Notes Week 3: Scheduling

TABLE OF CONTENTS

Agenda

1. Review; Where are we now?
2. Lab 1 Review
3. Content: Scheduling

Weekly Summary and Where are we?

Topics

- **Last Week:** Introduction to the Process abstraction and how to use it in C
- **This Week:** How the OS schedules processes (and how this relates to scheduling other things)
- **Next Week:** Concurrency and Synchronization

Assignments

- **Lab 0** (setting up your dev environment) should be submitted already!
- **Lab 1** (getting familiar with C, `gdb`, `vim`, `ctags`, etc) is being graded; we'll review in class today
- **Lab 2** (building `ls`) is out; due 30 Jan 2026 (NEXT Friday).
- **HW 1** is being graded; grades will be released by this Friday
- **HW 2** is due on Friday
- **HW 3** is out

Reading Summary

Reading for this week was (from OSTEP):

- **Chap 7**: Scheduling: Introduction)
- **Chap 8**: Scheduling: The Multi-Level Feedback Queue
- **Chap 9**: Scheduling: Proportional Share

Lab 1 Review

Why are we doing reviews?

- To learn from each other.
- To practice talking about your work.
- To help convince me you didn't use AI to do your work for you.
  - Or at least, if you used AI to help, you understand what it helped you do!

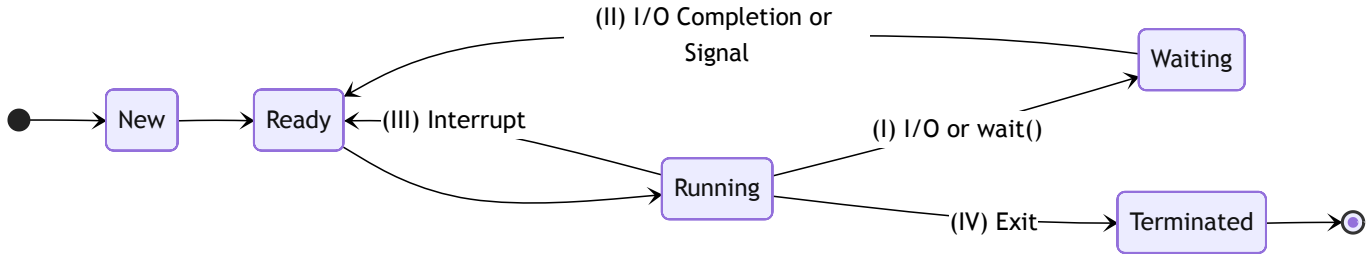
Introduction to Scheduling

High-level problem: operating system has to decide which process (or thread) to run.

Process States and Scheduling

This state diagram shows the lifecycle of a process.

Process State Diagram



The kernel can make scheduling decisions about a process during the following transitions:

- (I) Switches from running to waiting state
- (II) Switches from waiting to ready
- (III) Switches from running to ready state
- (IV) Exits

DEFINITION: PREEMPTIVE SCHEDULING
A process can be interrupted/suspended or re-scheduled (restarted) to run at any point

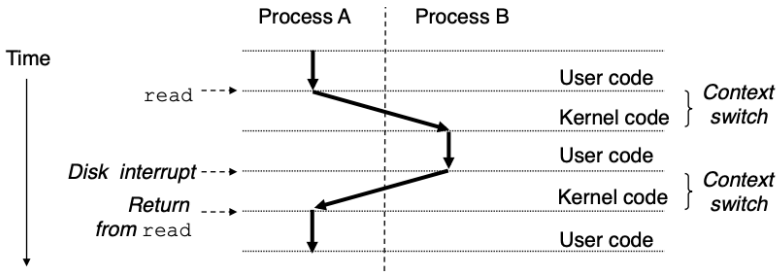
A process that is ***preempted*** means that it has been temporarily suspended; the process can be "interrupted".

