	8	31¼
	1					
	5					
	8					
	10					
	20					
	Worst FCFS					
Response Time	Best FCFS	85	8	153	32	69½
	1					
	5					
	8					
	10					
	20					
	Worst FCFS					

Earlier Example: RR vs. FCFS, Effect of Different Time Quanta (cont.)

Worst FCFS



	Quantum	T1	T2	T3	T5	Average
Waiting Time	Best FCFS	32	0	85	8	31¼
	1					
	5					
	8					
	10					
	20					
	Worst FCFS	68	145	0	121	83½
Response Time	Best FCFS	85	8	153	32	69½
	1					
	5					
	8					
	10					
	20					
	Worst FCFS	121	153	68	145	121¾

Earlier Example: RR vs. FCFS, Effect of Different Time Quanta (cont.)

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_1	P_3	P_1	P_3	P_1	P_3	P_3	P_3

0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128 133 141 149 153

	Quantum	T1	T2	T3	T5	Average
Waiting Time	Best FCFS	32	0	85	8	31¼
	1	84	22	85	57	62
	5	82	20	85	58	61¼
	8	80	8	85	56	57¼
	10	82	10	85	68	61¼
	20	72	20	85	88	66¼
	Worst FCFS	68	145	0	121	83½
Response Time	Best FCFS	85	8	153	32	69½
	1	137	30	153	81	100½
	5	135	28	153	82	99½
	8	133	16	153	80	95½
	10	135	18	153	92	99½
	20	125	28	153	112	104½
	Worst FCFS	121	153	68	145	121¾

Shortest Task First (SJF) Scheduling

- Could we always mirror best FCFS?
- Shortest Task First (SJF)
  - Run task that has least amount of computation to do
  - Sometimes called "Shortest Time to Completion First" (STCF)
- Shortest Remaining Time First (SRTF)
  - Preemptive version of SJF: If task arrives and has shorter time to completion than remaining time on current task, immediately preempt current task
  - Sometimes called "Shortest Remaining Time to Completion First" (SRTCF)
- These can be applied to whole program or current CPU burst
  - Key idea: get short tasks out of system
  - Big effect on short tasks, only small effect on long ones
  - Better average response time

SJF/SRTF Optimality

- SJF/SRTF minimize average response time! Why?
  - Consider alternative policy P (not SJF/SRTF) that is optimal
  - At some point, P chooses to run task that is not the shortest
  - Keep order of tasks the same, but run the shorter task first
  - This reduces average response time ⇒ contradiction!

SJF/SRTF Discussion

- SJF/SRTF are best you can do to minimize average response time
  - Provably optimal (SJF among non-preemptive, SRTF among preemptive)
  - Since SRTF is always at least as good as SJF, we can just focus on SRTF

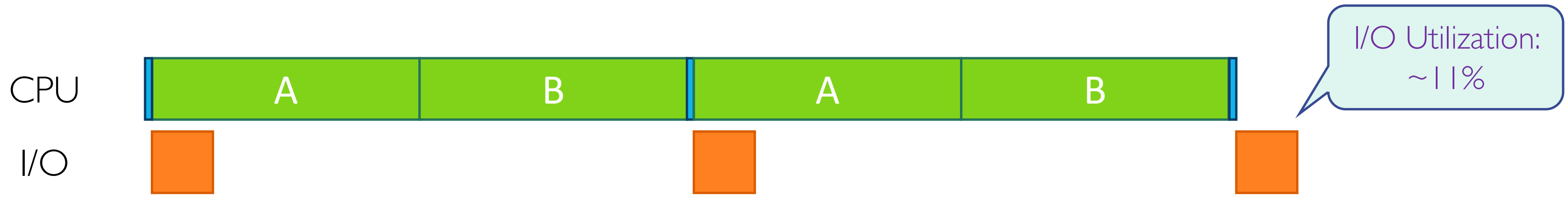
- Comparison of SRTF with FCFS
  - What if all tasks are the same length?
    - SRTF ⇒ FCFS (i.e., FCFS is best we can do if all tasks have the same length)
  - What if tasks have varying length?
    - Unlike FCFS, with SRTF, short tasks do not get stuck behind long ones

Mix of CPU and I/O Bound Tasks: FCFS vs. RR vs. SRTF

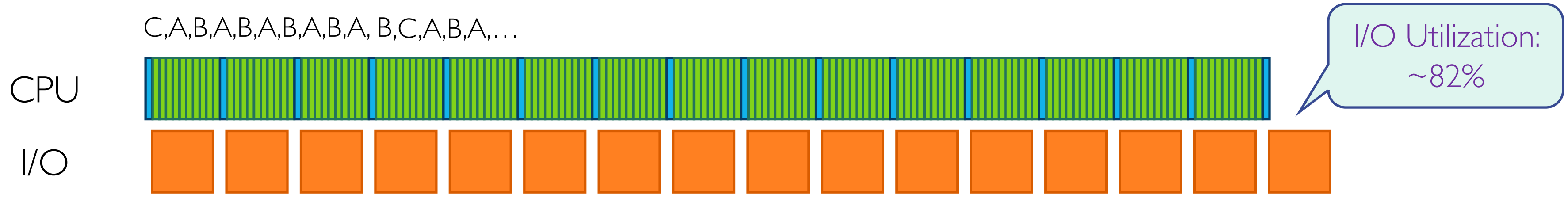
- Example: Suppose there are three tasks
  - A and B are both CPU bound with CPU bursts that last for a week
  - C is I/O bound with iterations of 1ms CPU burst followed by 9ms I/O burst
  - If A or B run by themselves, CPU utilization is 100% and I/O utilization is 0%
  - If C runs by itself, CPU utilization is 10% and I/O utilization is 90%
- With happens under FCFS scheduling policy?
  - Once A or B get in, keep CPU for two weeks ⇒ poor avg. response time
- What about RR or SRTF?
  - Easier to see with a timeline

Mix of CPU and I/O Bound Tasks: FCFS vs. RR vs. SRTF (cont.)

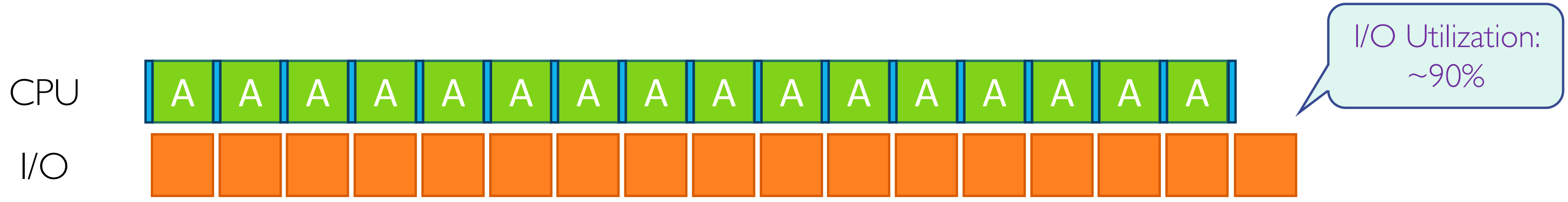
RR with 40ms time quantum



RR with 1ms time quantum



SRTF



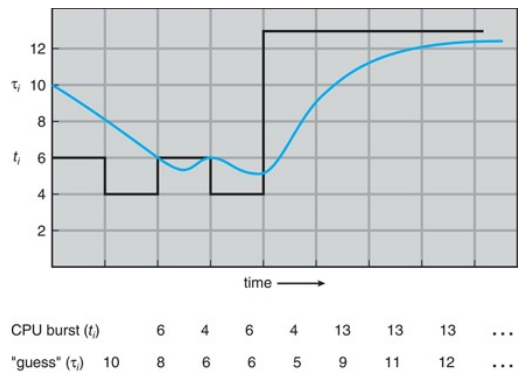
Downsides of SRTF

- Starvation: Large tasks may never run if short ones keep coming
- Overhead: Short tasks preempt long ones ⇒ too many context switches
- Unfair: Large tasks are penalized, there is high variance in response time

