CIT 5950
Section 7

Scheduling & Virtual Memory

Logistics

- HW2 (Threads) due Next Monday, February 27th @ 11:59pm

- Midterm will be released on Wednesday 3/1 at 12pm and will be available until Friday 3/3 at 12pm. You have the entire 48 hours to work on the exam

- **NO RECITATION NEXT WEEK; NO HOMEWORK OVER SPRING BREAK**
    - Enjoy your well-deserved rest!

Exercise 1 - scheduling

Quick review of scheduling algorithms

- **FCFS (First come first served)**
  - Simple to implement
  - Low throughput, slow response, no priority
- **SJF (Shortest Job First)**
  - Minimal average turnaround time
  - Need to use estimates, no priority, possible starvation
- **Round Robin**
  - Relatively fair
  - Need to choose time quantum correctly, no priority
- **Priority Round Robin**
  - Introduce priority
  - Many design choices (# levels, time quantum, priority assignment)

Exercise 1

Name	Arrival Time	Running Time
Bert	2	11
Ernie	0	8
Oscar	12	20
Grover	7	15
Elmo	10	4

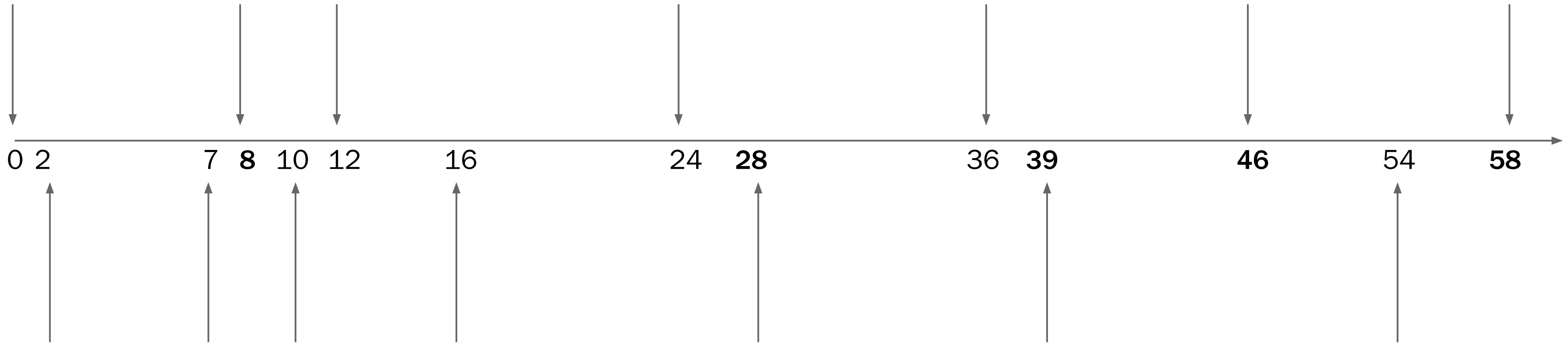
Consider the following set of tasks/processes:

Using the **Round Robin** scheduling algorithm and a time quantum/slice of 8, **what is the finishing time for each?**

What is the **average waiting time?**

Exercise 1

Round robin queue: Ernie -> Bert -> Grover -> Elmo -> Oscar



Ernie arrives and runs

Bert arrives

Grover arrives

Ernie finishes; Bert runs

Elmo arrives

Oscar arrives

Bert's time expires; Grover runs

Grover's time expires; Elmo runs

Elmo finishes; Oscar runs

Oscar's time expires; Bert runs

Bert finishes; Grover runs

Grover finishes; Oscar starts

Oscar's time expires; continues running

Oscar finishes. All done!

Exercise 1

What is the finishing time for each?