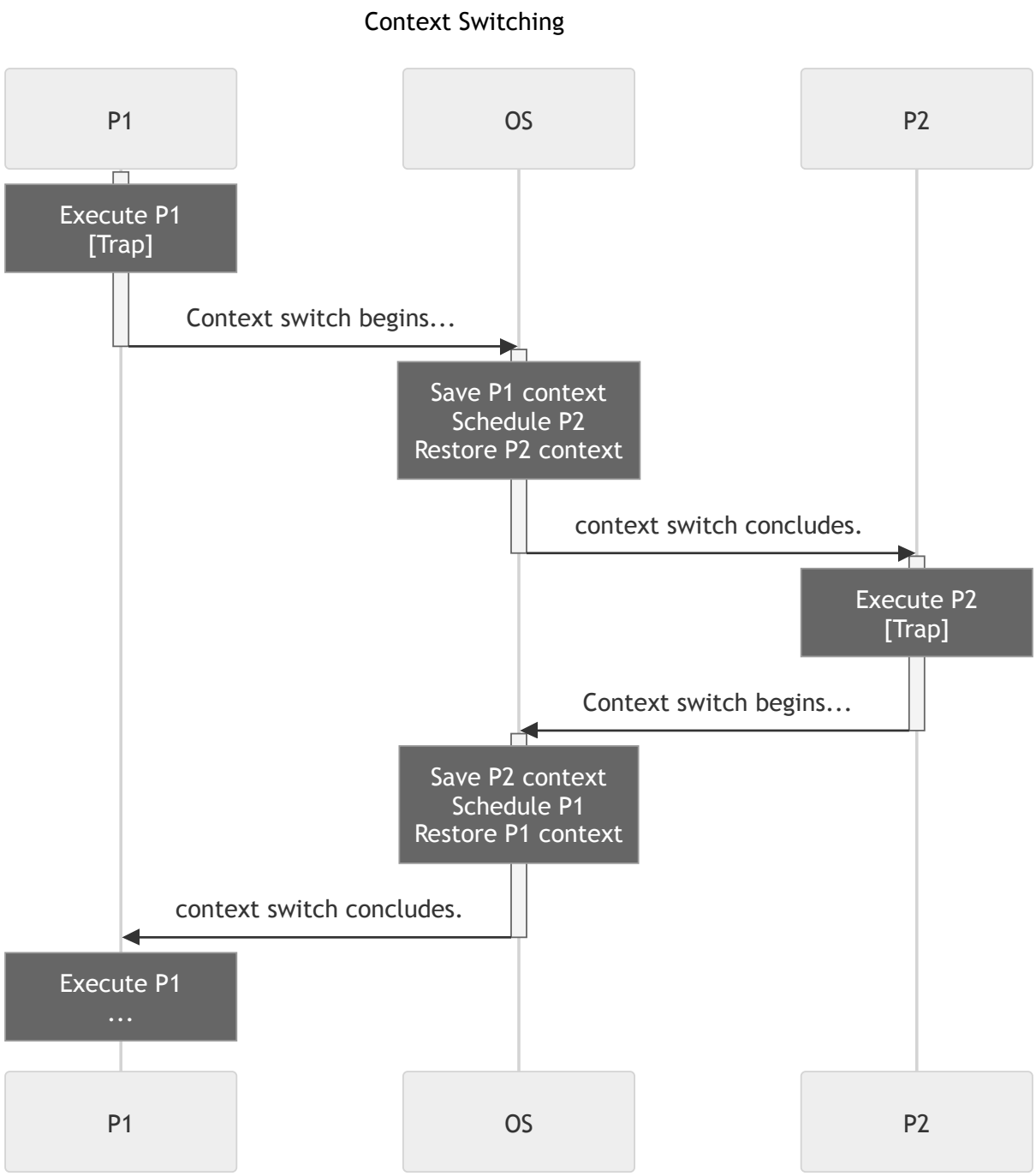
DEFINITION: NONPREEMPTIVE SCHEDULING
A process can be schedule/managed only at transitions (I) and (IV) as indicated above.

This means a process can only be preempted (temporarily suspended by the kernel) when it is blocked or exited/completed.

More on Context Switching

- **Motivation**: As noted last week, one CPU can run one process at a time; how do we run multiple process "at the same time" (multiplexing)?
- The kernel maintains a *context* for each process
  - The context is all the state the kernel needs to restart a pre-empted process, such as:
    - Values in the registers
    - The program counter
    - The user stack
    - status registers
    - kernel's stack
    - other kernel data such as the page table, process table, file table, ...
- **Context switch**: OS stops the running process and switches to another ready process.