- Impractical: We need to somehow predict future (but how?)
  - Some systems ask users
    - When you submit your task, you have to say how long it will take
    - Users could maliciously misreport length of their task
    - E.g., would it work if a supermarket uses SJF?
      - Customers could game the system: come with one item at a time
    - To prevent cheating, systems may kill tasks if they take too long
  - It's hard to predict task's runtime even for non-malicious users

Predicting Length of Next CPU Burst

- Adaptive: Dynamically make predictions based on past behavior
  - Works because programs have predictable behavior
    - If program was I/O bound in past, it'll likely be I/O bound in future
    - If behavior were random, this approach wouldn't help



- Example: Use estimator function on previous bursts
  - Let t_{n-1}, t_{n-2}, t_{n-3}, …, t_1 be previous CPU burst lengths
  - Estimate next burst $\tau_n = f(t_{n-1}, t_{n-2}, t_{n-3}, \ldots)$
  - Function **f** could be any time series estimator (e.g., Kalman filters, etc.)
  - For instance, exponential averaging $\tau_n = \alpha t_{n-1} + (1-\alpha)\tau_{n-1}$ with $(0 < \alpha \leq 1)$

Aside: Application Types

- Can we use past burst times to identify application types?
- Consider mix of *interactive* and *high-throughput* programs
  - How to best schedule them?
  - How to recognize one from the other?
    - Do you trust applications to say that they are "interactive"?
  - Should you schedule the set of applications identically on servers, workstations, pads, and cellphones?

Aside: Application Types (cont.)

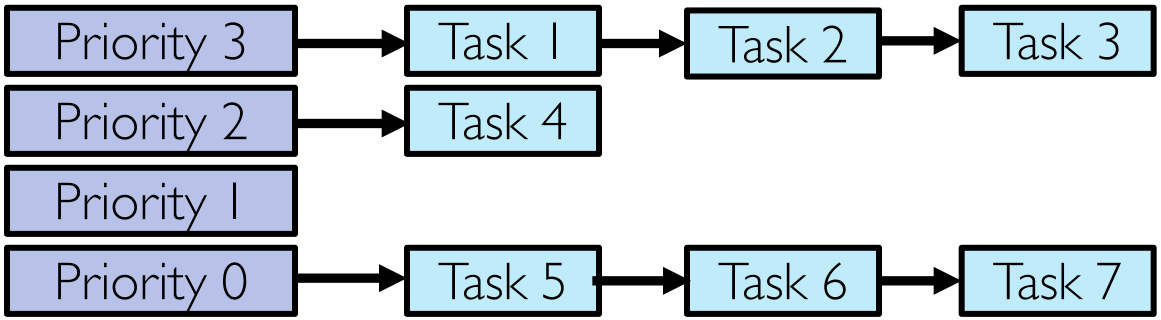
- Assumptions encoded into many schedulers
  - Applications that sleep a lot and have short bursts must be interactive
    - Give them high priority
  - Applications that compute a lot must be high-throughput apps
    - Give them lower priority, since they won't notice intermittent bursts from interactive applications
- In general, it is hard to characterize applications
  - What about applications that sleep for a long time, and then compute for a long time?
  - What about applications that must run under all circumstances

SRTF Final Notes

- Bottom line, we can’t really know how long tasks will take
  - However, we can use SRTF as yardstick for measuring other policies
  - Optimal, so we can’t do any better

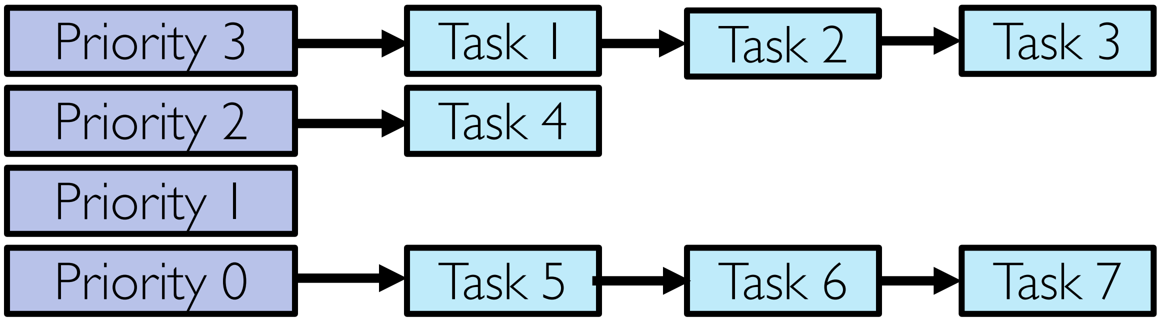
- Pros & cons of SRTF
  - Optimal (average response time) (+)
  - Hard to predict future (-)
  - Too many context switches (-)
  - Unfair (-)

Strict Priority Scheduling



- Execution plan
  - Always execute highest-priority runnable tasks to completion
  - Each queue can be threaded in RR with some time-quantum
- Notice any problems?
  - Starvation: Lower priority tasks don't get to run because higher priority tasks
  - 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

Strict Priority Scheduling (cont.)

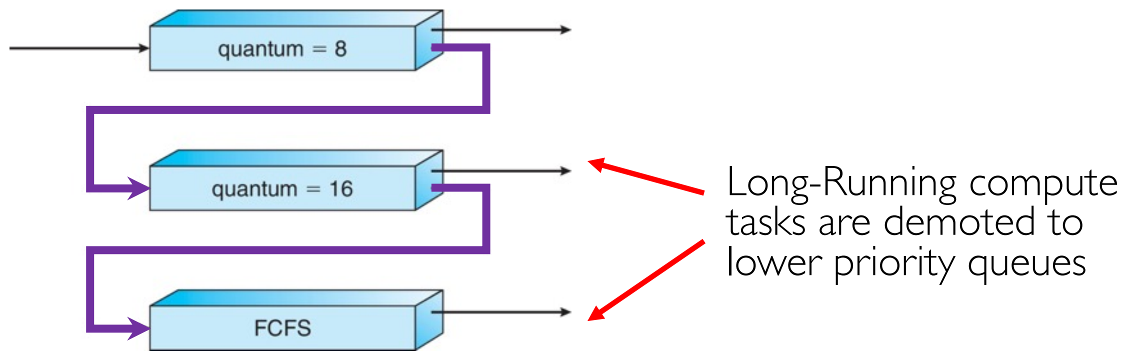


- How to fix problems?
  - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

Scheduling Fairness

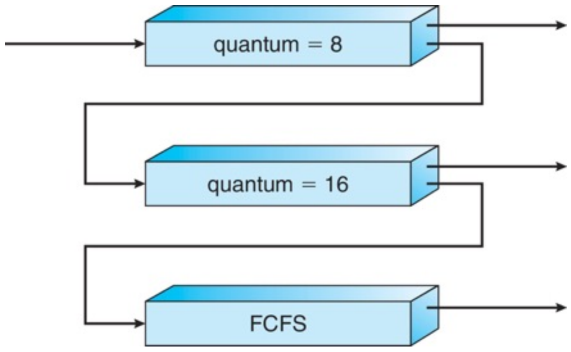
- Strict fixed-priority scheduling between queues is unfair (run highest, then next, etc.)
  - long running tasks may never get any CPU time
  - In Multics, shut down machine, found 10-year-old task
- One approach: Give each queue some fraction of CPU
  - What if there are 100 short tasks and only one long task?
    - Like express lanes in a supermarket, sometimes express lanes get so long, get better service by going into one of other lines
- Another approach: Increase priority of tasks that don't get service
  - What is done in some variants of UNIX
  - This is ad hoc; what rate should you increase priorities?
  - And, as system gets overloaded, no task gets CPU time, so everyone increases in priority $\Rightarrow$ Interactive tasks suffer
- Tradeoff: Fairness is usually gained by hurting average response time!

