Section 10: Final Review

Autumn 2022

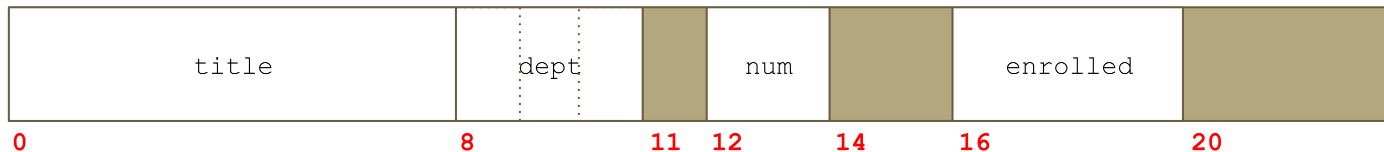
Final Exam Topics

- Arrays and Structs
    - alignment, fragmentation, buffer overflow
- Caching
    - locality, associativity, cache parameters and performance, AMAT
- Processes
    - fork, execv, exceptions, context switching, zombies
- Virtual Memory
    - paging, address translation, disk and swap space, protection and sharing
- Dynamic Memory Allocation
    - fragmentation, free lists (implicit, explicit, segregated), garbage collection, memory bugs
- C and Java

Arrays and Structs

Q1: Structs (A)

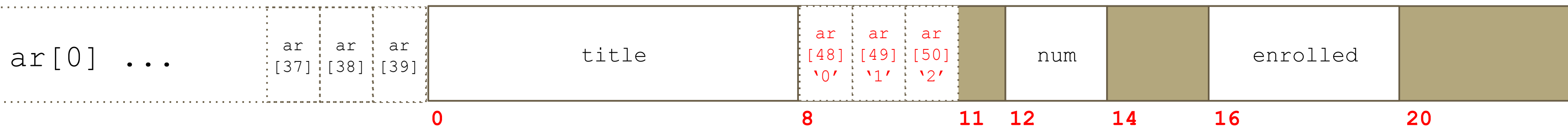
```
typedef struct { K:
 char* title; 8 // title (e.g. "HW SW INTERFACE")
 char dept[3]; 1 // dept (e.g. "CSE")
 short num; 2 // course number (e.g. 351)
 int enrolled; 4 // students enrolled
 } course; Kmax = 8
```



Q1: Structs (B)

Assume that an instance **course** c is allocated on the stack and an array **char** ar[] is allocated 40 bytes below c (*i.e.* &ar + 0x28 == (**char***)&c). Fill in the blanks below with the new ASCII characters stored in `c.dept` after the following loop is executed. Hint: recall that the values 0x30 to 0x39 correspond to the ASCII characters '0' to '9'.

```
for (int i = 0; i < 52; ++i) {
  ar[i] = i;
}
```



Caching

Q2: Caching (A)

We have 256 KiB of RAM and a 4-KiB L1 data cache that is 2-way set associative with 32-byte blocks and random replacement, write-back, and write allocate policies.

Physical Address:

256 KiB RAM → 2^{18} B of Physical Memory

Physical address is 18 bits long

Cache Parameters:

$C = 4$ KiB $= 2^{12}$ B

$K = 32 = 2^5$ B

$E = 2$

$S = 2^{12}/2^5/2 = 2^6$ sets

Tag bits	Index bits	Offset bits
7	6	5

Q2: Caching (B)

The code snippet below accesses two arrays of doubles. Assuming i is stored in a register and the cache starts *cold*, give the memory access pattern (read or write to which elements/addresses) and compute the **miss rate**.

```
#define SIZE 128
double src[SIZE]; // &src = 0x08000 (physical addr)
double dst[SIZE]; // &dst = 0x0E000 (physical addr)
for (int i = 0; i < SIZE; i += 1) {
  dst[i] = src[i];
  src[i] = i;
}
```

1: Read src[i]

2: Write dst[i]

3: Write src[i]

`&src = 0x08000 = 0b0010000|000000|00000`

TIO = 0x10 | 0x00 | 0x00|

`&dst = 0x0E000 = 0b0011100|000000|00000`

TIO = 0x1C | 0x00 | 0x00|

src[i] and dst[i] will always map to the same set & offset but will have different tags!