Notes

- P1 and P2 have no idea they've been suspended
- The OS (scheduler) decides which process to run next
  - What we're talking about today!
- If context switches happen frequently enough, it will seem to users that P1 and P2 are running "at the same time"

Context switching is not free; there's a cost– it takes time.

- Kernel has to save the current process's context
- Schedule the next process (decide which process is next)
- Load the context of the newly scheduled process
- **Result**: more frequent context switches will lead to worse throughput (higher overhead)

Metrics for Scheduling

As for any process or algorithm, we need to be clear on how we measure them so that we can compare them.

The metrics we'll start with assume the following about a process:

- A process arrives at some time (*arrival time*)
- At some point, it gets scheduled and executes for the first time (*first execution time*)
- At some point, the process completes (*completion time*)

```
timeline ----v---------v------------v---->
          arrival  1st execution   completion
```

DEFINITION: TURNAROUND TIME
The time for each process to complete (completion time - arrival time)

DEFINITION: WAITING/RESPONSE/OUTPUT TIME

- There are variations on this definition, but they all capture the "time spent waiting for something to happen" rather than the "time from launch to exit".
  - **Response time**: time between when job enters system and starts executing - first execution time - arrival time - Our book call's this "response time" - Some books call this "waiting time"
  - **Output time**: time from request to first response to the user (e.g., key press to character echo) - Some books call this "response time" - We won't use this today

DEFINITION: SYSTEM THROUGHPUT
# of processes that complete per unit time

We'll talke about a concept of **"fairness"**, which is intended to capture a perspective on how resources should be used and shared. There isn't one specific definition; here are a couple things to keep in mind. There are concrete metrics that reflect different priorities that are beyond the scope of this course.

- Fairness might include:
  - freedom from starvation (all processes have an opportunity to run on the CPU)
  - all users get equal time on CPU
  - having CPU proportional to resources needed
  - etc.

The Scheduling "Game"

Before we start looking at some different scheduling algorithms, we'll introduce a little "game" to allow us to explore and compare tradeoffs of different scheduling strategies.

Assumptions

- A single CPU
- We know how long each process will run
  - The knowledge of "how long the process will run" is a BIG assumption! It simplifies a lot for now, but doesn't get us anywhere that is practical.
- Ignore context switch costs
  - BUT: only as long as they are not happening too frequently
- Assume no I/O
  - (Completely unrealistic but lets us build intuition)
  - We'll loosen this assumption later tonight

Game input

- A set of processes, with IDs, arrival times and total run time

Game output

- A sequence of scheduling decisions: a timeline of what process runs at which point in time.

Scheduling Algorithms

We'll look at a couple of different approaches to scheduling by playing the Scheduling "Game".

FCFS/FIFO

- **FCFS**: First Come, First Served
- **FIFO**: First In, First Out

With FCFS or FIFO, run each job in the order arrived, and run each job until it's done.

Example: FCFS

We're given the following processes:

process	arrival	running
P1	0	24
P2	0+ϵ	3
P3	0+2ϵ	3

NOTE: The ϵ is an amount of time that is big enough to ensure that we know P1 arrives before P2, but it's small enough that P2 arrives in the same time slice as P1. (In fact, ϵ is small enough that 2ϵ is in the same time slice as P0). In other words, P1, P2 and P3 essentially arrive at the same time, but definitely P1 arrived before P2 which arrived before P3.

Arriving Process	P1 P2 P3															
Time ->	0	1	2	3	...	20	21	22	23	24	25	26	27	28	29	30
Running Process	P1	P1	P1	P1	P1	P1	P1	P1	P1	P2	P2	P2	P3	P3	P3	-

- average turnaround time?
  - P1 finishes at the end of time slice 23, so completion time is 24 - start time 0 ==> P1 turnaround time = 24
  - P2 finishes after time slice 26, and arrived at time slice 0 ==> 27 - 0 = 27
  - P3: 30 - 0 = 30.
  - Avg Turnaround time: $(24 + 27 + 30)/3 = 27$
- average response time?
  - P1: starts executing at time slice 0, which is when it arrived: 0 - 0 = 0
  - P2: starts executing at time slice 24: 24 - 0 = 24
  - P3: 27 - 0 = 27
  - Avg response time: $(0 + 24 + 27)/3 = 17$

QUESTION

Question: if the arrival sequence is `P2, P3, P1`, what will the average turnaround time and response time be?

FCFS ADVANTAGES

- It's simple!
- There is no starvation– all processes eventually get run
- There are few context switches– only 1 per process

FCFS DISADVANTAGES

- Short jobs can get stuck behind long ones– we're not necessarily using our resources efficiently.

SJF and STCF

- **SJF**: shortest job first
  - Schedule the shortest job first
  - Non-premptive: Won't interrupt a process until it's ready to be
- **STCF**: shortest time-to-completion first
  - STCF is the preemptive version of SJF: if job arrives that has a shorter time to completion than the remaining time on the current job, "immediately" preempt CPU to give to new job
- The big idea:
  - get short jobs out of the system
  - big (positive) effect on short jobs, small (negative) effect on large jobs

Example: SJF vs STCF

We have the following processes:

Process ID	Arrival Time	Running Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Turnaround:
P1: 7
P2: 10
P3: 4
P4: 11
=> 8

Response:
P1: 0
P2: 6
P3: 3
P4: 7
=> 4

Using SJF (non-pre-emptive):

Arriving Process	P1		P2		P3	P4										
Time ->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Running Process	P1	P1	P1	P1	P1	P1	P1	P3	P2	P2	P2	P2	P4	P4	P4	P4

Completed Scheduling Table (SJF)

QUESTION
Question: what is the average turnaround time for STCF?

Answer: ???

QUESTION
Question: what is the average response time for STCF?

Answer: ???

Turnaround Time:
P1: 16-0=16
P2: 7-2=5 => 7
P3: 5-4=1
P4: 11-5=6

Response Time:
P1: 0
P2: 0 => 0.5
P3: 0
P4: 2

Using STCF (pre-emptive):

Arriving Process	P1		P2		P3	P4										
Time ->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Running Process	P1	P1	P2	P2	P3	P2	P2	P4	P4	P4	P4	P1	P1	P1	P1	P1

Completed Scheduling Table (STCF)

QUESTION
Question: what is the average turnaround time for STCF?

Answer: $(16 + 5 + 1 + 6)/4 = 7$

QUESTION
Question: what is the average response time for STCF?

Answer: $(0 + 0 + 0 + 2)/4 = 0.5$

- We'll discuss advantages and disadvantages to after we relaxat the "no I/O" assumption

Round-robin (RR)

Round-robin executes one job for one time slice, then the next job in the queue for one time slice, and so on until all jobs have been fully executed.

- add timer
- preempt CPU from long-running jobs. per time slice (also called quantum)
- after time slice, go to the back of the ready queue
- most OSes do something of this flavor (see MLFQ below)

NOTE: RR allows us to start improving response time!

Example: RR

Assume we're using a time slice of 1 unit

P1 P2 - P1 - P2 - P1 - P2 . . . .

Processes:

Process ID	Arrival Time	Running Time
P1	0	50
P2	0	50

P1 - P2 - P3 - P1 - P2 - P3 . . .

Schedule:

Arriving Process	P1		P2		P3	P4										
Time ->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Running Process	P1	P2	P1	P2	A	P2	. .	~	)							

P1 P1 P2 P2 P1 A P2 P2

Completed Scheduling Table (RR)

QUESTION
Question: average turnaround time:

(99+100) /2 = 99.5

QUESTION
Question: average response time:

(0+1) / 2 = 0.5

ADVANTAGES RR

- fair allocation of CPU across jobs
- low average response time when job lengths vary
- good for output time if small number of jobs

DISADVANTAGES RR

- what if jobs are same length?
- in the example: 2 jobs of 50 time units each, a slice of 1
- average turnaround time: 100 (vs. 75 for FCFS)

Choosing the slice size

- want much larger than context switch cost
- pick too small, and spend too much time context switching
- pick too large, and response time suffers (extreme case: system reverts to FCFS)
- typical time slice is between 10-100 milliseconds.
  - context switches are usually microseconds or tens of microseconds (maybe hundreds)

Priority

- **Priority Scheme**: give every process a number (set by administrator), and give the CPU to the process with the highes priority (which might be the lowest or highest number, depending on the scheme)
- can be done preemptively or non-preemptively

Example: Priority

Given the following processes:

Process ID	Arrival Time	Running Time
P1 (high)	0	10
P2 (low)	0	5

- schedule: Execute P1 to completion, then P2

QUESTION
Question: average turnaround time:

(10 + 15) / 2 = 12.5

QUESTION
Question: average response time:

(0+10) / 2 = 5

- if we swap the priorities on P1 and P2 (P2 is high; P1 is low)

QUESTION
Question: average turnaround time:

(5+15) / 2 = 10

QUESTION

> Question: average response time:
>
> (0+5) / 2 = 2.5

- generally a bad idea to use strict priority because of starvation of low priority tasks.
  - **NOTE**: SJF is priority scheduling where priority is the predicted next CPU running time
- solution to this starvation is to increase a process's priority as it waits

Multi-level feedback queue (MLFQ)

- combine RR and Priority

Three big ideas:

- multiple queues, each with different priority. 32 queues, for example.
- round-robin within each queue (flush a priority level before moving onto the next one).
- feedback: process's priority changes, for example based on how little or much it's used the CPU
  - you can design different feedback mechanisms
  - one example: "downgrade" a process after 5 unit of time running
    - The book refers to this as an "allotment" of CPU cycles/time slices

Example MLFQ

Specs:

- 2 queues, each using RR
- When a process arrives, it goes to the tail of the high priority queue
- Time slice is 1 unit of time (in both queues)
- After a process has run for 2 units of time, it is moved to the tail of the low-priority queue

Our processes:

→ In High-Priority Queue.

Process ID	Arrival Time	Running Time
P1	0	4
P2	0	4
P3	4	4

Turnaround:
P1: 10
P2: 11
P3: 8
⇒ 29/3 = ~10

Response:
P1: 0
P2: 1
P3: 0
⇒ 1/3 ≈ 0.3

schedule:

Queue 2 (low)				P1	P1 P2		P1 P2 P3				P2 P3	P3				
Queue 1 (high)	P1 P2			P2	P3											
Time ->	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Running Process	P1	P2	P1	P2	P3	P3	P1	P2	P3	P1	P2	P3	–			

Completed Scheduling Table (MLFQ)

ADVANTAGES

- approximates STCF

DISADVANTAGES

- can't donate priority
- not very flexible
- not good for real-time and multimedia
- can be gameable: user puts in meaningless I/O to keep job's priority high

Notes for MTFQ

MTFQ has many ways it can be configured or parameterized:

- Which scheduling algorithm to use for each queue (we used RR with 1 time slice).
- The allotment, or when to demote a process to a lower priority queue (we used running time).
- When to promote a process to a higher priority queue to help prevent starvation (we didn't do this at all!)
- The number of queues (we used 2)
- How a process gets queued when it arrives (we put it in the tail of the high-priority queue)
  - Could put either at the head or tail of a queue, in any of the queues available

Lottery and stride scheduling

Citations:

- C. A. Waldsburger and W. E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. Proc. Usenix Symposium on Operating Systems Design and Implementation, November, 1994. http://www.usenix.org/publications/library/proceedings/osdi/full_papers/waldspurger.pdf
- C. A. Waldsburger and W. E. Weihl. Stride Scheduling: Deterministic Proportional-Share Resource Management. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, June 1995. http://www.psg.lcs.mit.edu/papers/stride-tm528.ps
- Carl A. Waldspurger. Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management, Ph.D. dissertation, Massachusetts Institute of Technology, September 1995. Also appears as Technical Report MIT/LCS/TR-667. http://waldspurger.org/carl/papers/phd-mit-tr667.pdf

Issue lottery tickets to processes

- Let p_i have t_i tickets
- Let T be total # of tickets, $T = \sum t_i$
- Chance of winning next slice is t_i/T
- Note lottery tickets not used up, more like season tickets

EXAMPLE

```
(with slice of 1 unit of time)
process      arrival     running
P1 (t_1=20)    0           20
P2 (t_2=10)    0           10
```

- schedule (many possibilities; for example, P2 P2 P2 P2 ... P1 P1 P1 ...)