Multi-Level Feedback Queue Scheduling



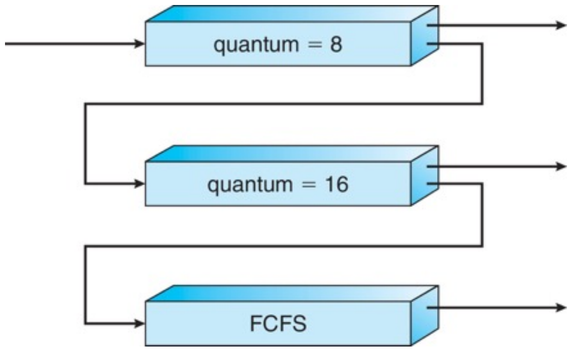
- Another method for exploiting past behavior (first use in CTSS)
  - Multiple queues, each with different priority
    - Higher priority queues often considered "foreground" tasks
  - Each queue has its own scheduling algorithm
    - E.g. foreground – RR, background – FCFS
    - Sometimes multiple RR priorities with quantum increasing exponentially (highest: 1ms, next: 2ms, next: 4ms, etc.)
- Adjust each task's priority as follows (details vary)
  - Task starts in highest priority queue
  - If timeout expires, drop one level
  - If timeout doesn't expire, push up one level (or to top)

Multi-Level Feedback Queue Scheduling (cont.)



- Result approximates SRTF
  - CPU bound tasks drop like a rock
  - Short-running I/O bound tasks stay near top
- Scheduling must be done between queues
  - Fixed priority scheduling
    - Serve all from highest priority, then next priority, etc.
  - Time slicing
    - Each queue gets fraction of CPU time
    - E.g., 70% to highest, 20% next, 10% lowest

Multi-Level Feedback Queue Scheduling (cont.)



- Countermeasure: user action that foil intent of OS designers
  - For multilevel feedback, put simple I/O's to keep task's priority high
  - Example of MIT Othello Contest
    - Cheater put printf's, ran much faster than competitors!
    - Of course, if everyone did this, wouldn't work!

Lottery Scheduling

- Give each task some number of lottery tickets
- On each time slice, randomly pick a winning ticket
- On average, CPU time is proportional to # of tickets given to task
- How to assign tickets?
  - Give tasks tickets proportional to their priorities
  - To approximate SRTF, give short tasks more and long tasks fewer
  - To avoid starvation, give every task at least one ticket (everyone makes progress)
- Compared to strict priority scheduling, lottery scheduling behaves gracefully as load changes
  - Adding or deleting one task affects all tasks proportionally, independent of how many tickets each task possesses

Lottery Scheduling Example

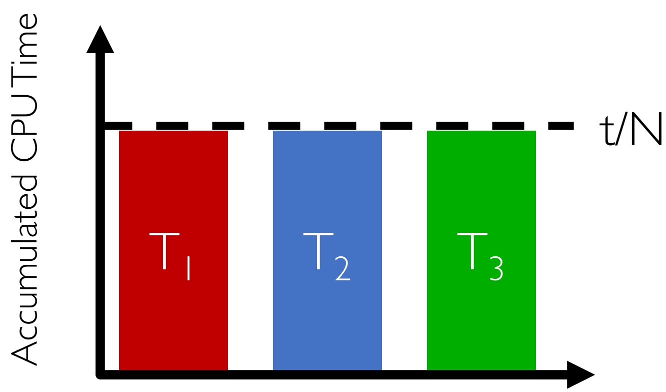
- Assume short tasks get 10 tickets, long tasks get 1 ticket

# short tasks/ # long tasks	% of CPU each short tasks gets	% of CPU each long tasks 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%

- What if too many short tasks to give reasonable response time?
  - If load average is 100, hard to make progress
  - One approach is to log some users out

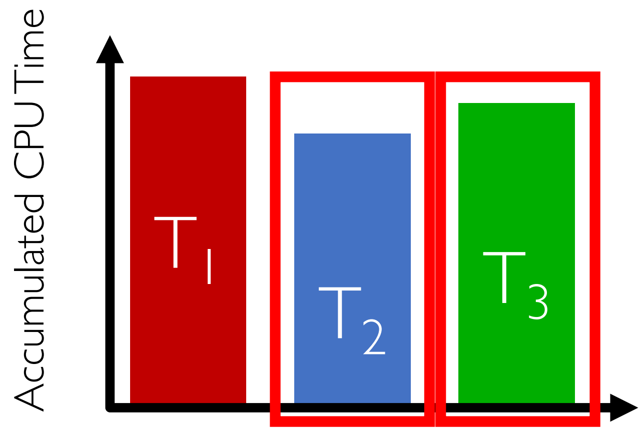
Max-Min Fair (MMF) Scheduling

- Always choose task with lowest accumulated CPU time so far
    - If chosen task doesn't have CPU burst, schedule second lowest …
    - Break ties randomly if multiple tasks equally have lowest CPU time
- Goal is to give each task equal share of CPU time
    - With *N runnable* threads, each thread should get 1/*N*th of CPU time
- At any time *t* we want to have



MMF Scheduling (cont.)

- Strict MMF causes too many context switches
  - It effectively turns to running one instruction of each task
- Relaxed MMF runs task with lowest accumulated CPU time for fixed time quantum before choosing next task



- Notice any problem?
  - Fixed quantum leads to poor response time as # of tasks increases

MMF Scheduling (cont.)

- Solution: Dynamically change time quantum
- Target latency: Time interval during which all tasks should run at least once
- Time quantum = Target latency / N
  - E.g., with 20ms target latency and 4 threads, time quantum is 5ms
- Notice any problem?
  - With 20ms target latency and 200 threads, time quantum becomes 0.1ms
  - Recall RR: Large context switching overhead if time quantum gets to small
- Minimum granularity: Minimum length of any time quantum
  - E.g., with target latency 20ms, 1ms minimum granularity, and 200 processes, time quantum is 1ms

Weighted Max-Min Fair Scheduling

- What if we want to give more to some and less to others (proportional share)?
- Key Idea: Assign weight w_i to each thread i
- MMF uses single time quantum for all tasks

$$Q = \frac{\text{Target latency}}{N}$$

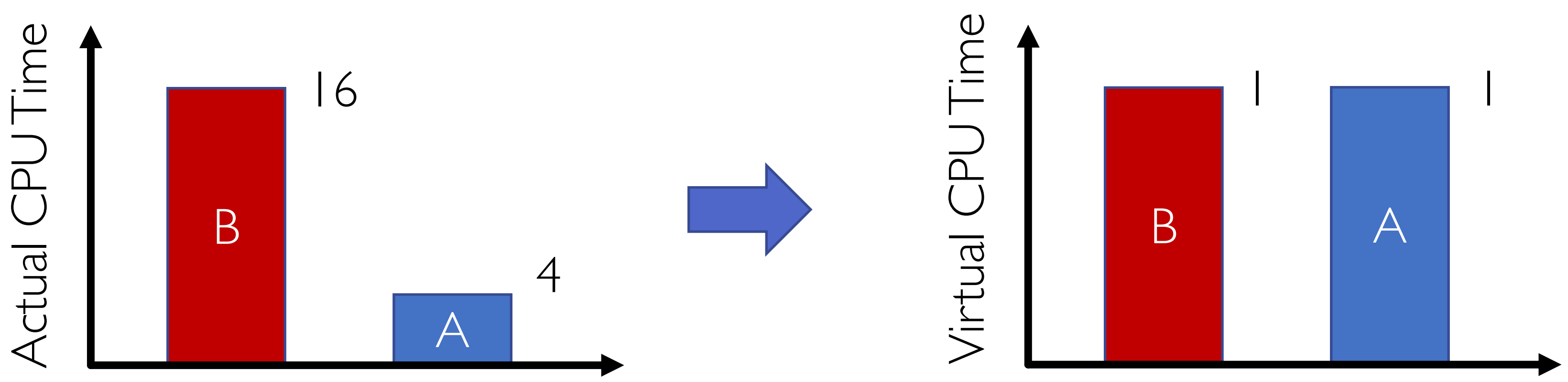
- Weighted MMF uses different time quanta for different tasks

$$Q_i = \frac{w_i \times \text{Target latency}}{\sum_{j=1}^{N} w_j}$$

- E.g., with 20ms target latency, 1ms minimum granularity, and 2 threads: A with weight 1 and B with weight 4
  - Time quantum for A is 4 ms
  - Time quantum for B is 16 ms

Weighted MMF Scheduling (cont.)