It's okay though: the cache is 2-way associative

Each block is 32B, so it can hold 4 doubles

Cache Set 0

1. RM src[0] 3. WH	4. RH src[1] 6. WH	7. RH src[2] 9. WH	10. RH src[3] 12. WH	Block 0
2. WM dst[0]	5. WH dst[1]	8. WH dst[2]	11. WH dst[3]	Block 1

Total per set: 10 hits, 2 misses

2/12 miss rate

Q2: Caching (C)

For each of the proposed (independent) changes, draw ↑ for "increased", — for "no change", or ↓ for "decreased" to indicate the effect on the **miss rate from Part B** for the code above:

Use float instead: ↓ **(more array elements can fit in a block, so there will be more hits)**

Double the cache size: **— (we didn't have any capacity problems)**

Half the associativity: ↑ **(a direct-mapped cache would cause conflict misses between src[i] and dst[i])**

No-write allocate: ↑ **(dst[i] would never be pulled into the cache, so writes into dst[i] will always be misses)**

Q2: Caching (D)

Assume it takes 160 ns to get a block of data from main memory. If our L1 data cache has a hit time of 5 ns and a miss rate of 5%, what is our average memory access time (AMAT)?

$$MP = 160ns$$

$$HT = 5ns$$

$$MR = 5\% = 0.05$$

$$AMAT = HT + MR \times MP$$

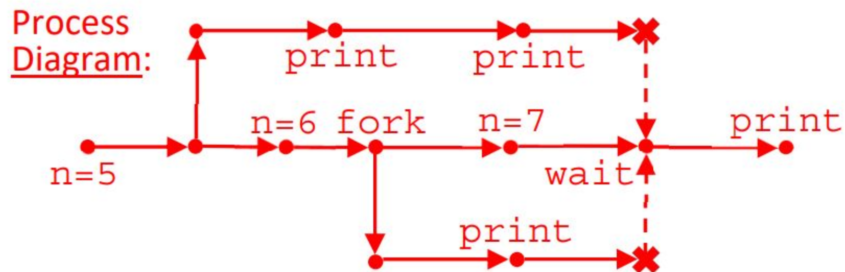
$$= 5\text{ ns} + 0.05 \times 160\text{ ns} = 5 + 8\text{ ns}$$

Processes

Q3: Processes (A)

```
void concurrent(void) {

 int n = 5;
 if (fork()) {
   n++;
   if (fork()) {
      n++;
      wait();
   }
   printf("%d, ", n);
   exit(0);
 } else {
   printf("%d, ", n);
 }
 printf("%d, ", n);
 exit(0);
}
```



7 Possible Outcomes:

1) 5, 5, 6, 7,
2) 5, 5, 7, 6,
3) 5, 6, 5, 7,
4) 5, 6, 7, 5,
5) 6, 5, 5, 7,
6) 6, 5, 7, 5,
7) 6, 7, 5, 5,

Q3: Processes (B)

For the following examples of exception causes, write "**S**" for synchronous or "**A**" for asynchronous from the perspective of the user process. (See Ed lesson RD20: Processes I for review)

Synchronous exceptions are caused by executing an instruction in the program
Asynchronous exceptions are caused by events external to the processor (i.e. interrupts)

System call _____**S**_____

Divide by zero ____**S**___

Segmentation fault ____**S**___

Key pressed _____**A**____

Everything but a key press is caused by an assembly instruction *within* your program.

Q3: Processes (C)

Fill in the following blanks with “**A**” for always, “**S**” for sometimes, and “**N**” for never if the following would be different when **context switching** to a *different* process? (See Ed lesson RD20: Processes I for review)

Process ID: **A (each processes always has their own unique process IDs (PIDs))**

Program: **S (different processes might be instances of the same program)**

PTBR: **A (each process always has their own unique page table)**

Condition Codes: **S (different processes might have the same condition codes coincidentally)**

Q3: Processes (D)

Is the following statement True or False? Provide a brief justification:

“a single process can execute multiple programs simultaneously”

False.

One process is dedicated to running one program at a time. The program defines the instructions, initial memory state, etc. of the process, so two programs can’t exist within the same process at once.

Virtual Memory

Q4: Virtual Memory (A)

Our system has the following setup:

- 15-bit virtual addresses and 2 KiB of RAM with 256-byte pages
- A 4-entry fully-associative TLB with LRU replacement
- A PTE contains bits for valid (V), dirty (D), read (R), write (W), and execute (X)

Compute the following values:

Page Offset Width: **8 bits (a page is 256B = 2^8 B)**

# of TLB Sets: **1 set (the TLB is fully-associative, so everything goes into the same set)**

# of Virtual Pages: **2^7 pages (15-bit VA → 2^{15} B in virtual address space → 2^{15} virtual bytes/2^8 B in a page = 2^7 virtual pages**

Minimum Width of PTBR: **11 bits (2 KiB RAM → 2^{11} B in physical address space → all physical address must be 11 bits wide)**

Q4: Virtual Memory (B)

Assuming that the TLB is in the state shown (permission bits: 1 = allowed, 0 = disallowed), give example addresses that will fulfill the following scenarios:

TLBT	PPN	Valid	D	R	W	X
0x20	0xc	1	0	1	0	0
0x7f	0xa	1	0	1	1	0
0x7e	0xf	1	0	1	1	0
0x04	0xe	1	0	1	1	1

15-bit virtual addresses

VPN | Page Offset → 7 bits | 8 bits

TLB Tag | TLB Index | Page Offset → 7 bits | 0 bits | 8 bits

A value in `%rip` that causes a TLB Hit and no exception:

%rip is instruction pointer, need permission to execute code.

