PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

CPU Scheduling II

Baskin Engineering UC SANTA CRUZ

Notices

- **Assignment 2** due 23:59 **Sunday February 2**
- **Lab 2** due 23:59 **Sunday February 9**
- No assignment next week, so plenty of time for Lab 2 ☺

Today's Lecture

- Recap
  - Waiting Time Calculation
  - Simple Scheduling Algorithms: FCFS and RR
- Shortest Job First (non-preemptive & preemptive)
- Advantages and Disadvantages of FCFS, RR, and SJF
- Multilevel Queue
- Multilevel Feedback Queue
- Scheduling in Pintos
- Priority Donation

Waiting Time (for any scheduling algorithm)

For a thread **T** that gets scheduled **N** times during it's lifetime:

arrival = time **T** initially enters the scheduling queue

start$_n$ = time **T** gets the CPU for the **n**th time

finish$_n$ = time **T** yields or is forced to yield the CPU for the **n**th time

$$\text{Waiting Time} = (\text{start}_1 - \text{arrival}) + \sum_{n=2}^{N} (\text{start}_n - \text{finish}_{n-1})$$

Simple Scheduling Algorithms

- **FCFS**: Bad guys finish first ☹
  - Threads that hog the CPU tend to prevent others from getting a chance to execute
- **RR**: Nice guys finish first ☺
  - Threads that do modest chunks of work before yielding or waiting for IO get more and earlier chances to use the CPU
- In time-sharing situations, RR is usually preferable
- But can we do better than Round Robin?

Shortest Job First

- Burst lengths seem to be critical, so…
- Select the ready job (process or thread) with shortest CPU burst
- When a new thread arrives with a shorter burst than the currently running job (the current thread) SJF can be:
  - **Non-preemptive**
    - Don't preempt current thread, let it complete this CPU burst, then run the newly arrived thread, unless… **(unless what ?)**
  - **Preemptive**
    - If arriving thread has CPU burst less than remainder of the current job's current burst, preempt (i.e. kick current thread off CPU and run the newly arrived thread)
    - Sometimes called **"Shortest Remaining Time First"** (SRTF)
- **Preemptive SJF** is optimal for time-sharing systems
  - Gives minimum mean waiting time for any given set of threads
    - **What other scheduling metric is likely (but not guaranteed) to be minimised?**

Non-Preemptive SJF

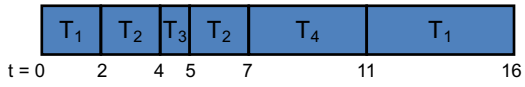
	$burst_1$	t_{arrive}
T_1	7	0
T_2	4	2
T_3	1	4
T_4	4	5

Mean waiting time = ((0-0) + (8-2) + (7-4) + (12-5)) / 4 = **4**

Mean turnaround time = ((7-0) + (12-2) + (8-4) + (16-5)) / 4 = **8**

Preemptive SJF

	$burst_1$	t_{arrive}
T_1	7	0
T_2	4	2
T_3	1	4
T_4	4	5



T_1 waiting time = (0-0) + (11-2) = 9
T_2 waiting time = (2-2) + (5-4) = 1
T_3 waiting time = 4-4 = 0
T_4 waiting time = 7-5 = 2

Mean waiting time = (9 + 1 + 0 + 2) / 4 = **3** (was 4)

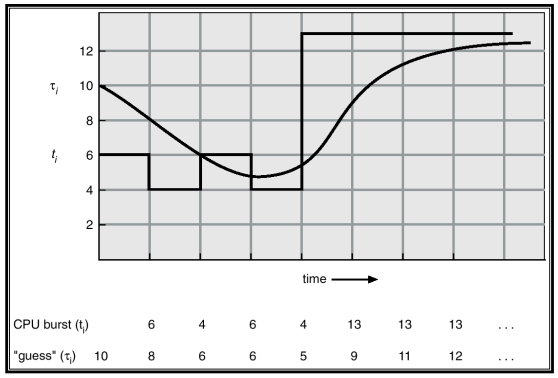
Mean turnaround time = ((16-0) + (7-2) + (5-4) + (11-5)) / 4 = **7** (was 8)

Unfortunately...

- **SJF needs to see into the future**
- Not currently possible ☹
- So we have to make a guess, but how?

- We don't actually know what the next CPU burst will be, so we predict it based on what happened in the past:
  - **Decaying weighted mean**, $\alpha = \{0 \rightarrow 1\}$
    $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau$
  - **Exponential mean**
    $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \alpha\, t_{n-1} + \ldots + (1 - \alpha)^j\, \alpha\, t_{n-j} + \ldots + (1 - \alpha)^{n+1}\, \tau_0$

CPU Burst Prediction



(Example of Decaying Weighted Mean)

Comparing Scheduling Algorithms

- **First Come First Served**
  - Pros: Simple
  - Cons: Short Jobs can get stuck behind long ones
- **Round Robin**
  - Pros: Better for short jobs (especially if they are shorter than the quantum)
  - Cons: Not great when jobs are same length (waste time context switching)
- **Shortest Job First**
  - Pros: Optimal for mean waiting time (i.e good for time-sharing)
  - Cons: Hard to predict the future - unfair on long jobs?