- Also track threads' *virtual runtime* rather than their true wall-clock runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly
- Linux *Completely Fair Scheduler* deploys very similar ideas

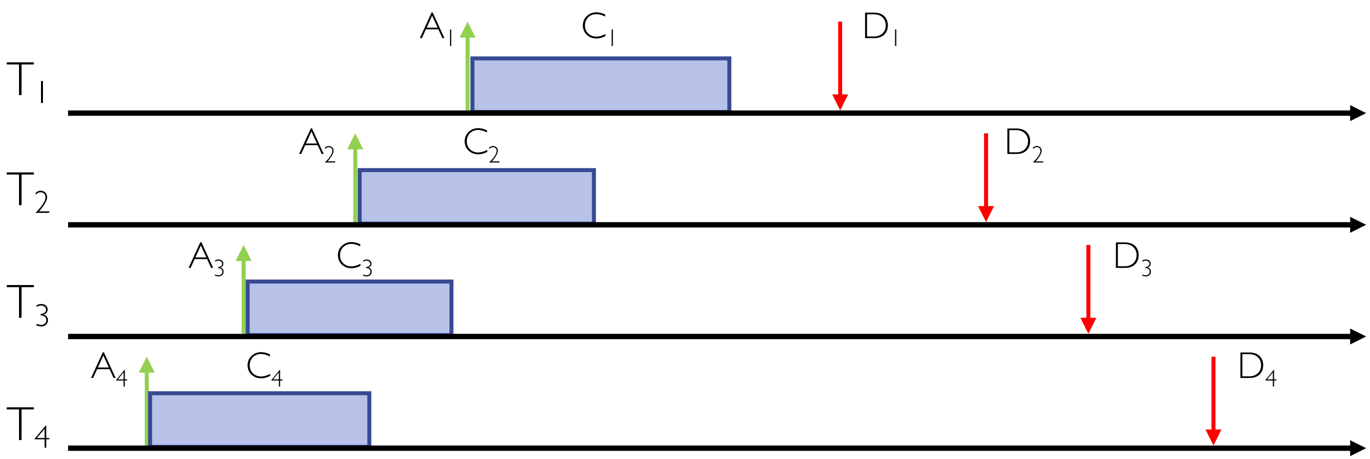


Real-Time Scheduling (RTS)

- Efficiency is important but predictability is essential
  - We need to predict with confidence worst case response times for systems
  - In RTS, performance guarantees are task and/or class centric and often ensured a priori
  - In conventional systems, performance is system/throughput oriented with post-threading (… wait and see …)
  - Real-time is about enforcing predictability, and does not equal fast computing!!!
- 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)*

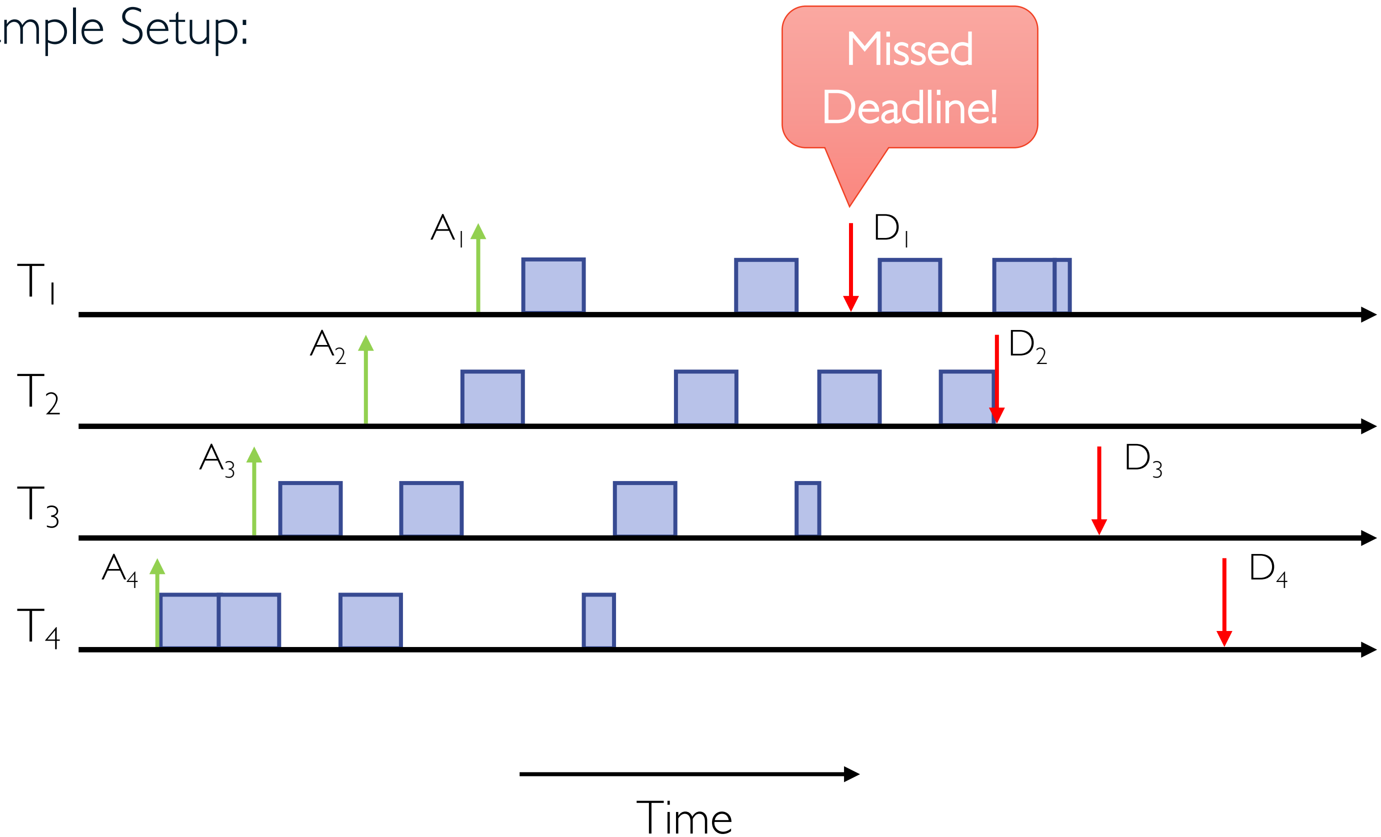
Real-Time Workload Characteristics

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



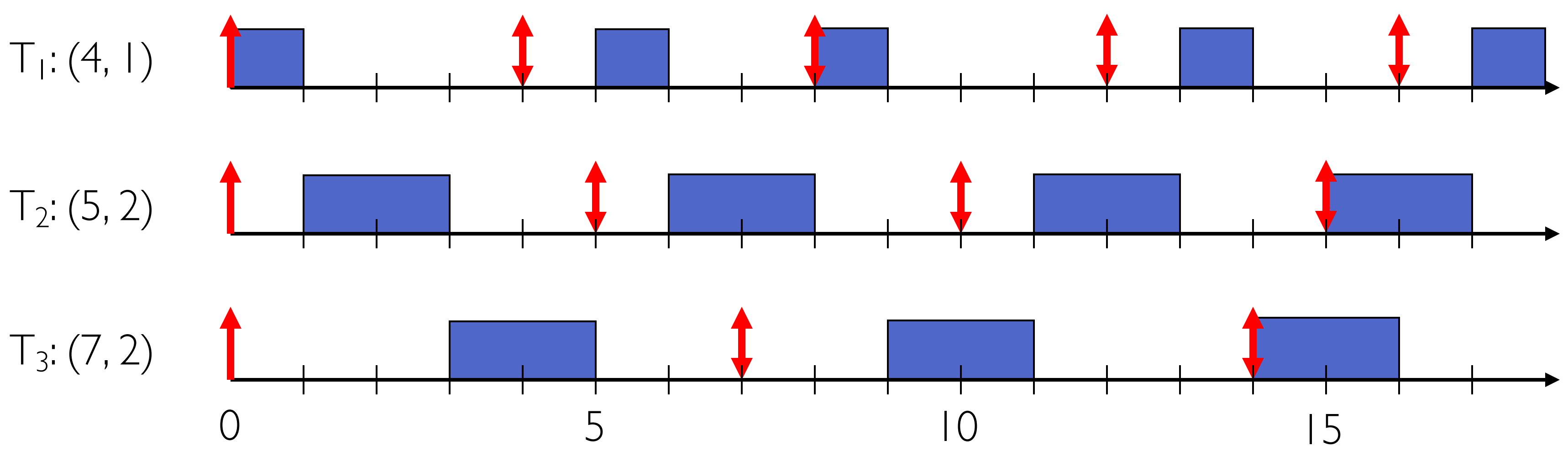
Real-Time Workload Characteristics

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



Earliest Deadline First (EDF)