QUESTION

Question: expected turnaround time for (P1, P2):

- A. (20, 30)
- B. (30, 20)
- C. (30, 30)

Answer: C

DEFINITION: EXPECTED TURNAROUND TIME

expected turnaround time = (running time) / (expected running time in each slice)

- P1: 20 / (20/30) = 30
- P1: 10 / (10/30) = 30

controls long-term average proportion of CPU for each process

can also group processes hierarchically for control

- subdivide lottery tickets
- can model as currency, so there can be an exchange rate between real currencies (money) and lottery tickets
- lots of nice features
  - deals with starvation (have one ticket –> will make progress)
  - don't have to worry that adding one high priority job will starve all others
  - adding/deleting jobs affects all jobs proportionally (T gets bigger)
  - can transfer tickets between processes: highly useful if a client is waiting for a server. then client can donate tickets to server so it can run.
  - note difference between donating tix and donating priority. with donating tix, recipient amasses enough until it runs. with donating priority, no difference between one process donating and 1000 processes donating
- many other details
  - ticket inflation for processes that don't use their whole slice
  - use fraction f of slice; inflate tix by 1/f until itnext gets CPU

DISADVANTAGES

- latency unpredictable
- expected error somewhat high
- for those comfortable with probability: this winds up being a binomial distribution. variance n*p*(1-p) –> standard deviation \proportional \sqrt(n),
  - where: - p is fraction of tickets owned - n is number of quanta
- in reaction to these disadvantages, Waldspurger and Weihl proposed *Stride Scheduling*
  - basically, a deterministic version of lottery scheduling. * less randomness –> less expected error.
  - see OSTEP (chap 9) for details

a note about Lottery algo

- randomness is important in CS, and lottery scheduling is one example
  - randomized algorithms in theory
  - stochastic gradient descent in deep learning
  - BitCoin's proof-of-work; Ethereum's proof-of-stake

ADVANTAGES

- can model as currency, so there can be an exchange rate between real currencies (money) and lottery tickets
- deals with starvation (have one ticket –> will make progress)
- don't have to worry that adding one high priority job will starve all others
- adding/deleting jobs affects all jobs proportionally (T gets bigger)
- can transfer tickets between processes: highly useful if a client is waiting for a server. then client can donate tickets to server so it can run.
- many other details
  - ticket inflation for processes that don't use their whole slice (like feedback in MLFQ)
  - use fraction f of slice; inflate tix by 1/f until it next gets CPU (improve fairness)

DISADVANTAGES

```
* latency unpredictable
* expected error somewhat high
```

- in reaction to these disadvantages, Waldspurger and Weihl proposed *Stride Scheduling*
  - basically, a deterministic version of lottery scheduling. less randomness –> less expected error.
  - see OSTEP (chap 9) for details

Vote for your favorite scheduling algorithm

Candidates: FIFO, STCF, RR, Prio, MLFQ, and Lottery

Winners (with votes):

"Best Turnaround Time" winner: STCF (48)

"Best Response Time" winner: RR (34)

"Best Fairness" winner: RR (33)

"Most popular algorithm" winner: MLFQ (21)

cheng's choices:

"Best Turnaround Time" winner: STCF "Best Response Time" winner: RR "Best Fairness" winner: Lottery "Most popular algorithm" winner: Lottery

- different winners and (sort of) diverse answers indicates:
- no "best" scheduling algorithms
- metric-oriented (or usually called workload-oriented) algorithms * we can game the metrics by not-very-useful algorithms * "best response time" algo: FIFO + always scheduling the new arrival process for 0.0001 sec

Incorporating I/O

(relax assumption from before)

motivating example:

- 3 jobs
  - P1, P2: both CPU bound, run for a week
  - P3: I/O bound, loop (1 ms of CPU, 10 ms of disk I/O)

process arrival running P1 0 1 week P2 0 1 week P3 0 30 sec (with 300sec I/O)

Process ID	Arrival Time	Running Time
P1	0	1 week
P2	0	1 week
P3	0	30 sec (with 300sec I/O)

- by itself, P3 uses ~90% of disk
- By itself, P1 or P2 uses 100% of CPU

QUESTION
what happens if we use FIFO? (arrival: P1, P2, P3)

- P1 and P2 will run, keep CPU for 2 weeks

QUESTION
what about RR with 100msec time slice?

- only get ~5% disk utilization

```
CPU:  [P1:100] [P2:100] [P3:1] [P1:100    ]...
disk: [---------idle---------] [P3:10] [--]...
```

QUESTION
what about RR with 1msec time slice?