Want TLB entry with V=1, X=1 → Want VPN/TLB Tag of 0x04

So any address between 0x0400-0x04FF will work

A *write* address that causes a TLB Hit and segmentation fault:

Want TLB entry with V=1, W=0 → Want VPN/TLB Tag of 0x20
So any address between 0x2000-0x20FF will work

Dynamic Memory Allocation

Q5: Memory Allocation (A)

```
1 #include <stdlib.h>
2 float pi = 3.14;
3
4 int main(int argc, char *argv[]) {
5   int year = 2019;
6   int* happy = malloc(sizeof(int*));
7   happy++;
8   free(happy);
9   return 0;
10}
```

Consider the C code shown above. Assume that the malloc call succeeds and happy and year are stored in memory (not in a register).

Fill in the following blanks with “<” or “>” or “UNKNOWN” to compare the *values* returned by the following expressions just before return 0.

&year **>** &main

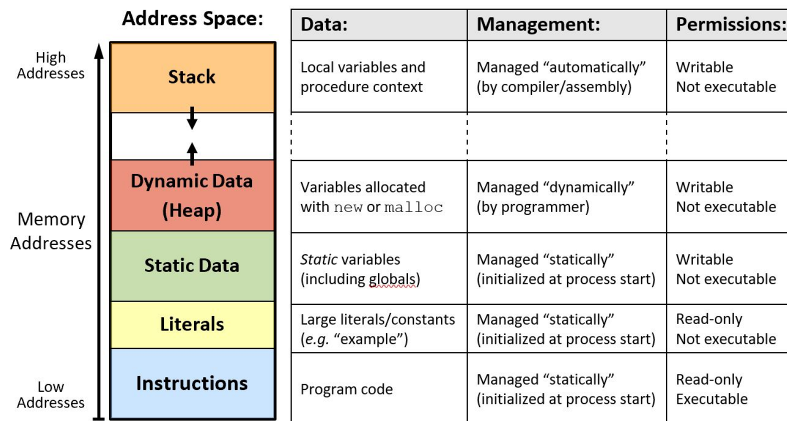
&year would return an address in the stack segment
&main would return an address in the instructions segment

happy **<** &happy

happy would return an address in the heap segment
&happy would return an address in the stack segment

&pi **<** happy

&pi would return an address in the static data segment
happy would return an address in the heap segment



Address Space:	Data:	Management:	Permissions:
Stack	Local variables and procedure context	Managed “automatically” (by compiler/assembly)	Writable Not executable
Dynamic Data (Heap)	Variables allocated with new or malloc	Managed “dynamically” (by programmer)	Writable Not executable
Static Data	*Static* variables (including globals)	Managed “statically” (initialized at process start)	Writable Not executable
Literals	Large literals/constants (*e.g.* “example”)	Managed “statically” (initialized at process start)	Read-only Not executable
Instructions	Program code	Managed “statically” (initialized at process start)	Read-only Executable

Q5: Memory Allocation (B)

The code above has two memory-related errors. Use the line numbers in the code to describe what the errors are and where they occur.

```
1 #include <stdlib.h>
2 float pi = 3.14;
3
4 int main(int argc, char *argv[]) {
5   int year = 2019;
6   int* happy = malloc(sizeof(int*));
7   happy++;
8   free(happy);
9   return 0;
10}
```

Error #1: **On line 6 we are requesting more memory than we need. We should be requesting size of int (4 bytes), not size of int* (8 bytes). Alternatively we could have meant to declare happy to be of type int** (a pointer to a pointer to an int) so that we would have needed 8 bytes to hold a pointer to an int.**

Error #2: **On line 8 we are calling free on a pointer that was not the one returned to us by malloc. In line 7 we are incrementing happy (a pointer to an int that was returned to us by malloc).**

Q5: Memory Allocation (C)

Give one advantage that next fit placement policy has over a first fit placement policy in an implicit free list implementation.

Next fit searches the list starting where the previous search finished. **This should often be faster than first fit because it avoids re-scanning unhelpful blocks.**

First fit always starts searching at the beginning of the list. In an implicit free list this is particularly bad because the “free” list actually contains all allocated blocks as well as free blocks. So starting from the beginning of the list is likely to traverse many allocated blocks each time.

Q5: Memory Allocation (D)

List two reasons why it would be hard to write a garbage collector for the C programming language.