- Tasks are periodic with period P and computation C in each period: (P, C)
- Preemptive priority-based dynamic scheduling
- Tasks' (current) priority is based on how close their deadline is
- Scheduler always schedules active task with closest deadline



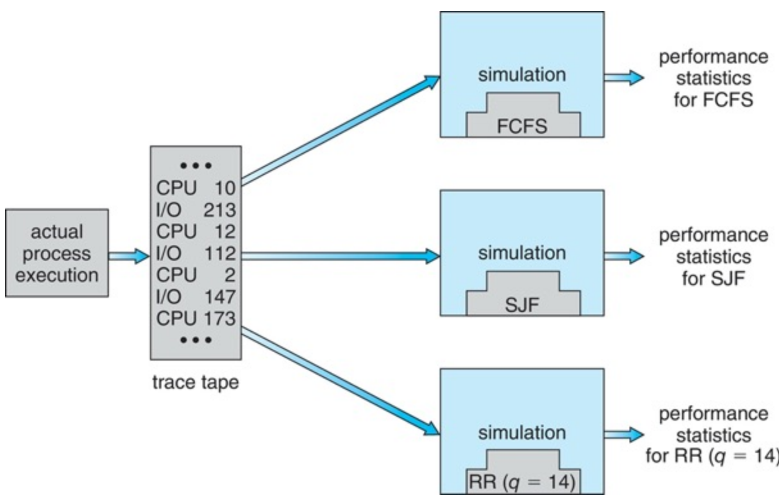
EDF: Feasibility Testing

- Even EDF won't work if you have too many tasks
- For n periodic tasks with computation time C_i and deadline and period D_i, feasible schedule exists if

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

How to Evaluate Scheduling Algorithms?

- Deterministic modeling
  - Take predetermined workload and compute performance of each algorithm
- 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



Starvation and Sample Bias

- Suppose you want to compare scheduling policies
  - Create some infinite sequence of arriving tasks
  - Start measuring
  - Stop at some point
  - Compute ART for finished tasks between start and stop
- Is this valid or invalid?
  - SJF and FCFS would complete different sets of tasks
    - Their ARTs are not directly comparable
    - E.g., suppose you stopped at any point in FCFS vs. SJF slide

Solutions for Sample Bias

- For both systems, measure for long enough that # of completed tasks >> # of uncompleted tasks

- Start and stop system in idle periods
  - Idle period: no work to do
  - If algorithms are work-conserving, both will complete the same set of tasks

Choosing Right Scheduling Algorithm

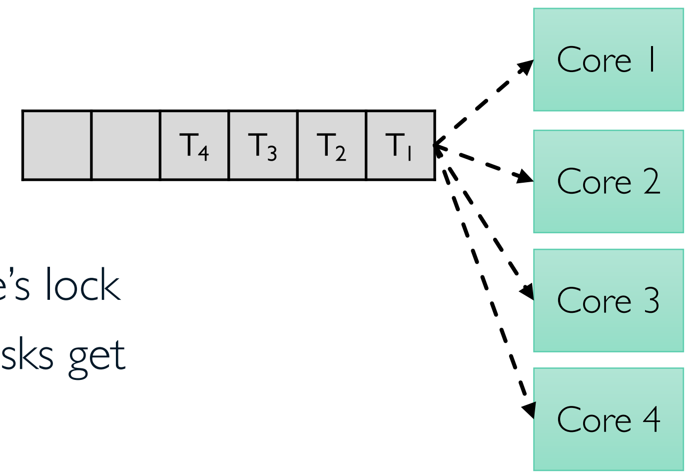
I Care About:	Then Choose:
CPU Throughput	FCFS
Avg. Response Time	SRTF Approximation
I/O Throughput	SRTF Approximation
Fairness (CPU Time)	Linux CFS
Fairness – Wait Time to Get CPU	Round Robin
Meeting Deadlines	EDF
Favoring Important Tasks	Priority

Outline

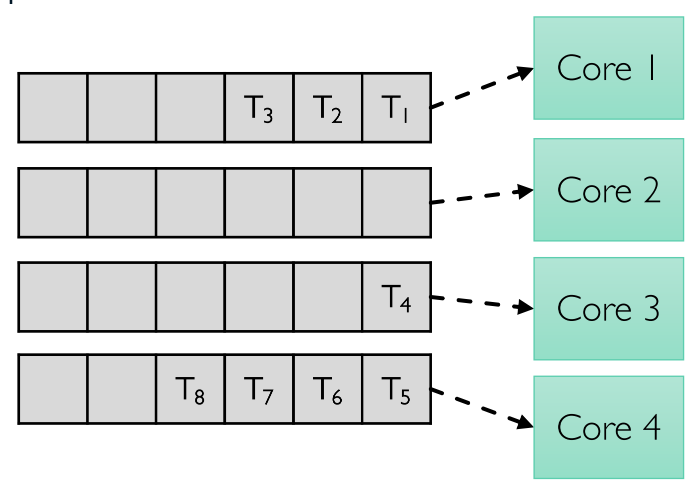
- Definitions
  - Response time, throughput, scheduling policy, …
- Uniprocessor policies
  - FCFS, SJF/SRTF, Round Robin, …
  - Real-time scheduling
- Multiprocessor policies
  - Oblivious scheduling, gang scheduling, …

Multicore Processor Scheduling

- There could be one ready queue for all cores



- Notice any problems?
  - Single bottleneck: Contention for ready queue's lock
  - Limited cache reuse: Lack of data locality as tasks get scheduled on different cores
- Solution: each core has its own private ready queue



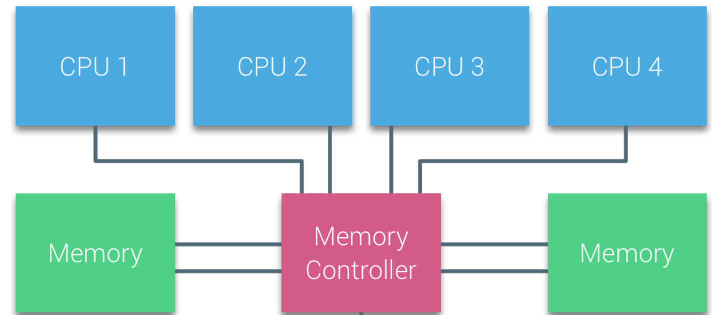
- Notice any problems?
  - Load balancing: Some cores might be idle while tasks pile up on others ready queues
- One solution: Work stealing
  - Idle cores steal waiting task from busy ones

