# CS 354 - Machine Organization & Programming
# Tuesday Feb 14 and Thursday Feb 16, 2023

**Midterm Exam - Thursday, February 23th, 7:30 - 9:30 pm**

- **Room: Students will be assigned a room based and sent email with that room**
- **UW ID required**
- **#2 pencils required**
- **closed book, no notes, no electronic devices (e.g., calculators, phones, watches)**
- **see "Midterm Exam 1" on course site Assignments for topics**
- see sample exam cover page (last page of this outline)

**Project p2A:** Due on or before Friday, Feb 17th

**Project p2B:** Due on or before Friday, Feb 24th

**Homework hw1:** Due on Monday Feb 13th (solution available Wed morning)

**Homework hw2:** Due on Monday Feb 20th (solution available Wed morning)

**Last Week**

| | |
|---|---|
| Command-line Arguments<br>Recall 2D Arrays<br>2D Arrays on the Heap<br>2D Arrays on the Stack<br>2D Arrays: Stack vs. Heap | Array Caveats<br>Meet Structures<br>Nesting in Structs and Arrays of Structs<br>Passing Structures |

**This Week**

| | |
|---|---|
| Pointers to Structures (from L6)<br>Standard & String I/O in `stdio.h`<br>File I/O in `stdio.h`<br>Copying Text Files<br><br>Three Faces of Memory | Virtual Address Space<br>C's Abstract Memory Model<br><br>Meet Globals and Static Locals<br>Where Do I Live?<br>Linux: Processes and Address Spaces<br>Exam Sample Cover Page |
| **Next Week**: The Heap & Dynamic Memory Allocators (p3)<br>Read: B&O 9.1, 9.2, 9.9.1-9.9.6<br>9.1 Physical and Virtual Addressing<br>9.2 Address Spaces<br>9.9 Dynamic Memory Allocation<br>9.9.1-9.9.6 | |

# Standard and String I/O in `stdio.h`

## Standard I/O

### Standard Input

```
getchar //reads 1 char
gets    //reads 1 string ending with a newline char, BUFFER MIGHT OVERFLOW
```

don't use for string

**`int scanf(const char *format_string, &v1, &v2, ...)`**

reads formatted input from the console keyboard
returns number of inputs stored, or EOF if error/end-of-file occurs before any inputs

*format string* contains format chars and chars to skip

*format specifiers* %d, %f, %c, %i, %p, %s

octal, decimal, hex

*whitespace* input separator (space, tab, newline)
leading whitespace is skipped

### Standard Output

```
putchar //writes 1 char
puts    //writes 1 string
```

**`int printf(const char *format_string, v1, v2, ...)`**

writes formatted output to the console terminal window
returns number of characters written, or a negative if error

*format string* format specifiers and chars to print

use \n to flush output buffer

### Standard Error

**`void perror(const char *str)`**

writes formatted error output to the console terminal window

## String I/O

input

```
int sscanf(const char *str, const char *format_string, &v1, &v2, ...)
```

reads formatted input from the specified `str`
returns number of characters read, or a negative if error

```
int sprintf(char *str, const char *format_string, v1, v2, ...)
```

writes formatted output to the specified `str`
returns number of characters written, or a negative if error

# File I/O in `stdio.h`

**Standard I/O Redirection** IN LINUX //file extensions (.txt, .exe) rarely used in linux

a.out < input_file > output_file //overwrites output_file
>> output_file //appends output_file

**File I/O** in C p2A, p2B

File Input

`fgetc/~~getc~~, ungetc` //reads 1 char at a time from a file

p2A `fgets` //reads 1 string terminate with a newline char or EOF

**`int fscanf(FILE *stream, const char *format_string, &v1, &v2, ...)`**

reads formatted input from the specified `stream`
returns number of inputs stored, or EOF if error/end-of-file occurs before any inputs