Name	**Finishing Time**
Bert	39
Ernie	8
Oscar	58
Grover	46
Elmo	28

What is the **average waiting time?**

Waiting time = finish – running – arrival

Bert: 39 – 11 – 2 = 26
Ernie: 8 – 8 – 0 = 0
Oscar: 58 – 20 – 12 = 26
Grover: 46 – 15 – 7 = 24
Elmo: 28 – 4 – 10 = 14

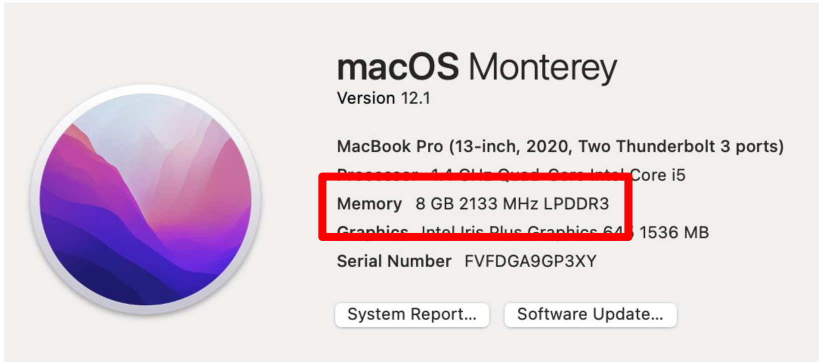
Average = (26 + 0 + 26 + 24 + 14) / 5 = 90 / 5 = 18

Review - Virtual Memory

Memory Overload!

Suppose I want to run these games simultaneously …

8 GB RAM!!! **4 GB!!!** **8 GB!!!**



macOS Monterey
Version 12.1

MacBook Pro (13-inch, 2020, Two Thunderbolt 3 ports)
Processor 1.4 GHz Quad-Core Intel Core i5
Memory 8 GB 2133 MHz LPDDR3
Graphics Intel Iris Plus Graphics 645 1536 MB
Serial Number FVFDGA9GP3XY

System Report... Software Update...

… on my humble laptop with 8GB random access memory (RAM)

(warning: don't do it!)

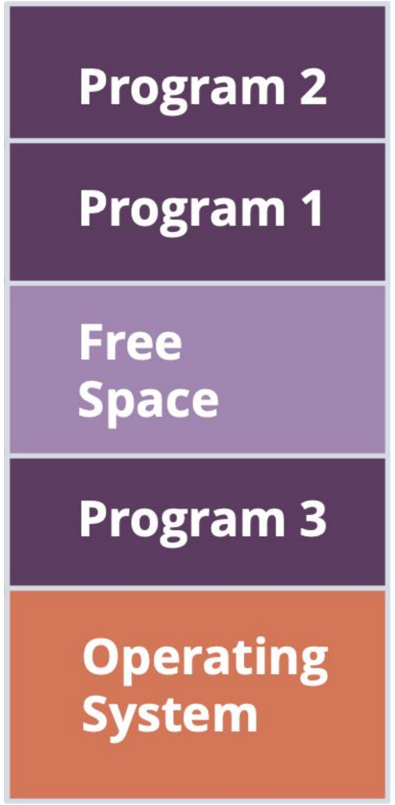
Why not physical memory?

First-pass solution:

Can we store program contents (code + data + heap + stack) directly in physical memory (RAM)?

We can divide up RAM spaces so that each program occupies a fixed partition of RAM, preventing accidental overwrites

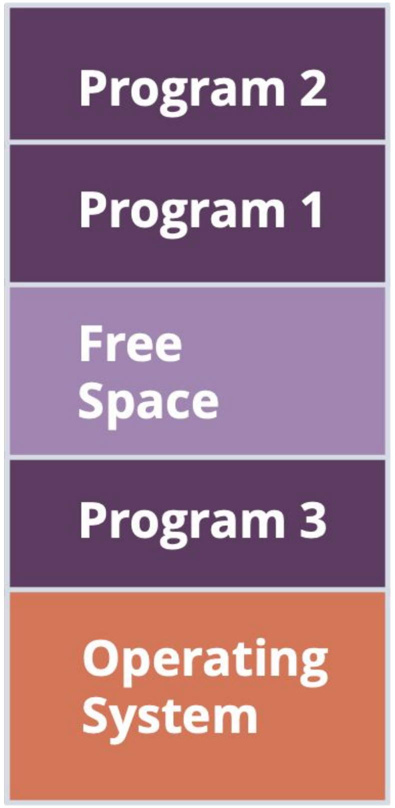
Problems with this approach?