Processor Affinity

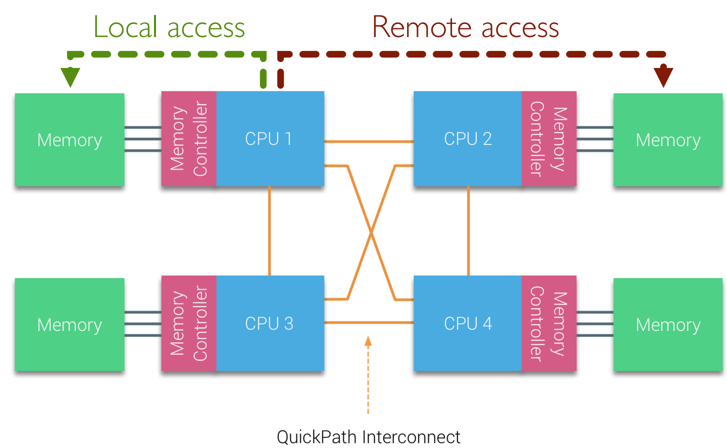
- When task run on core, cache contents of that core stores recent memory accesses by that task
- This is referred to as core affinity of tasks
- Load balancing may affect core affinity as task migrate between cores
- Performance of migrated task suffers because it loses contents of what it had in cache of the core it was moved off of
  - Migration is justified only if performance loss is less than waiting time
- Soft affinity: OS tries to keep tasks on same core, but no guarantees
- Hard affinity: OS allows tasks to specify set of cores they may run on

NUMA and CPU Scheduling

- Uniform memory access (UMA): Cores experience same, uniform access time to any memory module



- Non-uniform memory access (NUMA): Cores access their local memory modules faster than remote memory modules

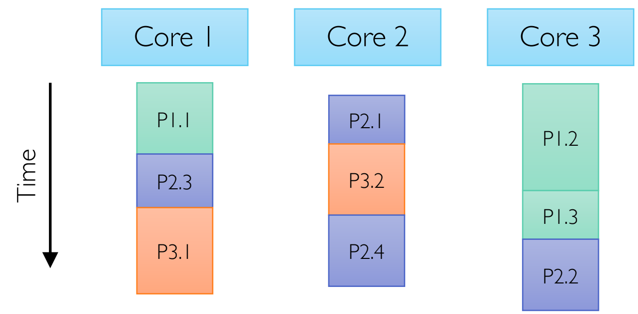


- If OS is NUMA-aware, it will assign memory closes to core that task is running on

Photos from: http://www.evoventurepartners.com

Scheduling Multithreaded Programs

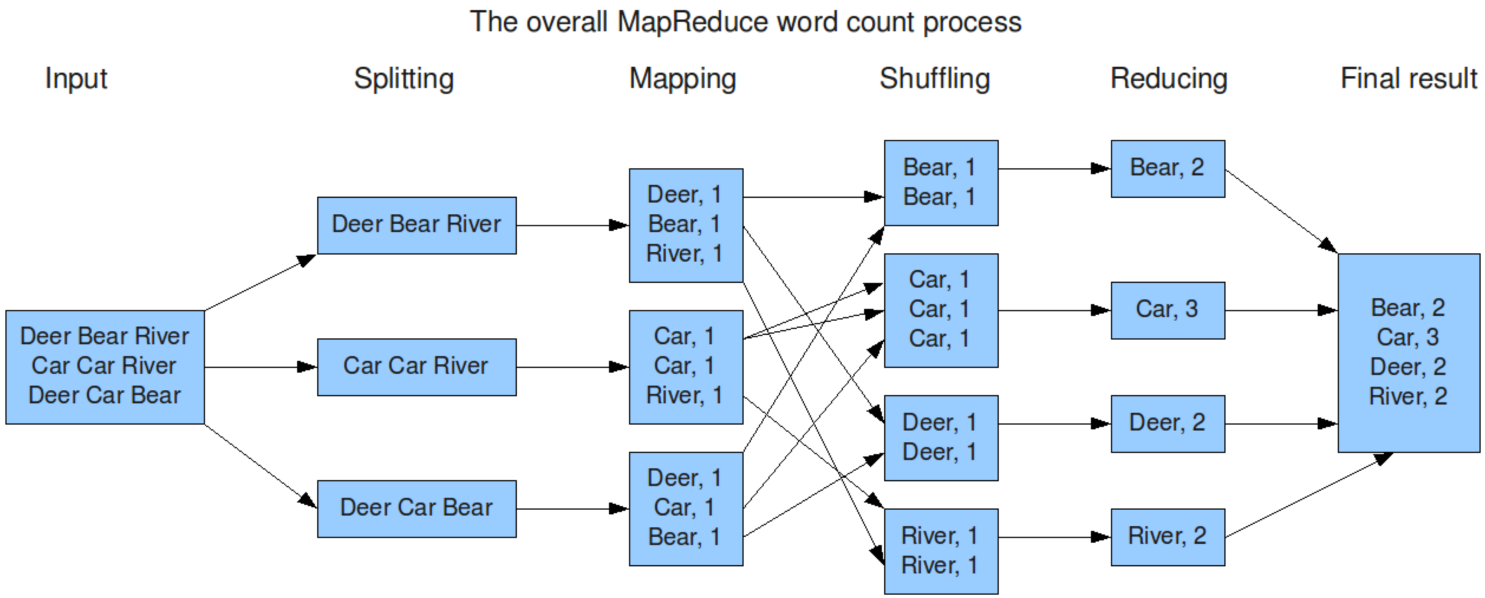
- So far, we assumed that there is one thread per program
- Now, consider scheduling multithreaded programs on multicore processor
- At any given time, multiple threads from same program could be running



Px.y: thread y in process x

- Oblivious scheduling: Cores independently schedule threads in their queue
  - Each thread is treated as independent task
- What happens if one thread gets time-sliced while others are still running?
  - Assuming program uses locks and condition variables, it will still be correct
  - Performance, however, could suffer if threads actually depend on one another

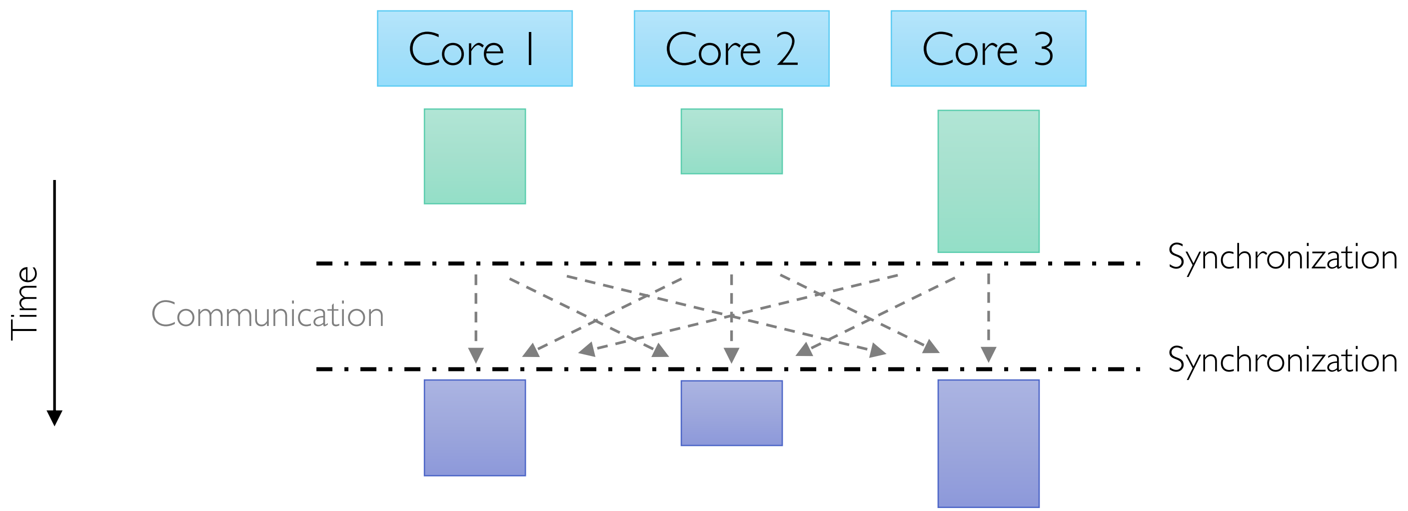
Problem with Oblivious Scheduling: Bulk Synchronous Delay



- Data parallelism is common programming design pattern (e.g., Google MapReduce)
  - Data is split into roughly equal sized chunks
  - Chunks are processed independently on different cores
  - Once all chunks are processed, cores synchronize and communicate their results to next stage of computation

Photo from cs.calvin.edu

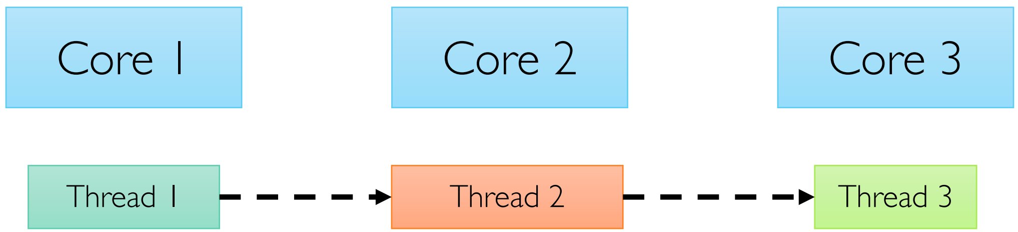
Problem with Oblivious Scheduling: Bulk Synchronous Delay (cont.)