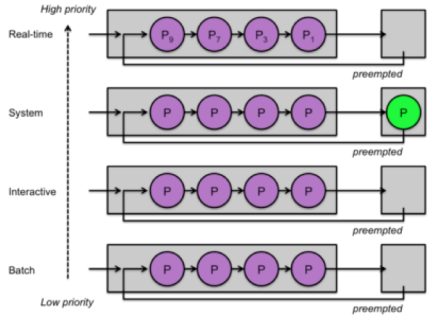
Multilevel Queue (a.k.a "multiple queues")

- Recognize that threads can be classified into different groups
- The single ready queue becomes, say, two queues:
  - Foreground (interactive)
  - Background (batch)
- Each queue can have it's own scheduling algorithm, for example:
  - Foreground - RR
  - Background - FCFS

Priority Scheduling

- A **priority** is associated with each thread
  - SJF is a priority scheduler that uses the predicted CPU burst as the priority value
  - Alternatively, we can explicitly assign priorities
- Priority determines which ready queue the thread will join
- CPU is allocated to the thread with the highest priority, as usual, there are two types:
  - **Preemptive**
  - **Non-preemptive**
- Simplest form is **Fixed Priority Scheduling**
  - Serve all from highest priority queue, then all from next highest priority queue, and so on

Multiple Queues for Multiple Priorities



One problem is that the process needs to be assigned to the most suitable priority queue *a priori*

If a CPU-bound process is assigned to a short-quantum, high-priority queue, that's suboptimal for the process and for overall throughput

Remember: Only one CPU - Diagram is not the best ☹

Multi Level Queues : Problems?

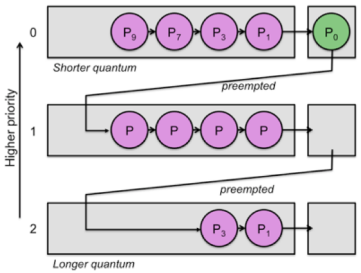
- There is a possibility of starvation
- One solution is to **queue time slice** - each ***queue*** gets a certain minimum proportion of CPU time which it can schedule amongst its threads, for example:
  - 80% to foreground in RR
  - 20% to background in FCFS
- Another solution is to apply **runtime feedback**

Multilevel Feedback Queue

- Introduce scheduling to move jobs between queues
- A **multilevel feedback queue** scheduler is defined by:
  - Number of queues
  - Scheduling algorithms for each of the queues
  - Methods used to determine:
    - **Initial** queue for thread
    - When to **promote** thread
    - When to **demote** thread

A Simple Queue Demotion Scheme

- Priority is lowered when quantum exceeded
- Q_0 drained before $Q_{>0}$
  - This discriminates against long jobs
  - But improves life for shorter jobs
- Each queue may have its own scheduling algorithm



Remember: Only one CPU - Diagram is not the best ☹

Queue Demotion : Problems?

- In general, multi-level feedback queues favour I/O bound processes by keeping them at high priorities
- But... what would happen if an I/O intensive process such as a text editor first needed to perform some CPU-intensive work such as initializing a lot of structures and tables?
  - Although it would get scheduled at a high priority initially, it would find itself using up its quantum and demoted to a lower level
  - There too, it may not be done with its tasks and be demoted to yet lower levels before it blocks in I/O
  - Now it will be doomed to run at a low priority level, providing potentially inadequate interactive performance ☹

Queue Promotion

- **Process Aging** can rescue an interactive program that had a period of CPU-intensive activity and was "punished" down to a lower priority queue
  - The unfortunate process in promoted back to a higher priority when it gets older than some threshold
- Alternatively, we can ***raise*** the priority of a process whenever it becomes blocked on I/O

Pintos Scheduling

- Pintos implements a single ready queue with a quantum
- In Lab 2, you are implementing a priority scheduler ☺
- There are 64 priorities, default is 31, 0 is lowest
- Threads can change priorities
- Every queue (ready, and any related to concurrency primitives like semaphores and condition variables) must reflect these priorities when removing (or adding) elements
- **General principle is that no higher priority thread should have to wait while a lower priority thread executes**
- **Preemption is super important, so get it right!**
- But priority donation is problematic ☹

Priority Donation : Example Pseudo Code

```
Thread A (priority 20) {
  acquire Lock
  create  Thread B
  create  Thread C
  release Lock
}

Thread B (priority 40) {
  acquire Lock
  release Lock
}

Thread C (priority 30) {
  // nothing
}
```

High priority threads should run to completion before those with lower priorities.