Why not physical memory?

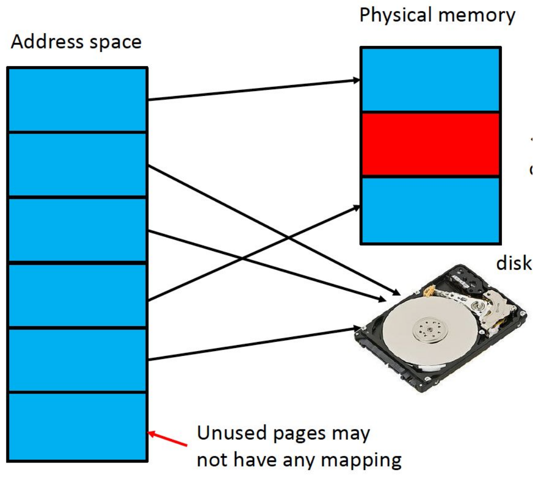
Problems with first-pass solution:

- We would run out of memory and crash!
  - We can't run any program larger than 8GB (or the partition we assign to the program)
- Potential inefficiencies in memory use
  - E.g. program 1 gets 7GB space, but only ends up using 2GB
- Compiler will need to know a program's partition beforehand
- Difficult to keep track of individual segments within a partition (e.g. enforcing read-only restriction)



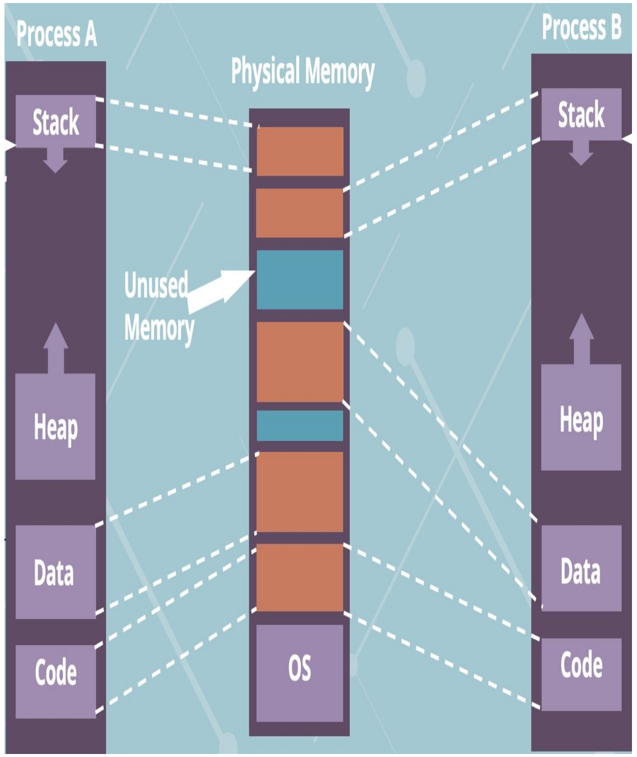
Introducing Virtual Memory

- **Virtual memory is not real memory. It is an address representation system that we use to make memory appear larger than it is**
- We tell each program that it can use all the addresses in the "address space"
  - In a 64-bit machine, address space is 2^64, with addresses from 0 to 2^64 - 1
- In behind the scenes, each (used) virtual memory address is mapped to a space in the RAM or the disk
  - The Memory Management Unit (MMU) takes care of this



Benefits of virtual memory

- We are no longer bounded by the size of the RAM!
  - Disk space is typically much larger (and comes at a cheaper price)
- We do not need to worry about process isolation
  - The MMU will take care of that
- More flexibility and efficiency in managing memory
  - The MMU can switch the actual storage location without the user/program noticing
  - Less likely to have “gaps” / fragmentation in memory



Why pages and page table?