- At each step, computation is limited by the slowest task
- If task is preempted on one core, its work is delayed, stalling all other cores

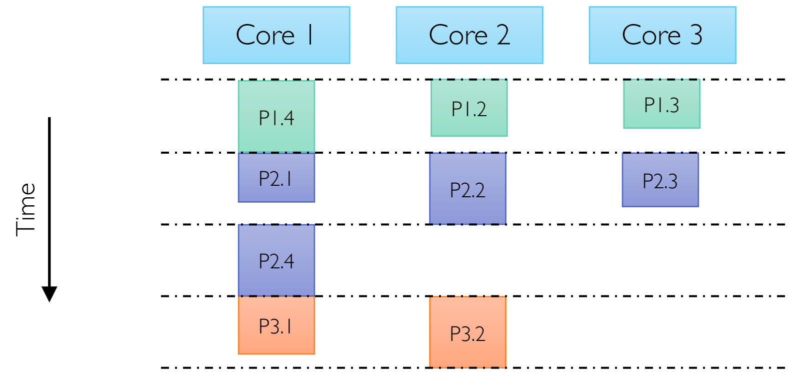
Photo from cs.calvin.edu

Problem with Oblivious Scheduling: Producer-Consumer Delay



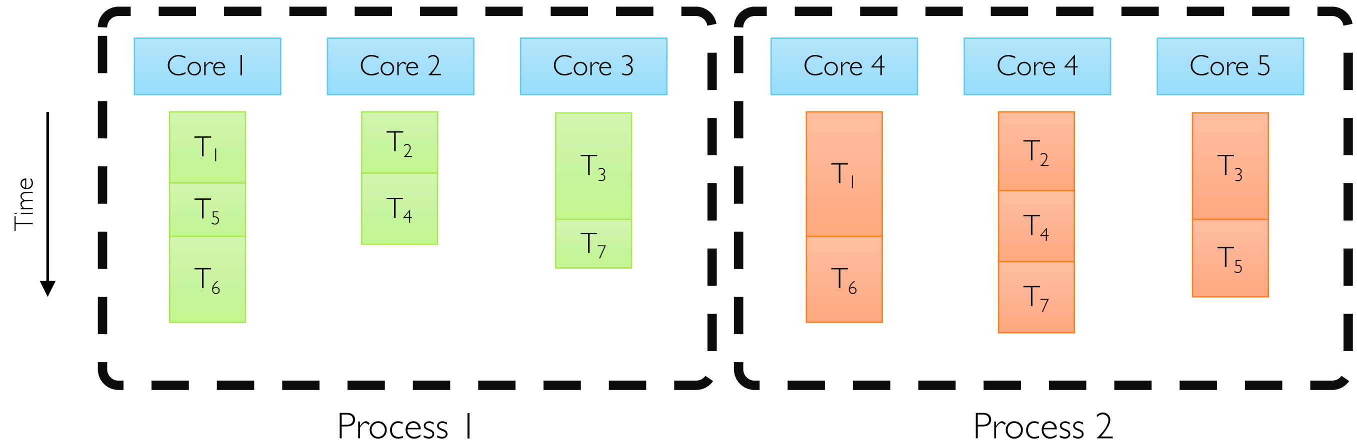
- Producer-consumer design patter is also very common
- Preempting a thread on one core stalls all others in the chain
- Some other problems with oblivious scheduling
  - Preempting a thread on the critical path will slow down the entire process
  - Preempting lock holder stalls others until lock holder is re-scheduled

Gang Scheduling



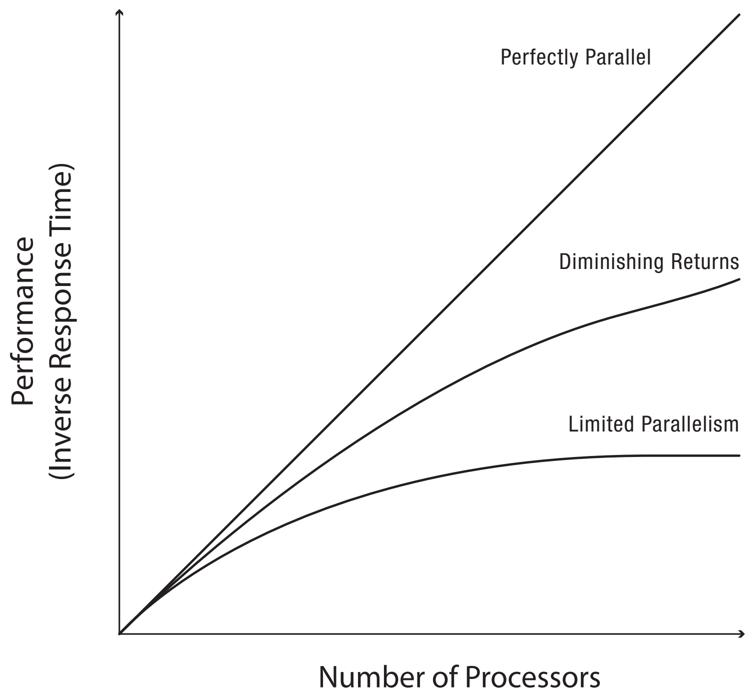
- Time is divided into equal intervals
- Threads from same process are scheduled at beginning of each interval
- Notice any problems?
  - CPU cycles are waisted when threads have different lengths
  - Some cores remain idle when a process doesn't have enough tasks for all cores

Space Sharing



- Each process is assigned a subset of cores
  - Minimizes processor context switches

How Many Cores Does a Process Need?

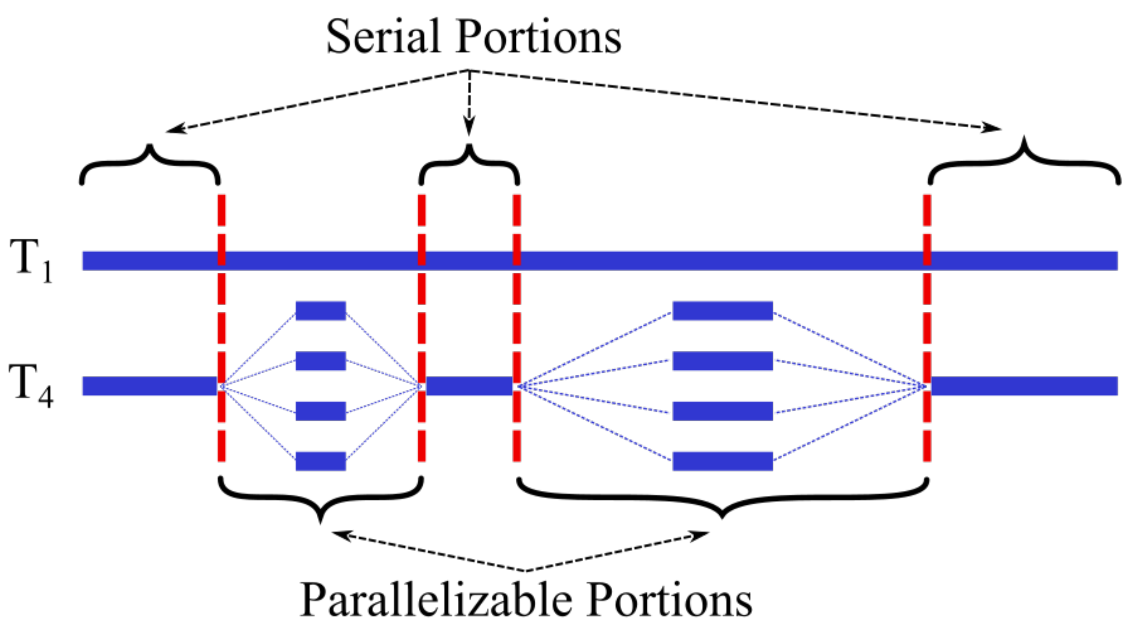


- There are overheads
    - E.g., creating extra threads, synchronization, communication
- Overheads shift the curve down