File Output

`fputc/~~putc~~` //writes 1 char at a time

`fputs` //writes 1 string

**`int fprintf(FILE *stream, const char *format_string, v1, v2, ...)`**

writes formatted output to the specified `stream`
returns number of characters written, or a negative if error

**Predefined File Pointers**

`stdin` is console keyboard
`stdout` is console terminal window
`stderr` is console terminal window, second stream for errors

printf("hello\n"); = fprintf(stdout, "hello\n");

**Opening and Closing**

"r" read
"w" write

`FILE *fopen(const char *filename, const char *mode)`

opens the specified `filename` in the specified `mode`
returns file pointer to the opened file's descriptor, or NULL if there's an access problem

`int fclose(FILE *stream)`

flushes the output buffer and then closes the specified `stream`
returns 0, or EOF if error

# Copying Text Files

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

   if (argc != 3) {
      fprintf(stderr, "Usage: copy inputfile outputfile\n");
      exit(1);
   }


   FILE *ifp = fopen(argv[1], "r");
   if (ifp == NULL) {
      fprintf(stderr, "Can't open input file %s!\n", argv[1]);
      exit(1);
   }


   FILE *ofp = fopen(argv[2], "w");
   if (ofp == NULL) {
      fprintf(stderr, "Can't open output file %s!\n", argv[2]);
      fclose(ifp);
      exit(1);
   }


   const int bufsize = 257; //WARNING: assumes lines <= 256 chars
   char buffer[bufsize];//STACK
   while ( fgets(buffer, bufsize, ifp)  != NULL) {
      fputs(buffer, ofp);
   }
   fclose(ifp);
   fclose(ofp);




   return 0;
}
```

# Three Faces of Memory

❋ *Abstraction:* manage complexity by focusing on relevant details

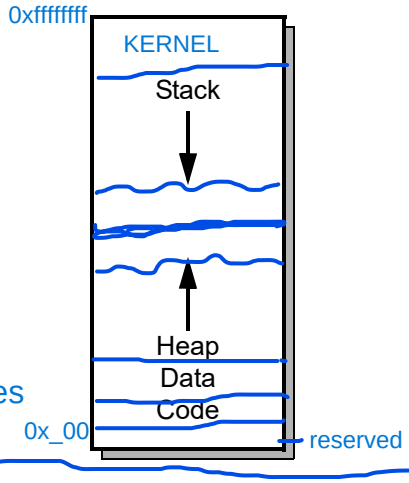
**Process View = Virtual Memory**

Goal: provide a simple view

CS354

*virtual address space (VAS)*: an illusion by the OS that each process has its own contiguous memory space

*virtual address*: simulated address that a process generates



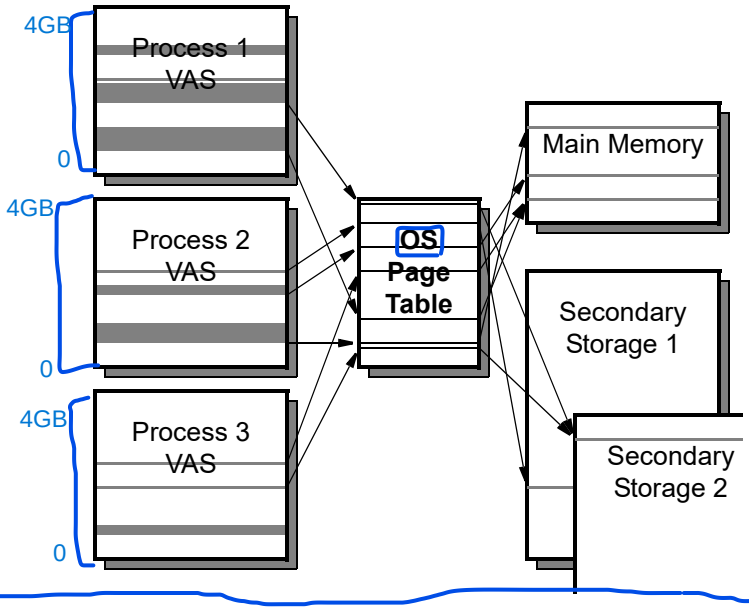

**System View = Illusionist (CS 537)**

Goal: make memory shareable & secure

*pages*: 4K (kilobytes) mem block managed by OS

*page table*: OS data struct to track page allocation

VAS -> PAS for each process (virtual to physical address)



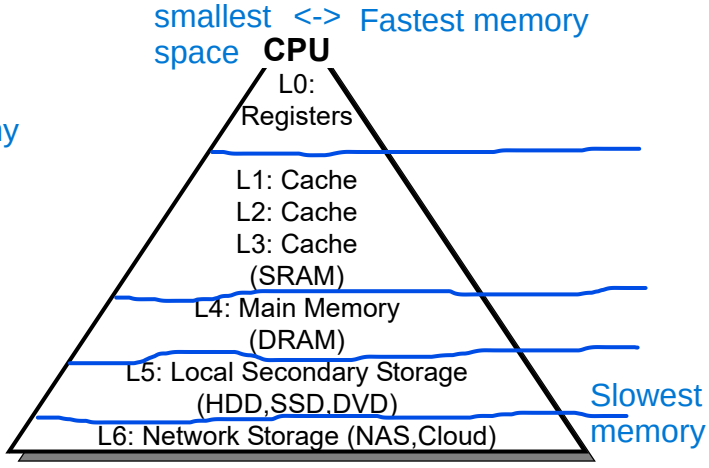

**Hardware View = Physical Memory**

Goal: Keep CPU busy

*physical address space (PAS)*: Multi-level hierarchy ensures frequently accessed data is close to CPU

CS354 Later

*physical address*: actual address used to access machine's memory



# Virtual Address Space (IA-32/Linux)

**32-bit Processor = 32-bit Addresses => $2^{32}$ = 4,294,967,296 = 4GB Address Space**

11111111111111111111111111111111 = 0xFFFFFFFF

*address space*: a range of valid addresses for a process

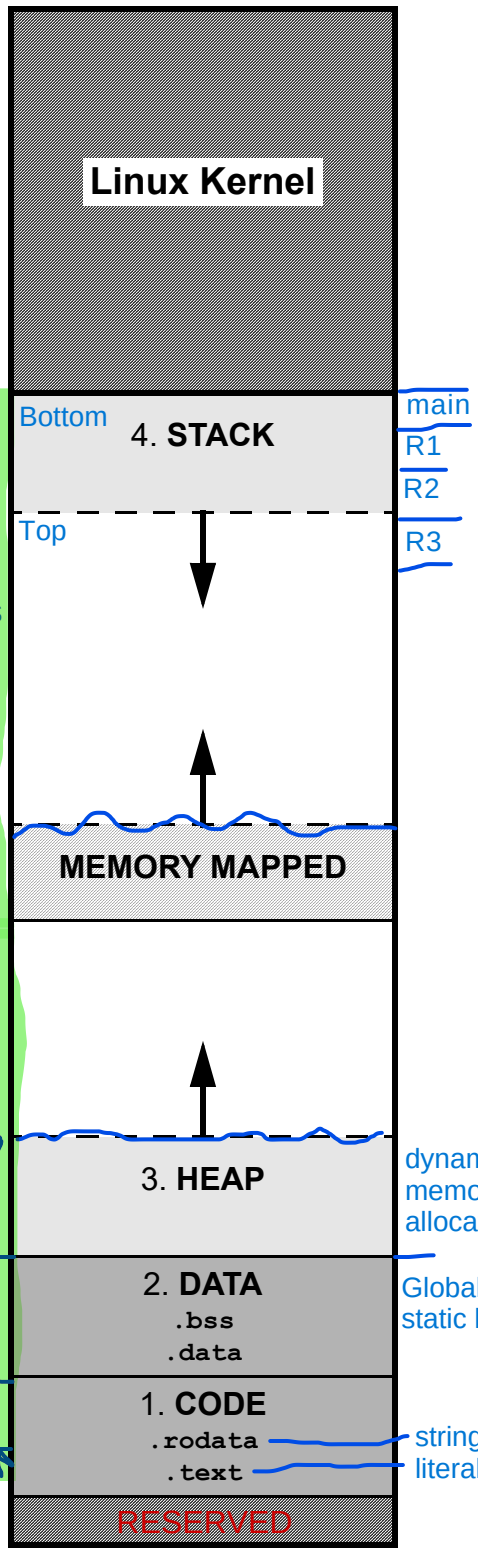
*process*: a running process

11000000000000000000000000000000 = 0xC0000000

*kernel*: memory resident part of OS

*user process*: process that is not in KERNEL. "our" processes

- *Every user process* has this simple view of memory




for DLL's, file I/O, large heap allocation

*****REMEMBER CODE, DATA, HEAP, AND STACK

brk shows amount of mem allocated to heap

dynamic memory allocation

Global and static locals

string literals

00001000000010010000000000000000 = 0x08048000

00000000000000000000000000000000 = 0x00000000