- **OK, virtual memory is good. We get it. Why do we need this extra thing called pages and page table?**
  - Imagine a mapping of every individual virtual address to physical address - the lookup table will be as big as the address space (e.g. 2^64)!
  - Dividing virtual memory addresses into chunks make it easier to manage
- **A page is a unit of virtual memory (e.g. 4KB)**
  - Each page in virtual memory maps to a frame in physical storage (RAM + disk) of the same size
  - The mapping is recorded in the **page table**
  - Each process has one page table

This table will be very large and inefficient!

Virtual address	Valid	Physical address
x1A23	0	X2253 (RAM)
x399A	1	X5001 (Disk)
x7282	0	X3AB2 (RAM)

Keeping track of memory in page level is easier

Virtual page #	Valid	Physical Page Number
0	0	null
1	1	0
2	1	1
3	0	disk

VM Calculation Cheat Sheet

Address space = 2 ^ (# bits in the address, e.g. 64 for 64-bit machine)

Size of virtual memory = Address space * Addressability (# Bytes in each address)

Number of pages = Size of virtual memory / size of a page

Size of a page = size of a frame

Number of frames = Physical Memory (RAM) space / size of a frame

Number of bits to represent pages or frames = Log_2 (number of pages or frames)

Page number = first N bits of the virtual address, N = number of bits to represent pages or frames

1 KB = 2^10 B 1 MB = 2^20 B 1 GB = 2^30 B

Exercise 2
Pages &
Page tables

Exercise 2

Consider a system as follows:

- 32-bit address space
- 16-bit addressable
- 1GB of physical memory
- page sizes of 64kB

a) How many **pages** are there in virtual memory?

b) How many **frames** are there in physical memory?

c) How many **bits** are there in each address' page number?

d) Consider the virtual address xABCDEF01. What is its **page number** in **hexadecimal**?

Exercise 2 (a)

Consider a system as follows:

- 32-bit address space
- 16-bit addressable
- 1GB of physical memory
- page sizes of 64kB

How many **pages** are there in virtual memory?

2^17 or 128k

32-bit address space -> 2^32 addresses

16-bit (2-byte) addressable -> each address is 2 bytes -> 2^33B virtual memory

Each page is 64kB

2^33B (virtual memory size) / 64kB (page size) = 2^33 / 2^16 = 2^17 (or 128k)

Exercise 2 (b)

Consider a system as follows:

- 32-bit address space
- 16-bit addressable
- 1GB of physical memory
- page sizes of 64kB

How many **frames** are there in physical memory?

2^14 or 16k

Frame size = page size

Physical memory is 1GB, so 1GB (total size) / 64kB (frame size) = 2^30 / 2^16 = 2^14 (or 16k)

Exercise 2 (c)

Consider a system as follows:

- 32-bit address space
- 16-bit addressable
- 1GB of physical memory
- page sizes of 64kB

How many **bits** are there in each address' page number?

17

There are 2^17 pages

To represent N pages we need log_2 N bits

So we need log 2^17 = 17 bits

Exercise 2 (d)

Consider a system as follows:

- 32-bit address space
- 16-bit addressable
- 1GB of physical memory
- page sizes of 64kB

Consider the virtual address xABCDEF01. What is its page number in **hexadecimal**?

x1579B

In binary, the virtual address is 1010 1011 1100 1101 1110 1111 0000 0001

There are 2^17 pages so the first 17 bits are the page number.

So the page number is 1010 1011 1100 1101 1.

In hexadecimal, that's x1579B.

Page Replacement Algorithms

Page replacement algorithms

- The RAM can only hold up to a certain number of pages
  - If the RAM is full, we need to “evict” pages or move them to the disk
  - How should we decide which pages to evict?
- Goal: optimize (minimize) number of times we need to fetch something from the disk (**page faults**)
  - **FIFO** (first in first out): evict the page that first entered physical memory
  - **LRU** (least recently used): evict the page that has not been used for the longest time

Exercise 3
Page Eviction algorithms

Exercise 3

We have a byte-addressable system that has a 16-bit address space, 32kB of physical memory, and page sizes of 8kB. Assume the page table is initially empty, and then a process generates the following sequence of virtual addresses:

x3311
x1234
x1255
x3456
xA349
x7777
xB222
x6222

a) If virtual address `x5324` is requested next, which page will be evicted if using a First In First Out (FIFO) replacement algorithm?

b) Instead of using FIFO, which page will be evicted if using a Least Recently Used (LRU) replacement algorithm?

c) Rather than using FIFO or LRU, imagine that the system could look into the future and see that the next four virtual address requests (after x5324) would be x1A23, x399A, x7282, and x4A32. Knowing this information, which page should be evicted when the request for x5324 generates a page fault?

Exercise 3 (a)

We have a byte-addressable system that has a 16-bit address space, 32kB of physical memory, and page sizes of 8kB. Assume the page table is initially empty, and then a process generates the following sequence of virtual addresses:

x3311	**001**1
x1234	**000**1
x1255	**000**1
x3456	**001**1
xA349	**101**0
x7777	**011**1
xB222	**101**1
x6222	**011**0

If virtual address `x5324` is requested next, which page will be evicted if using a First In First Out (FIFO) replacement algorithm?

001

The page number is the first three bits, because there are eight virtual pages (16-bit address space = 64k addresses; each holds 1 byte so 64kB total; 8kB per page so 64kB/8kB = 8).

The four virtual addresses above have page numbers 000, 001, 101, and 011, and those take up the four frames (there are four frames because there's 32kB physical memory, and 32kB/8kB = 4).

When we get virtual address x5324, the page number is 010, and this causes an eviction.

The one that's oldest will be evicted, which in this case is the first one to be loaded, which is page number 001.

Exercise 3 (b)

We have a byte-addressable system that has a 16-bit address space, 32kB of physical memory, and page sizes of 8kB. Assume the page table is initially empty, and then a process generates the following sequence of virtual addresses:

x3311	**001**1
x1234	**000**1
x1255	**000**1
x3456	**001**1
xA349	**101**0
x7777	**011**1
xB222	**101**1
x6222	**011**0

Instead of using FIFO, which page will be evicted if using a Least Recently Used (LRU) replacement algorithm?

000

Using LRU, it's page number **000** that has been last used furthest in the past, so it gets evicted.

Exercise 3 (c)

We have a byte-addressable system that has a 16-bit address space, 32kB of physical memory, and page sizes of 8kB. Assume the page table is initially empty, and then a process generates the following sequence of virtual addresses:

x3311	0011
x1234	0001
x1255	0001
x3456	0011
xA349	1010
x7777	0111
xB222	1011
x6222	0110

Rather than using FIFO or LRU, imagine that the system could look into the future and see that the next four virtual address requests (after x5324) would be x1A23, x399A, x7282, and x4A32. Knowing this information, which page should be evicted when the request for x5324 generates a page fault?

101

As explained above, when x5324 is requested, the page numbers that are in the page table (i.e., that are mapped to frames) are 000, 001, 101, and 011, and the page number for address x5324 is 010 (the first three bits).

Given the requests indicated above, it would make sense to indicate page number **101** (which contains the addresses xA349 and xB222), since it is not used in any of the subsequent requests, whereas all the other page numbers are.