東南大學
SOUTHEAST UNIVERSITY

OPERATING SYSTEM CONCEPTS

Chapter 5. CPU Scheduling

A/Prof. Kai Dong

Warm-up

Process Execution — Time-sharing

OS @run (kernel mode)	**Hardware**	**Program (user mode)**
		Process *A*
		…
	timer interrupt	
	save regs(*A*) to k-stack(*A*)	
	move to *kernel mode*	
	jump to *timer handler*	
handle *trap*		
call switch() routine		
save regs(*A*) to PCB(*A*)		
restore regs(*B*) from PCB(*B*)		
switch to k-stack(*B*)		
return-from-trap (into *B*)		
	restore regs(*B*) from k-stack(*B*)	
	move to *user mode*	
	jump to *B*’s PC	
		Process *B*
		…

Warm-up

Scheduler & Dispatcher

- Who makes the decision of which runs next?
  - **CPU scheduler**
- Who performs the context switch?
  - **Dispatcher**

Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

CPU Scheduler

- **Short-term scheduler** selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- **Timing**: CPU scheduling decisions may take place when a process
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Basic Concepts

Dispatcher

- **Dispatcher** module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency — time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

Methodology on Evaluation

- How to conduct evaluation in a scientific paper?
- Benchmark
  - Comparing algorithms
  - Datasets, or scenario assumptions
  - Certain metrics
- For CPU scheduling
  - Scheduling algorithms
  - Workload assumptions
  - Scheduling criteria

Scheduling Criteria

Common Scheduling Criteria

- **CPU utilization** — keep the CPU as busy as possible
- **Throughput** — # of processes that complete their execution per time unit
- **Turnaround time** — amount of time to execute a particular process

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- **Waiting time** — amount of time a process has been waiting in the ready queue
- **Response time** — amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Simple Scheduling Algorithms
Workload Assumptions

- Assumptions:
  1. Each job runs for the same amount of time. $A, B, C, \cdots$
  2. All jobs arrive at the same time. $T_{arrival}$
  3. Once started, each job runs to completion.
  4. All jobs only use the CPU (they perform no I/O)
  5. The run-time of each job is known.

Simple Scheduling Algorithms

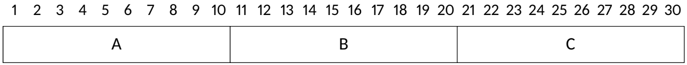
FCFS

- **First Come, First Served** (**FCFS**) scheduling or sometimes First In, First Out (FIFO).
- Three jobs arrive in the system, *A*, *B*, and *C*, at roughly the same time ($T_{arrival} = 0$).
  - Assume that while they all arrived simultaneously, *A* arrived just a hair before *B* which arrived just a hair before *C*.
  - Assume also that each job runs for 10 seconds.
- What will the average turnaround time be for these jobs?

Simple Scheduling Algorithms

FCFS (contd.)

- First draw a **Gantt Chart**



- Then compute
  - *A* finished at 10, *B* at 20, and *C* at 30.
  - The average turnaround time is

$$\frac{10 + 20 + 30}{3} = 20$$

Simple Scheduling Algorithms

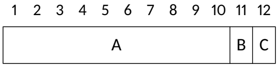
FCFS (contd.)

- Assumptions:
    1. Each job runs for the same amount of time. *A, B, C,* · · · ~~Each job runs for the same amount of time. *A, B, C,* · · ·~~
    2. All jobs arrive at the same time. $T_{arrival}$
    3. Once started, each job runs to completion.
    4. All jobs only use the CPU (they perform no I/O)
    5. The run-time of each job is known.
- Question:
    - Is FIFO optimal given assumptions 1-5?
    - Let's relax assumption 1. How does FIFO perform now?
    - What kind of workload could you construct to make FIFO perform poorly?

Simple Scheduling Algorithms

FCFS (contd.)

- Assume three jobs (A, B, and C), but this time A runs for 10 seconds while B and C run for 1 second each.