In this example, **B** gets blocked by the lower priority **A** which is holding the shared lock.

C has a higher priority than **A,** so once **C** starts running (without priority donation), **A** will not be able to run until **C** completes, at which point **A** will release the lock and **B** will have a chance to run.

Desired completion order is **B**, **C**, **A**.

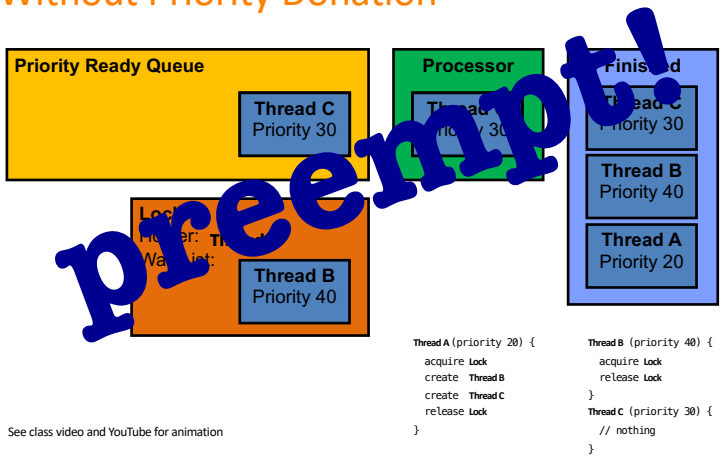
Without priority donation, completion order is **C**, **B**, **A** ☹


```
// src/tests/threads/priority-donate-single.c

static struct lock lock;

void test_priority_donate_single(void)
{
    thread_set_priority(20);

    lock_init(&lock);
    lock_acquire(&lock);

    thread_create("Thread B", 40, b_thread_func, NULL);
    thread_create("Thread C", 30, c_thread_func, NULL);

    lock_release(&lock);
    msg("Thread A finished.");
}

static void b_thread_func(void *aux UNUSED) {
    lock_acquire(&lock);
    lock_release(&lock);
    msg("Thread B finished.");
}

static void c_thread_func(void *aux UNUSED) {
    msg("Thread C finished.");
}
```



```
// src/tests/threads/priority-donate-single.ck

# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(priority-donate-single) begin
(priority-donate-single) Thread B finished.
(priority-donate-single) Thread C finished.
(priority-donate-single) Thread A finished.
(priority-donate-single) end
EOF
pass
```


Without Priority Donation



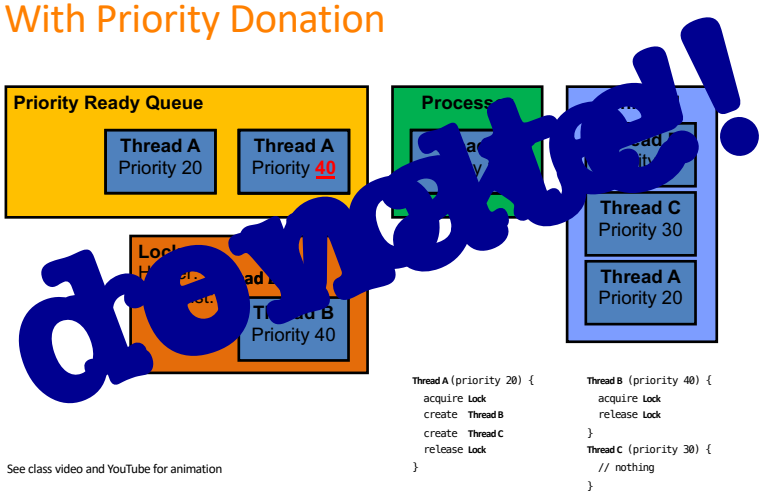

```
Thread A (priority 20) {
  acquire Lock
  create  Thread B
  create  Thread C
  release Lock
}
```

```
Thread B (priority 40) {
  acquire Lock
  release Lock
}
Thread C (priority 30) {
  // nothing
}
```

See class video and YouTube for animation

```
// src/tests/threads/priority-donate-single.ck

# -*- perl -*-
use strict;
use warnings;
use tests::tests;
check_expected ([<<'EOF']);
(priority-donate-single) begin
(priority-donate-single) Thread B finished.
(priority-donate-single) Thread C finished.
(priority-donate-single) Thread A finished.
(priority-donate-single) end
EOF
pass
```


With Priority Donation



See class video and YouTube for animation

Next Lecture

- Case Study: 4.4 BSD Unix
- Multi Core & Multi Processor Scheduling
- Lab 2 Secret Sauce