Reason #1: **Pointers in C can point to a location other than the beginning of a block of memory on the heap.**

Reason #2: **In C you can “hide” pointers e.g. by casting them to longs.**

Java and C

Q6: Java vs. C

This is an open-ended question, just make sure to avoid repeating the same point but worded differently or listing the same point but worded in opposite ways for both questions.

(A) Describe two distinct ways or things that you think C does better than Java.

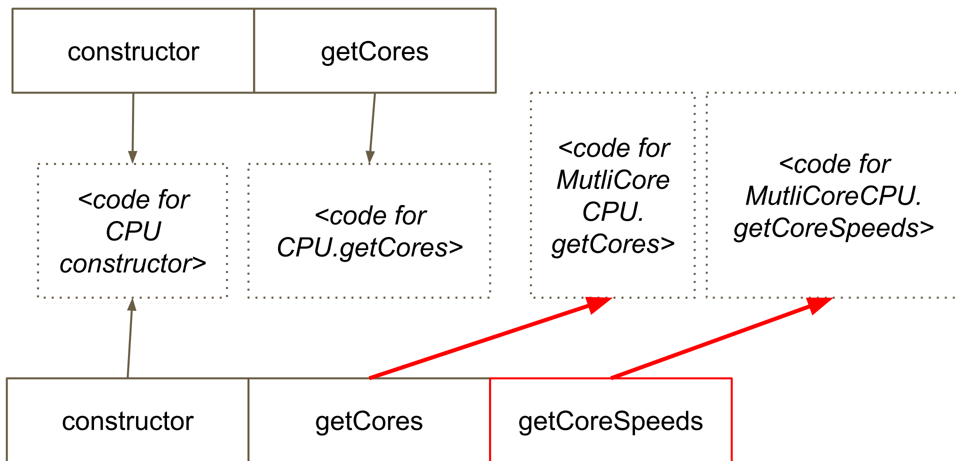
Possible Answer Topics: Pointer Manipulation, Memory Usage/Management, Compiles Faster/ Runs Faster, Closer to Machine Code (Lower Level), Programmer has more direct control

(B) Describe two distinct ways or things that Java does better than C.

Possible Answer Topics: Safety (e.g. type casting, bounds checking, exception handling, simpler code), Higher Level (less responsibility for the programmer), Data Structures (also OOP, Classes), More Portable, References vs. Pointers

Q7: Java Classes (A)

The vtable for CPU is shown below. Annotate the diagram with the *changes* that we would need to make for the vtable of MultiCoreCPU.




```java
class CPU {
    float clockSpeed;
    int cacheSize;
    int cacheAssoc;

    int getCores() {
        return 1;
    }
}

class MultiCoreCPU extends CPU {
    int numberOfCores;
    float[] coreSpeeds = new float [16];

    int getCores() {
        return numberOfCores;
    }

    float[] getCoreSpeeds() {
        return coreSpeeds;
    }
}
```

Q7: Java Classes (B)

You may assume that the alignment for this JVM implementation is the same as C on x86-64, and that fields are stored in memory in the order that they are declared. (see Lecture 27: Java and C)

How much space does an instance of CPU take up?

header	8B
vptr	8B
float clockSpeed	4B
int cacheSize	4B
int cacheAssoc	4B
padding	4B

32B

```
class CPU {
    float clockSpeed;
    int cacheSize;
    int cacheAssoc;

    int getCores() {
        return 1;
    }
}

class MultiCoreCPU extends CPU {
    int numberOfCores;
    float[] coreSpeeds = new float [16];

    int getCores() {
        return numberOfCores;
    }

    float[] getCoreSpeeds() {
        return coreSpeeds;
    }
}
```

Q7: Java Classes (C)

How much space does an instance of MultiCoreCPU take up?

header	8B
vptr	8B
float clockSpeed	4B
int cacheSize	4B
int cacheAssoc	4B
int numberOfCores	4B
float[] coreSpeeds	8B

These 8 bytes for coreSpeeds are a pointer to the array object

40B

```
class CPU {
    float clockSpeed;
    int cacheSize;
    int cacheAssoc;

    int getCores() {
        return 1;
    }
}

class MultiCoreCPU extends CPU {
    int numberOfCores;
    float[] coreSpeeds = new float [16];

    int getCores() {
        return numberOfCores;
    }

    float[] getCoreSpeeds() {
        return coreSpeeds;
    }
}
```

Q7: Java Classes (D)

Give an example of something that is allowed in C, but *not* in Java, because it would prevent the garbage collector from working properly.

Possible examples:

- **pointers to middle of structs/objects**
- **casting pointers to other types**
- **being able to read a pointer and modify the address it points to**
- **...**