Amdahl's Law

[G. Amdahl 1967]

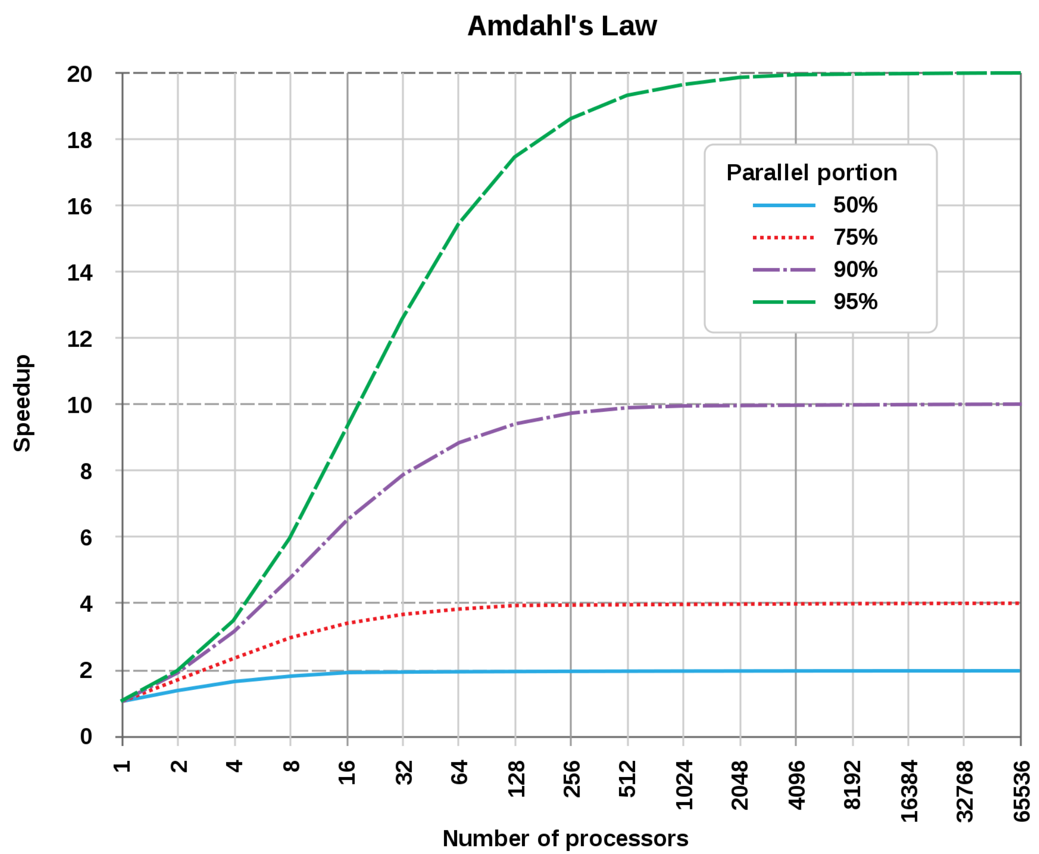
- Architects use it to estimate upper bounds on speedups



$$\text{Speedup(x)} = \frac{T_1}{T_x} = \frac{T_1}{(1-F)T_1 + \frac{FT_1}{x}} = \frac{x}{x(1-F)+F}$$

Amdahl's Law (cont.)

[G. Amdahl 1967]



$$\text{Speedup}(\text{x}) = \frac{T_1}{T_x} = \frac{T_1}{(1-F)T_1 + \frac{FT_1}{x}} = \frac{x}{x(1-F)+F}$$

What Portion of Code is Parallelizable?

[Allen Karp and Horace Flatt 1990]

- Expert programmers may not know!
- Fortunately, we can measure speedup

Measure

Set

Calculate

$$s(x) = \frac{x}{x(1-F)+F}$$

Karp-Flatt Metric

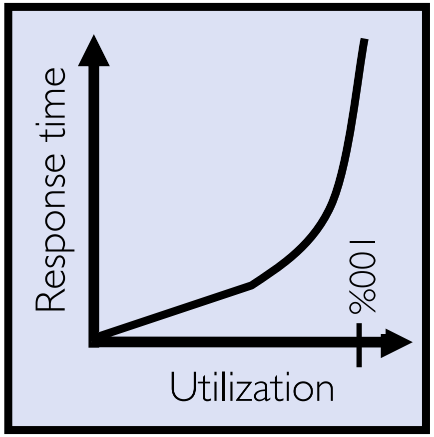
$$F = \left(1 - \frac{1}{s(x)}\right)\left(1 - \frac{1}{x}\right)$$

?

Fig. criticallyrated.files.wordpress.com

A Final Word On Scheduling

- When do details of scheduling policy and fairness really matter?
  - When there aren't enough resources to go around
- When should you simply buy faster cores?
  (Or network link, or expanded highway, or …)
  - Buy it when it will pay for itself in improved response time, assuming you're paying for worse response time in reduced productivity, customer angst, etc…
  - Might think you need X fully utilized core, but usually you will have to buy more than X because response time goes to infinity as utilization approaches100%



- Interesting implication of this curve
  - Most scheduling algorithms work fine in linear portion of curve, fail otherwise
  - Argues for buying faster resources when hit knee of curve

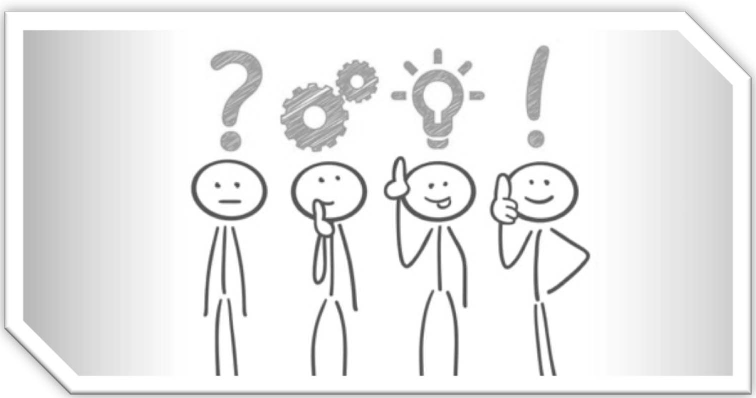
Summary (1 of 2)

- First-Come, First-Served (FCFS)
  - Threads are served in the order of their arrival
- Round-Robin (RR)
  - Give each thread a small amount of CPU time when it executes; cycle between all ready threads
- Shortest Task First (SJF) / Shortest Remaining Time First (SRTF):
  - Run whatever task that has the least amount of computation to do/least remaining amount of computation to do
- Multi-level Feedback Queue (MFQ)
  - Multiple queues of different priorities and scheduling algorithms
- Lottery Scheduling
  - Give each thread a priority-dependent number of tickets

Summary (2 of 2)

- Max-Min Fair (MMF)
  - Give each task equal share of CPU time
- Real-Time Scheduling
  - Need to meet a deadline, predictability essential
- Oblivious Scheduling
  - Each core schedules its own threads
- Gang Scheduling
  - Schedule tasks from same process at the same time
- Space Sharing
  - Give each process some number of cores

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny