Scheduling

Module 09

Reading

This module covers Chapter 2.4 in the text.

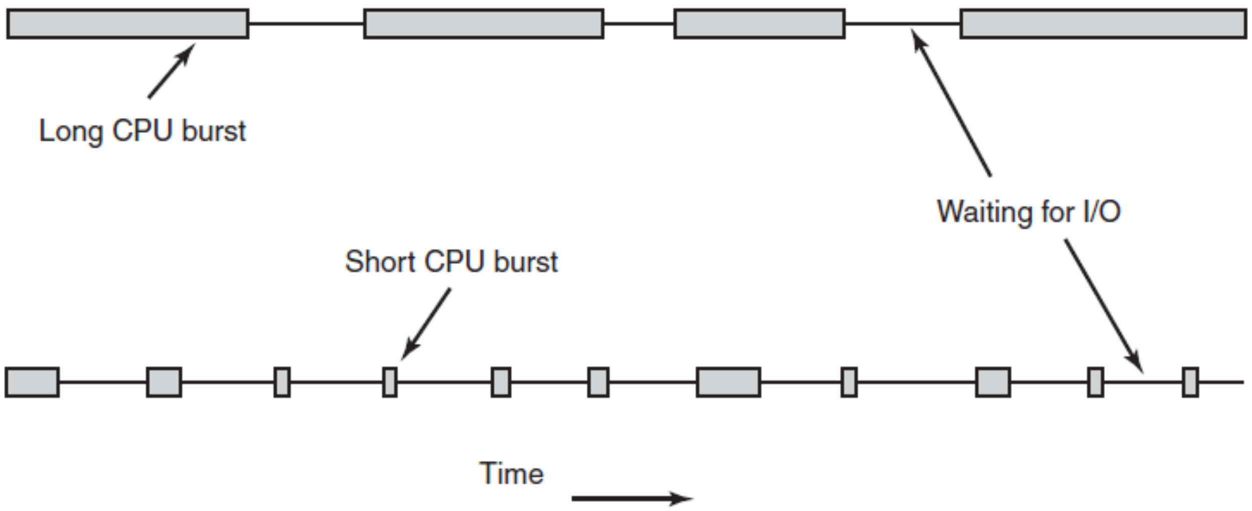
Man behind the curtain...

We’ve been writing synchronization code in a state of paranoia - because we never know when the schedule is going to interrupt us!

Now we will look at how the schedule actually makes these decisions, and how deterministic it really is.

What are we scheduling?

All processes alternate between CPU instructions and an I/O call



CPU - bound process

I/O - bound process

Categorizing processes

I/O Bound programs don’t compute much between I/O calls - little CPU usage

CPU Bound programs have lots of calculations between I/O calls

The length of the I/O is irrelevant

What are we actually scheduling?

- We are scheduling CPU-bursts.

- Once an I/O call is made, the process is blocked and we schedule another **ready** process’s CPU burst

When does the scheduler schedule?

- Its critical to understand that the schedule can only run when the OS is on the CPU
- It must make a decision who should run next, and set the PC appropriately.
- Once the PC is set - the schedule is ***not running.***

When can the scheduler run?

- Once a process starts running, the OS won't run until one of the following occurs:
  - The process makes a system call like **`fork`** or **`exit`**
  - The process yields voluntarily (**sleep**, **`yield`**)
  - The process issues a blocking call (**read**)
  - An interrupt occurs (pending I/O completes)

```
while ( true );    // oh... no.
```

Preemptive Scheduling

- **Nonpreemptive** systems are at the mercy of processes triggering a scheduling opportunity.
- **Preemptive** systems employ a **hardware timer** to issues an **interrupt** after a time period.
  - This time period is called a ***quantum***
  - The schedule can set the timer before starting a process.

When can the scheduler run?

- Once a process starts running, the OS won't run until one of the following occurs:
  - The process makes a system call like `fork` or `exit`
  - The process yields voluntarily (**sleep**, `yield`)
  - The process issues a blocking call (**read**)
  - An interrupt occurs (pending I/O completes)
  - A timer interrupt goes off (quantum expiration)

Measuring effectiveness

- A scheduling **algorithm** is a procedure for deciding which process runs next
- If we are going to *evaluate* scheduling algorithms, we need ***metrics***
  - CPU Utilization
  - Throughput
  - Turnaround Time
  - Wait Time
  - Response Time
  - Fairness

Different types of systems need to optimize different metrics

Measuring effectiveness

- A scheduling **algorithm** is a procedure for deciding which process runs next
- If we are going to *evaluate* scheduling algorithms, we need ***metrics***
    - CPU Utilization
    - Throughput
    - Turnaround Time
    - Wait Time
    - Response Time
    - Fairness

Different types of systems need to optimize different metrics

We're going to defer "priority" for a moment...

Types of Systems

Batch Systems: Payroll systems, banking systems, data collection tasks..

Interactive Systems: Likely every computer you've ever used.

Real-Time Systems: Missile guidance, satellite control, and other silly programs.

Batch System Priorities

Maximize Throughput

Minimize Turnaround

These are clearly linked - but not exactly the same.

Throughput can be achieved by ignoring long jobs!

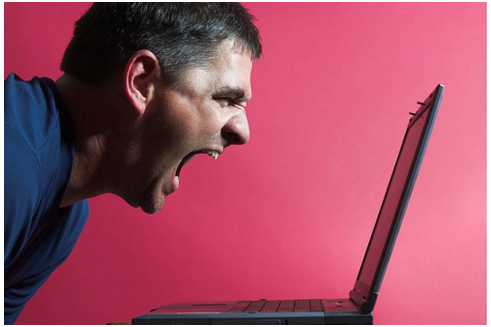
Keep CPU utilization high

Drives Throughput/Turnaround

The CPU should never be idle
Minimize time OS is on CPU

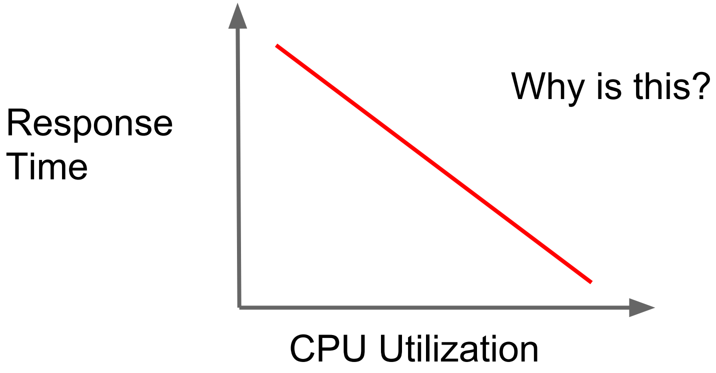
Interactive Systems

- When users are clicking and typing, priorities change.
    - Users was **their program** to run **now.**
    - Overall system performance is not quite as important as **foreground** process performance
    - Ability to switch tasks is **critical**

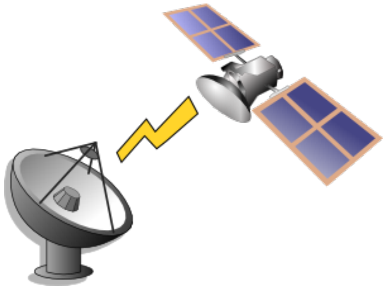


Interactive Systems

- Interactive Systems should minimize **wait time** and minimize **response time**.
- As we will see, response time can only be minimized at the expense of CPU utilization… and we know what that means!



Real-time Systems



- Real-time <u>does not mean</u> “super fast”
- Real time means you have very specific times when certain things need to execute
  - It might be every ten minutes, run a 10 second task - the point is **<u>we don’t mean “around”</u>** every ten minutes!
- Real-time systems are another animal, we’re going to leave them alone for now

Scheduler Implementation

- Processes are modeled by entries in a **process table / process control block**
    - Entries contain *accounting information*, along with other information about *restarting* the process
    - The schedule maintains a **ready queue** of all processes ready to run.

- When the schedule runs, it consults the table and its own records and picks a process
    - It restores registers/PC/stack etc of the chosen process.
    - This is called a **context switch**.

First Come, First Serve

Similar to FIFO queue, this algorithm simply picks the *next* process from the **ready queue.**

- **This is a non-preemptive algorithm:**
  - The process selected runs until it blocks, yields, or terminates

Let’s model a hypothetical execution of a set of CPU bursts with FCFS scheduling

Shortest Job First (SFJ)

Is the wait time for FCFS any good? What would be optimal?

- Select the shortest job first
    - Still non-preemptive

Let’s model a hypothetical execution of a set of CPU bursts with SJF scheduling

Why is SJF Optimal?

Let's assume we have four bursts to schedule (A, B, C, and D), run in order

A waits no one	0
B waits for A	A
C waits for A and B	A + B
D waits for A, B, and C	A + B + C
Total Wait Time:	**3A + 2B + C**

What's the shortest job?

- A short job is really a **short CPU burst**
  - Processes with short CPU burst are I/O Bound

Critical Insight:

Prioritize I/O bound processes over CPU bound processes!

- This also makes sense from a resource optimization perspective!

Preemptive SJF

- We can also simulate preemption, and a dynamic ready queue, and still employ SJF
    - Model ***arrival*** times along with CPU bursts
    - Schedule updates burst time when process is interrupted.
    - Remember - new arrivals will constantly enter the ready queue (I/O completions)

Let’s model a hypothetical execution of a set of CPU bursts with “Least Remaining” scheduling

Implementing SJF?


```
while (true) {
    x = random()
    if ( x % 2 == 0 ) {
        read_disk();    I/O (CPU Burst is over)
    }
    else {
        do_math();    CPU burst's reign of terror continues.
    }
}
```

Approximating Shortest Job First

- We could keep accounting data and use the past to predict the future
- Need to be careful, the more time we spend doing this, the worse CPU utilization gets!

Later we'll see how to do this better.

Lets look at FCFS, with preemption

- FCFS is great for batch systems, but bad for interactive - poor task switching
- Round Robin employs FCFS, but uses a **quantum** to preempt running process
- What happens to response time as quantum time changes?
- What happens to CPU utilization?

Let’s model a hypothetical execution of a set of CPU bursts with Round-Robin scheduling

Priorities

- We’ve established that I/O should be given priority over CPU - bound processes
    - I like to call this ***dynamic*** or ***internal*** **priority**
    - It can change as we get a sense of how I/O or CPU-bound the process is.
- However - processes may be ***assigned*** priorities
    - I’ll call this ***static*** or ***external*** **priority**

Priority Scheduling

We can simply always choose the highest priority process

Let's model a hypothetical execution of a set of CPU bursts with **Priority** scheduling

Problem: Starvation

Solution: Aging

Lottery Scheduling

- Priority Scheduling is very deterministic:
    - The low priority process **never** runs!

- Lottery scheduling provides a **probabilistic** result, where high priority task are simply ***likely*** to run.

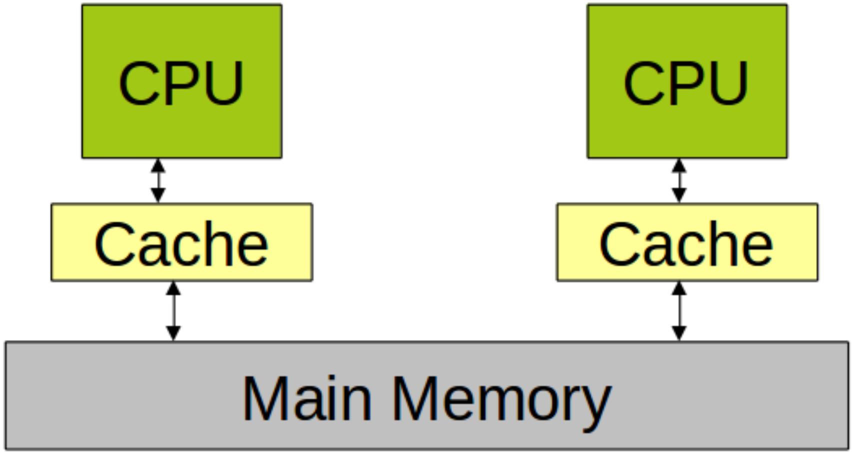
Scheduling multi-core systems

- **Goal:** Load balancing
  - We don't want one core to be idle while a bunch of processes are waiting on another!

- Implementation Options
  - Split-Ready Queue
  - Common-Ready Queue

Processor Affinity

Common-ready queue seems to make sense for load balancing… but wait!



- The desire to keep a process on a specific CPU is called processor **affinity**.
- In practice, this is so important that split ready queues are typically used instead of a common-ready queue.
- **Pull Migration**: If a CPU's ready queue is empty, go look for another ready queue to pull from.