  - A finished at 10, B at 11, and C at 12.
  - The average turnaround time is

$$\frac{10 + 11 + 12}{3} = 11$$

Simple Scheduling Algorithms

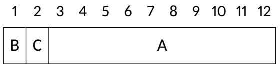
FCFS (contd.)

- **Convoy effect** — number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.
- Question
  - If you are the scheduler, who to run first?
  - B or C runs first, and A runs at last.

Simple Scheduling Algorithms

SJF

- **Shortest Job First** (**SJF**) — shortest job first, then the next shortest, and so on.



  - *B* finished at 1, *C* at 2, and *A* at 12, .
  - The average turnaround time is

$$\frac{12+1+2}{3} = 5$$

Simple Scheduling Algorithms

Priority

- The SJF algorithm is a special case of the general **priority**-scheduling algorithm.
- Example

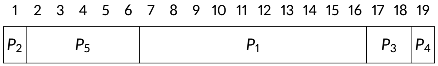
Procsss	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Draw the Gantt chart for priority scheduling, considering all processes P_1-P_5 arrived at time 0.

Simple Scheduling Algorithms

Priority (contd.)

- We assume that low numbers represent high priority



- Problem: **Starvation** — low priority processes may never execute
- Solution: **Aging** — as time progresses increase the priority

Simple Scheduling Algorithms

HRRN

- **Highest Response Ratio Next** (**HRRN**) — the next job is not that with the shorted estimated run time, but that with the highest response ratio defined as

$$response\_ratio = 1 + \frac{waiting\_time}{estimated\_run\_time}$$

- Another modification of SJF or Priority.

Simple Scheduling Algorithms

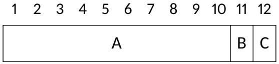
SJF (contd.)

- Assumptions:
  1. ~~Each job runs for the same amount of time. *A*, *B*, *C*, ···~~
  2. All jobs arrive at the same time. $T_{arrival}$
     ~~All jobs arrive at the same time. $T_{arrival}$~~
  3. Once started, each job runs to completion.
  4. All jobs only use the CPU (they perform no I/O)
  5. The run-time of each job is known.
- Question:
  - Is SJF optimal given assumption 2-5?
  - Let's relax assumption 2. How does SJF perform now?
  - What kind of workload could you construct to make SJF perform poorly?

Simple Scheduling Algorithms

SJF (contd.)

- Assume three jobs (A, B, and C), but this time A arrives at $T = 0$ and needs to run for 10 seconds, whereas B and C arrive at $T = 1$ and each need to run for 1 second.



  - A finished at 10, B at 11, and C at 12.
  - The average turnaround time is

$$\frac{10 + (11 - 1) + (12 - 1)}{3} = 10.33$$

Simple Scheduling Algorithms

SJF (contd.)

- Assumptions:
  1. ~~Each job runs for the same amount of time. $A, B, C, \cdots$~~
  2. ~~All jobs arrive at the same time. $T_{arrival}$~~
  3. Once started, each job runs to completion. ~~Once started, each job runs to completion.~~
  4. All jobs only use the CPU (they perform no I/O)
  5. The run-time of each job is known.
- To address this concern, we need to relax assumption 3.

Simple Scheduling Algorithms

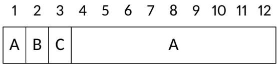
Preemptive Vs. Non-preemptive

- **Preemptive**, and **non-preemptive** schedulers — Whether a job can preempt another job
- All modern schedulers are preemptive.
- The dispatcher performs a context switch.
- Think of preemptive version of SJF and Priority.