- get nearly 90% disk utilization
  - but lots of preemptions

```
CPU:  [P1:1] [P2:1] [P3:1] [P1:1] [P2:1] [P1:1]...
disk: [------------------] [P3: 10            ...
```

QUESTION

what about STCF?

- would give us good disk utilization ...

```
CPU:  [P3:1] [P1:10] [P3:1] [P1:10]...
disk: [----] [P3:10] [----] [P3:10]...
```

(what about P2? starvation)

ADVANTAGES TO STCF

- disk utilization (see above)
- low overhead (no needless preemptions)

DISADVANTAGES TO STCF

- long-running jobs can get starved
- requires predicting the future

Predicting future performance

MTQF is an example of an algorithm that tries to predict how a process is going to perform in the future, and attempts to use past behavior as a predictor of future performance in order to be most efficient or fair in the use of resources.

What can be used to help us predict the future?

- Exponentially weighted moving average (EWMA) a good idea
  - t_n: length of proc's nth CPU running time
  - τ_{n+1}: estimate for n+1 running time
  - choose α, 0 < α <= 1
  - set $\tau_{n+1} = \alpha * t_n + (1 - \alpha) * \tau_n$
  - reacts to changes, but smoothly

How Linux Schedules

- the current Linux scheduler (post 2.6.23), called CFS (Completely Fair Scheduler), roughly reinvented the ideas of Stride Scheduling
- If you're interested in CFS: "Inside the Linux 2.6 Completely Fair Scheduler"

[maybe talk about "design vs. implementation"]

Scheduling Problems in Modern Systems

- you may have an illusion that scheduling is "easy"
- a problem in practice: Reconfigurable Machine Scheduling Problem (RMS)
- scheduling problem in practice:
  - NVIDIA A100 GPU has a new feature: Multi-Instance GPU (MIG)
  - MIG can partition one GPU into 7 small GPUs (call it 1/7 instance)
  - small GPUs can merge into larger ones: two 1/7 instances => one 2/7 instance (with limitations)

- different deep neural network (DNN) jobs have non-linear performance on different sized instances
  - for example, inceptionv3, two 1/7 instances have higher throughputs than 2/7
- Question: give a set of DNN jobs and a set of GPU, how to maximize the throughput? (or minimize #GPUs used)
  - this is an NP-complete problem
  - see some viable solution in this NEU paper:
    - "Serving DNN Models with Multi-Instance GPUs: A Case of the Reconfigurable Machine Scheduling Problem" https://arxiv.org/pdf/2109.11067.pdf
- point: scheduling problem becomes complicated and important in new era (e.g., cloud computing and AI).

Summary

- Scheduling comes up all over the place
- *m* requests share *n* resources
  - disk arm: which read/write request to do next?
  - memory: which process to take physical page from?
- This topic was popular in the days of time sharing, when there was a shortage of resources all around, but many scheduling problems become not very interesting when you can just buy a faster CPU or a faster network.
- **Exception 1:** web sites and large-scale networks often cannot be made fast enough to handle peak demand (flash crowds, attacks) so scheduling becomes important again. For example may want to prioritize paying customers, or address denial-of-service attacks.
- **Exception 2:** real-time systems: soft real time: miss deadline and CD or MPEG decode will skip hard real time: miss deadline and plane will crash

Plus, at some level, every system with a human at the other end is a real-time system. If a Web server delays too long, the user gives up. So the ultimate effect of the system may in fact depend on scheduling!

- Cloud computing (or huge datacenters) makes scheduling "fancy" again... ...with new problems (like RMS) ...and "scale effect": improving 1% CPU efficiency on you laptop is not interesting, whereas saving 1% CPU time for Google is a BIG deal!
- In principle, scheduling decisions shouldn't affect program's results
- This is good because it's rare to be able to calculate the best schedule
- So instead, we build the kernel so that it's correct to do a context switch and restore at any time, and then *any* schedule will get the right answer for the program
- This is a case of a concept that comes up a fair bit in computer systems: the policy/mechanism split. In this case, the idea is that the *mechanism* allows the OS to switch any time while the *policy* determines when to switch in order to meet whatever goals are desired by the scheduling designer
- But there are cases when the schedule *can* affect correctness
  - multimedia: delay too long, and the result looks or sounds wrong
  - Web server: delay too long, and users give up

Next time

Threads!