trying to access reserved memory (0x0) causes seg fault

# C's Abstract Memory Model

1. **CODE Segment**

   Contains: the program

   **.text** section instructions

   **.rodata** section string literals

   ro = read only

   Lifetime: entire program's execution

   Initialization: from EOF by LOADER

   Access: READ-ONLY

2. **DATA Segment**

   Contains: global variables and static local

   Lifetime: entire program's execution

   Initialization: from EOF by LOADER

   **.data** section from code, for variables initialized to non-zero values

   **.bss** section from code, for variables that are initialized to zero or uninitialized

   Block Started Symbol

   Access: read/write

3. **HEAP** (AKA Free Store)

   Contains: memory allocated and freed by the programmer during execution

   Lifetime: managed by the programmer, malloc/calloc/realloc until free

   Initialization: none by default

   Access: read/write

4. **STACK** (AKA Auto Store)

   Contains: memory in stack frames, auto allocated and freed by function call/return

   *stack frame* (AKA activation record) S.F., A.R., parameters, local variables, temp variables, more...

   Lifetime: from declaration to end of scope, often a "closing brace" }

   Initialization: none by default

   Access: read/write

# Meet Globals and Static Locals

**What?**

A *global variable* is
- declared outside of a function
- accessible to all functions in a source file
- allocated in DATA segment

A *static local variable* is
- declared in a function with static modifier
- accessible only within the function
- allocated in DATA segment

**Why?**

for storage that exists during entire program

❋ *In general, global variables* should not be used!
*Instead use* local variables that are passed to the callee function

**How?**