Simple Scheduling Algorithms

Preemptive SJF

- **Preemptive Shortest Job First** (**preemptive SJF**, or Shortest Time-to-Completion First (STCF), or Shortest-Remaining-Time First (SRTF)



  - *A* finished at 12, *B* at 2, and *C* at 3.
  - The average turnaround time is

$$\frac{12 + (2 - 1) + (3 - 1)}{3} = 5$$

Simple Scheduling Algorithms

Response Time

- Turnaround time is a performance metric, another metric of interest is fairness.
- Users may sit at a terminal and demand interactive performance from the system.
- The **response time** of a job is the time from when the job arrives in a system to the first time it is scheduled.

$$T_{response} = T_{firstrun} - T_{arrival}$$

- Preemptive SJF is not good for response time.
- Can you build a scheduler that is sensitive to response time?

Simple Scheduling Algorithms

RR

- **Round-Robin** (**RR**) — instead of running jobs to completion, RR runs a job for a time slice (sometimes called a scheduling quantum) and then switches to the next job in the run queue.
- RR is sometimes called time-slicing.
- Assume three jobs *A*, *B*, and *C* arrive at the same time in the system, and that they each wish to run for 5 seconds, the length of time slice is 1 second

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	A	B	C	A	B	C	A	B	C	A	B	C

  - The average response time is

$$\frac{0+1+2}{3}=1$$

Simple Scheduling Algorithms

RR (contd.)

- For the length of time slice, the shorter the better?
  - Take into account the cost in context switch.
- RR is awful in turnaround time. Is it?
  - Any policy that is fair, performs poorly on performance metrics such as turnaround time.
  - Trade-off between fairness and performance.

Simple Scheduling Algorithms

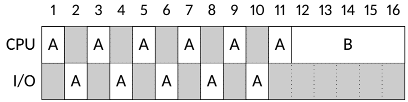
Incorporating I/O

- Assumptions:
  1. ~~Each job runs for the same amount of time. $A, B, C, \cdots$~~
  2. ~~All jobs arrive at the same time. $T_{arrival}$~~
  3. ~~Once started, each job runs to completion.~~
  4. All jobs only use the CPU (they perform no I/O) ~~All jobs only use the CPU (they perform no I/O)~~
  5. The run-time of each job is known.
- Let's relax assumption 4.

Simple Scheduling Algorithms

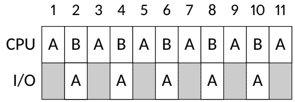
Incorporating I/O (contd.)

- Assume two jobs, *A* and *B*. *A* runs for 1 second and then issues an I/O request which also takes 1 second, *B* simply uses the CPU for 5 seconds and performs no I/O.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
CPU	A		A		A		A		A		A	B				
I/O		A		A		A		A		A						

- Break a job to sub-jobs due to I/O requests.



	1	2	3	4	5	6	7	8	9	10	11
CPU	A	B	A	B	A	B	A	B	A	B	A
I/O		A		A		A		A		A	

Simple Scheduling Algorithms

Relax All Assumptions

- Assumptions:
  1. ~~Each job runs for the same amount of time. $A, B, C, \cdots$~~
  2. ~~All jobs arrive at the same time. $T_{arrival}$~~
  3. ~~Once started, each job runs to completion.~~
  4. ~~All jobs only use the CPU (they perform no I/O).~~
  5. The run-time of each job is known. ~~The run-time of each job is known.~~
- Let's relax assumption 5.
  - How to schedule without perfect knowledge?
  - How to trade-off between fairness and performance?

Simple Scheduling Algorithms

Determining Length of Next CPU Burst in SJF

- Can only estimate the length — should be similar to the previous one
  - Then pick process with shortest predicted next CPU burst
- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. $t_n =$ actual length of n^{th} CPU burst
  2. $\tau_{n+1} =$ predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Estimate: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$
- Commonly, α set to 1/2

Simple Scheduling Algorithms

A Hybrid — Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground — RR
  - background — FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice — each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Simple Scheduling Algorithms

In Class Exercise

Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

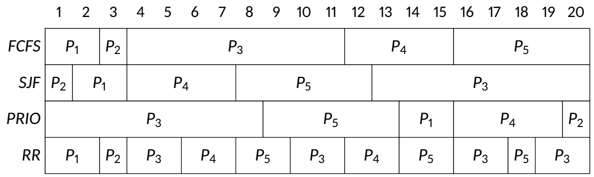
The processes are assumed to have arrived in the order of P_1, P_2, P_3, P_4, P_5, all at time 0.

1. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a larger priority number implies a higher priority), and RR (quantum $= 2$).
2. What is the average turnaround time for each of the scheduling algorithms in part 1?
3. What is the waiting time of each process for each of these scheduling algorithms?
4. Which of the algorithms results in the minimum average waiting time (over all processes)?

In Class Exercise

Key to

Q1:


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
FCFS P_1 P_2 P_3 P_4 P_5
SJF P_2 P_1 P_4 P_5 P_3
PRIO P_3 P_5 P_1 P_4 P_2
RR P_1 P_2 P_3 P_4 P_5 P_3 P_4 P_5 P_3 P_5 P_3

Q2:

	FCFS	SJF	PRIO	RR
$T_{turnaround}$ (ms)	$\frac{2+3+11+15+20}{5}$ =10.2	$\frac{1+3+7+12+20}{5}$ =8.6	$\frac{8+13+15+19+20}{5}$ =15	$\frac{2+3+20+13+18}{5}$ =11.2

Q3 & Q4:

		P_1	P_2	P_3	P_4	P_5	Avg	
$T_{waiting}$ (ms)	FCFS	0	2	3	11	15	6.2	
	SJF	1	0	12	3	7	4.8	Minimum
	PRIO	13	19	0	15	8	11	
	RR	0	2	12	9	13	7.2	

Multilevel Feedback Queue

- Relax all assumptions: How to schedule without perfect knowledge?
- **Multi-level Feedback Queue** (**MLFQ**)
  - optimize turnaround time
  - minimize response time
- The multilevel queue (MLQ) has a number of distinct queues, each assigned a different priority level.
  - Jobs are permanently assigned to one queue.
  - MLQ uses priorities to decide which job should run at a given time.
    - A job with higher priority is chosen to run.
    - Use round-robin (or other) scheduling among the jobs which have the same priority.

Multilevel Feedback Queue

MLQ: Basic Rules

- Two basic rules for MLQ:
  - Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
  - Rule 2: If Priority(A) = Priority(B), A & B run in RR.
- Can priority change overtime?
- MLFQ: what is feedback?
- MLFQ varies the priority of a job based on its behavior.
  - MLFQ will try to learn about processes as they run, and thus use the history of the job to predict its future behavior.

Multilevel Feedback Queue

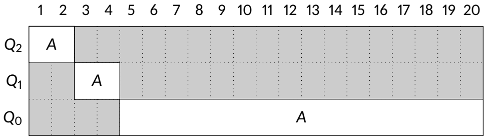
How To Change Priority

- For a new coming job ...
- The scheduler doesn't know whether a job will be a short job or a long-running job.
- The scheduler first assumes it might be a short job (given high priority).
- If it actually is a short job, it will run quickly and complete;
- If it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long job.

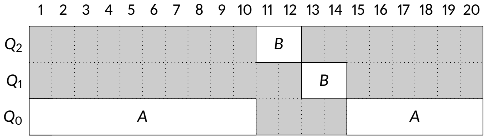
Multilevel Feedback Queue

How To Change Priority (contd.)

- A long-running job *A*:



- Along came a short job *B*:



Multilevel Feedback Queue

MLFQ: MLQ + Feedback

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in RR.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
- Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.
- Is our current version of MLFQ perfect?

Multilevel Feedback Queue

Limitations

- Starvation
  - If there are “too many” interactive jobs in the system, they will combine to consume all CPU time, and thus long-running jobs will never receive any CPU time (they starve).
- Gaming the scheduler
  - Doing something sneaky to trick the scheduler into giving you more than your fair share of the resource (e.g., by running for 99% of a time slice before relinquishing the CPU).
- Changeable program behaviors
  - A program may change its behavior over time; what was CPU bound may transition to a phase of interactivity. With our current approach, such a job would be out of luck and not be treated like the other interactive jobs in the system.

Multilevel Feedback Queue

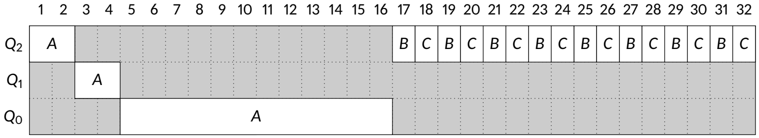
The Priority Boost

- How to solve these problems?
- Periodically boost the priority
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.
- Some problems are solved
  - First, processes are guaranteed not to starve.
  - Second, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

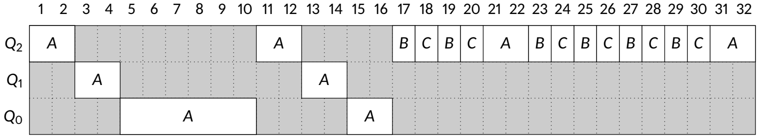
Multilevel Feedback Queue

The Priority Boost (contd.)

- Without Rule 5



- With Rule 5 (priority boost every 10 ms)



Multilevel Feedback Queue

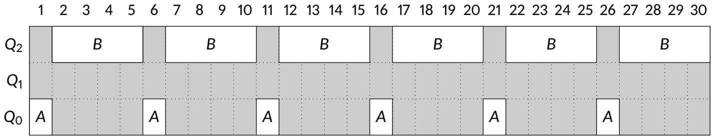
Better Accounting

- How to prevent gaming of the scheduler?
  - ~~Rule 4a: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).~~
  - ~~Rule 4b: If a job gives up the CPU before the time slice is up, it stays at the same priority level.~~
- Better accounting of CPU time
  - Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

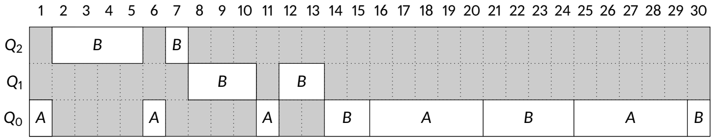
Multilevel Feedback Queue

Better Accounting (contd.)

- With old Rule 4



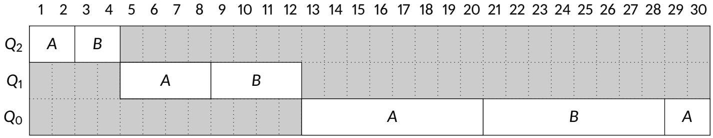
- With new Rule 4 (time slice = 5 ms)



Multilevel Feedback Queue

Tuning MLFQ

- How big should the time slice be per queue?
- Varying time-slice length across different queues.
  - Lower priority, longer quanta.



Multilevel Feedback Queue

MLFQ Rules: The Final Version

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in RR.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

Multilevel Feedback Queue

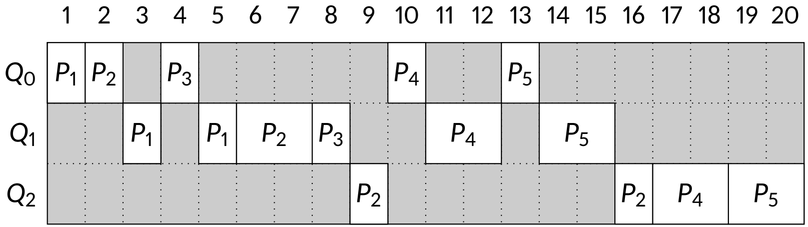
In Class Exercise

Given the table below showing the process information of a system, based on multilevel feedback queuing scheduling scheme. Suppose there are five levels in the system and within each level the FCFS scheduling is used. Draw a Gantt chart to show the time of CPU allocated to each process until all processes are finished using time quantum $q = 2^i$, (where q = time allocated for a process to run in its turn, and i ranges from 0 to 4 indicating the i^{th} level queue).

Process	Arrival Time	Service Time
P_1	0	3
P_2	1	5
P_3	3	2
P_4	9	5
P_5	12	5

Multilevel Feedback Queue

Key


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Q_0: P_1 P_2 P_3 P_4 P_5
Q_1: P_1 P_1 P_2 P_3 P_4 P_5
Q_2: P_2 P_2 P_4 P_5

Lottery Scheduling

Proportional Share

- Suppose a virtualized data center, where
  - You might like to assign one-quarter of your CPU cycles to the Windows VM and the rest to your base Linux installation
- Proportional-share scheduler (fair-share scheduler)
  - Instead of optimizing for turnaround or response time, a scheduler might instead try to guarantee that each job obtain a certain percentage of CPU time.

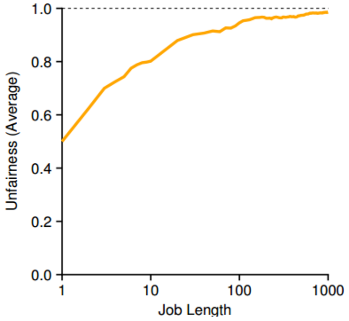
Lottery Scheduling

- Every so often, hold a lottery to determine which process should get to run next;
- Processes that should run more often should be given more chances to win the lottery.
- Tickets are used to represent the share of a resource that a process (or user or whatever) should receive.
  - Ticket currency — Currency allows a user with a set of tickets to allocate tickets among their own jobs in whatever currency they would like; the system then automatically converts said currency into the correct global value.
  - Ticket transfer — A process can temporarily hand off its tickets to another process.
  - Ticket inflation — A process can temporarily raise or lower the number of tickets it owns.

Lottery Scheduling

An Unfairness Metric

- Suppose two jobs competing against one another, each with the same number of tickets and same run time.
- An unfairness metric U:
  - The time the first job completes divided by the time that the second job completes.
  - With a perfect fair scheduler, two jobs should finish at roughly the same time, i.e., U=1.



Lottery Scheduling

Probabilistic Vs. Deterministic

- Lottery scheduling is probabilistic.
  - Only as the jobs run for a significant number of time slices does the lottery scheduler approach the desired fairness.
- Stride scheduling, is a deterministic fair-share scheduler.
  - Each job in the system has a stride, which is inverse in proportion to the number of tickets it has.
  - Every time a process runs, we will increment a counter for it (called its pass value) by its stride to track its global progress.
  - At any given time, pick the process to run that has the lowest pass value so far.

Lottery Scheduling

Stride Scheduling

- Suppose three processes (A, B and C), with stride values of 100, 200 and 40, and all with pass values initially at 0.

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	**Who Runs?**
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

- **Why using probabilistic, not deterministic?**

Lottery Scheduling

Why Not Deterministic?

- Lottery scheduling has one nice property that stride scheduling does not: no global state.
  - Imagine a new job enters in the middle of our stride scheduling example above; what should its pass value be? Should it be set to 0? If so, it will monopolize the CPU.

Thread Scheduling

- Distinction between user-level and kernel-level threads
- On operating systems that support threads, it is kernel-level threads — not processes — that are being scheduled
  - Kernel thread scheduled onto available CPU is system-contention scope (SCS) — competition among all threads in system
- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Typically done via priority set by programmer
  - Known as process-contention scope (PCS) since scheduling competition is within the process