Real World Example: Sun Solaris

- Priority - Based Scheduling
  - Separate Queue for each priority level:
  - Real-time, System, Time Sharing, Interactive
- Time-sharing and Interactive have ***dynamically changing priorities***
  - 0-60, 0 being the highest

Real World Example: Sun Solaris

			Priority after...	
	Priority	Quantum Time	Quantum Expiration	Sleep/IO Completion
Lowest	0	200	0	50
	10	160	0	51
	20	120	15	52
	35	80	25	54
	50	40	40	58
Highest	59	20	49	59

Why penalize quantum completion?

Why boost on I/O completion?

Real World Example: Windows

Priority Scheduling via a “grid”

	real-time	high	above nomal	nomal	below nomal	idle
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above nomal	25	14	11	9	7	5
nomal	24	13	10	8	6	4
below nomal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Real World Example: Windows

Columns are *externally defined through Win32 API - static*

	real-time	high	above nomal	nomal	below nomal	idle
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above nomal	25	14	11	9	7	5
nomal	24	13	10	8	6	4
below nomal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Real World Example: Windows

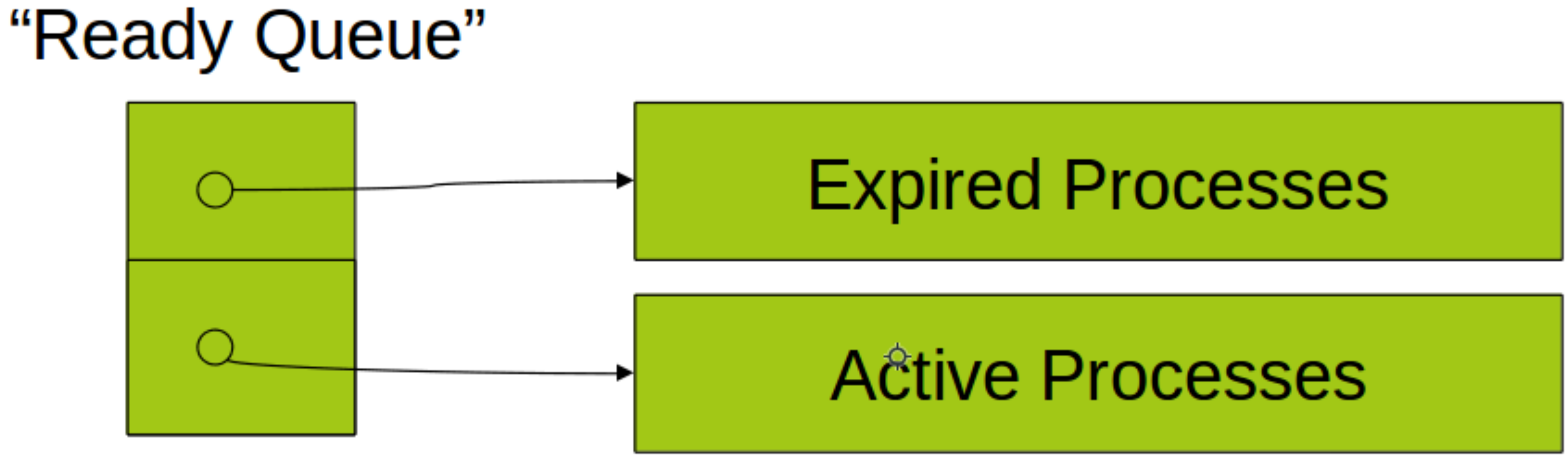
Rows are **dynamically adjusted**

	real-time	high	above nomal	nomal	below nomal	idle
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above nomal	25	14	11	9	7	5
nomal	24	13	10	8	6	4
below nomal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Real World Example: Linux

- Priority based again!
- Priorities are externally defined (only)
  - High priority tasks get larger quantums
  - Processes occupy a **split ready queue**.
    - Expired
    - Active

Real World Example: Linux



- Only process in Active queue are selected
- On pre-emption, kicked into Expired queue
- When active empties, switch pointers

Take-away from Real-World Examples

- Real Operating systems use a **blend** of strategies
  - They all allow external priorities
  - They all favor I/O and penalize CPU
  - They achieve their goals differently!

Next Up...

After Exam 2 we will start memory management

Chapter 3.1-3.2