```
#include <stdio.h>
int g = 11; //global

void f1(int p) { //parameter (very similar to local variable)
   static int x = 22; //static local
   x = x + p * g;
   printf("%d\n", x);
}

int main(void) {
   f1(g);
   g = 2;
   int g = 1; //local variable that shadows global variable
   f1(g);
   return 0;
}
```

*shadowing*: when a local variable blocks access to a global variable of same name

❋ *Avoid shadowing; don't use the same identifier* for local and global variables

## Where do I live?

which segment of VAS contains memory for each variable

→ Identify the segment (and section) for each memory allocation in the code below.

```
#include <stdio.h>
#include <stdlib.h>

int gus = 14;          DATA .data
int guy;               DATA .bss

int madison(int pam) {                STACK

    static int max = 0; DATA .bss
    int meg[] = {22,44,88};           STACK
    int *mel = &pam;
    max = gus--;
    return max + meg[1] + *mel;
}

int *austin(int *pat){                STACK

    static int amy = 33; DATA .data
    int *ari = malloc(sizeof(int)*44);   STACK        HEAP
    gus--;
    *ari = *pat;
    return ari;
}

int main(int argc, char *argv[]) {    STACK

   int vic[] = {33,66,99};                SAA
   int *wes = malloc(sizeof(int));        STACK      HEAP
   *wes = 55;
   guy = 66;
   free(wes);
   wes = vic;
   wes[1] = madison(guy);
   wes = austin(&gus);
   free(wes);
   printf("Where do I live?");
   return 0;
}
```

❋ *Arrays, structs, and variables* can live in DATA, HEAP, STACK

Pointer var can store any address, SEG FAULT if dereference outside of your memory

# Linux: Processes and Address Spaces

## Process and Job Control

- Linux is a multitasking operating system where you can run multiple processes concurrently

`ps` snapshot of user processes

`jobs` only user process

`&` put process in background
ctrl+z suspend running process

`bg` put a suspended process in the background
`fg` bring suspended process to foreground

ctrl+c stop a running process

## Program Size

size <executable or object_file>  a.out, prog.

```
$gcc -m32 myProg.c
$size a.out
   text    data     bss     dec     hex filename
   1029     276       4    1309     51d a.out
```

CODE  DATA  TOTAL

## Virtual Address Space Maps

- Linux enables you to see the virtual address space of each process

```
$pmap <pid_of_process>

$cat /proc/<pid_of_process>/maps

$cat /proc/self/maps
```

cat /proc:

| [ ] Lec 001 9:30am TR<br>[ ] Lec 002 1:00pm TR<br>Lecture | Print Netid | Print Name (first last) |
|---|---|---|

Computer Sciences 354

# Midterm Exam 1 Secondary

Thursday, October 6th, 2022

60 points (15% of final grade)

Instructors: Debra Deppeler

1. MARK an X in box by your lecture number above.
2. PRINT your NETID (UW login name not your photo id number) in box above.
3. PRINT your first and last name in box above.
4. FILL-IN all fields and their bubbles on the scantron form (use # 2 pencil).
   (a) LAST NAME - fill in your last (family) name starting at leftmost column.
   (b) FIRST NAME - fill in first five letters of your first (given) name.
   (c) IDENTIFICATION NUMBER is your UW Student ID number.
   (d) Under ABC of SPECIAL CODES, write your lecture number as a three digit value 001 or 002.
   (e) Under F of SPECIAL CODES, write the number 2 for Secondary and fill in the number (2) bubble.
5. DOUBLE-CHECK THAT YOU HAVE FILLED IN ALL ID FIELDS and that you have FILLED IN ALL CORRESPONDING BUBBLES ON SCANTRON.
6. Taking this exam indicates that you agree: to not write answers in large letters and to keep your answers covered; to not view or use another's work or any unauthorized devices in any way; to not make any type of copy of any portion of this exam; and that you understand that being caught doing any of these actions, or other actions that permit any student to submit work that is not wholly their own will result in automatic failure of the exam and possible failure of the course. Penalties are reported to the Deans Office for all involved.

| Parts | Number of Questions | Question Format | Possible Points |
|---|---|---|---|
| I | 10 | Simple Choice | 20 |
| II | 10 | Multiple Choice | 30 |
| III | 2 | Written | 10 |
| | 22 | Total | 60 |

Assumptions unless instructions explicitly state otherwise:
addresses and integers are 4 bytes.
code questions are about C and IA-32/x86 assembly code on our Linux platform.

## Reference: Powers of 2

$2^5 = 32, 2^6 = 64, 2^7 = 128, 2^8 = 256, 2^9 = 512, 2^{10} = 1024$
$2^{10} = K, 2^{20} = M, 2^{30} = G$
$2^A * 2^B = 2^{A+B}, 2^A / 2^B = 2^{A-B}$

Turn over and put away all electronic devices and wait for the proctor to signal the start of the exam.