Multiple-Processor Scheduling

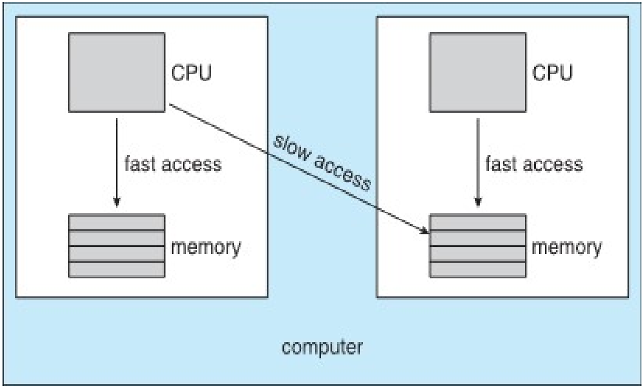
Cache

- Caches are based on the notion of locality
  - Temporal locality: when a piece of data is accessed, it is likely to be accessed again in the near future
  - Spatial locality: if a program accesses a data item at address *x*, it is likely to access data items near *x* as well
- Caches in multi-processor Architecture
  - Cache Affinity (a.k.a., Processor Affinity) — A process, when run on a particular CPU, builds up a fair bit of state in the caches of the CPU. The next time the process runs, it is often advantageous to run it on the same CPU, as it will run faster if some of its state is already present in the caches on that CPU

Multiple-Processor Scheduling

Cache Affinity

- Memory-placement algorithms can also consider affinity.



Multiple-Processor Scheduling

Asymmetric Multiprocessing

- **Asymmetric Multiprocessing** — single-queue multiprocessor scheduling (SQMS)
- Simply reuse the basic framework for single processor scheduling, by putting all jobs that need to be scheduled into a single queue
- Cache affinity problem

$$Queue \longrightarrow A \longrightarrow B \longrightarrow C \longrightarrow D \longrightarrow E \longrightarrow NULL$$

CPU_0	A	E	D	C	B
CPU_1	B	A	E	D	C
CPU_2	C	B	A	E	D
CPU_3	D	C	B	A	E

CPU_0	A	E	A		
CPU_1	B		E	B	
CPU_2	C			E	C
CPU_3	D				E

Multiple-Processor Scheduling

Symmetric Multiprocessing

- **Symmetric Multiprocessing** — multi-queue multiprocessor scheduling (MQMS)
- One queue per CPU. Each queue will likely follow a particular scheduling discipline, such as round robin.

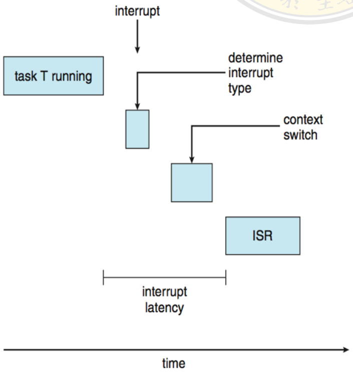
$Queue_0 \rightarrow A \rightarrow C \rightarrow NULL$ $Queue_1 \rightarrow B \rightarrow D \rightarrow NULL$

CPU_0	A	C	A	C	A	C	B
CPU_1	B	D	B	D	B	D	

- Load imbalance problem.
  - Migration — By migrating a job from one CPU to another, true load balance can be achieved.

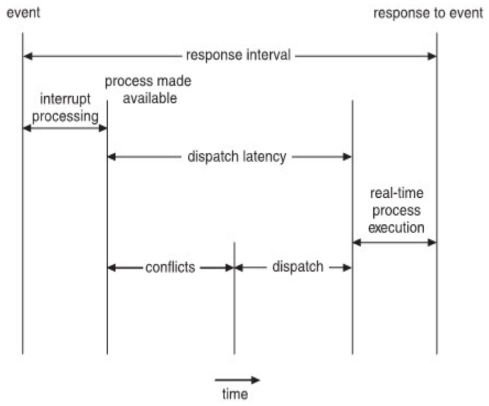
Real-Time CPU Scheduling

- **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
- Two types of latencies affect performance
  - Interrupt latency — time from arrival of interrupt to start of routine that services interrupt
  - Dispatch latency — time for scheduling dispatcher to take current process off CPU and switch to another



Real-Time CPU Scheduling

- Conflict phase of dispatch latency:
  - Preemption of any process running in kernel mode
  - Release by low-priority process of resources needed by high-priority processes



event
response to event
response interval
interrupt processing
process made available
dispatch latency
real-time process execution
conflicts
dispatch
time