CS 354 - Machine Organization & Programming
Tuesday Feb 13th and Thursday Feb 15th, 2024

Midterm Exam - Thursday, February 22nd, 7:30 - 9:30 pm

- **Room: Students will be assigned a room 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**

PM BYOL: Start p2A and p2B if you have not yet started either

Activity A04: due on or before this week Saturday

Homework hw1: Due on or before this week Monday (solution available Wed morning)

Homework hw2: Due on or before next week Monday

Project p2A: Due on or before this week Friday, Feb 16

Project p2B: Due on or before next week Friday, Feb 23

Week 4 Learning Objectives (at a minimum be able to)

- use **<stdio.h>** functions: **printf**, **scanf**, **fopen**, **fclose**, **fgets**, **fputs**
- use predefined file pointers: **stdin** and **stdout**
- use format specifiers: **%c %f %i %d %s %p %x**
- use Linux I/O redirection at the command line: < input_file > output_file >> append_file
- describe C's abstract memory model: **Process View = Virtual Memory**
- diagram C's abstract memory model: **CODE**, **DATA**, **HEAP**, **STACK**
- meet IA-32 memory hierarchy: **Hardware View = Physical Memory**
- understand difference and use of **global** vs **static local** variables

This Week

Pointers to Structures (from last week) Standard & String I/O in `stdio.h` File I/O in `stdio.h` Copying Text Files Three Faces of Memory Virtual Address Space	C's Abstract Memory Model Meet Globals and Static Locals Where Do I Live? Linux: Processes and Address Spaces Exam Sample Cover Page

Next Week: The Heap & Dynamic Memory Allocators (p3)
Read: B&O 9.1, 9.2, 9.9.1-9.9.6
- 9.1 Physical and Virtual Addressing
- 9.2 Address Spaces
- 9.9 Dynamic Memory Allocation
- 9.9.1-9.9.6

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

Standard I/O

string: sequence of char w null terminating char

Standard Input

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

gets char from std input by default, reads 1 char and leaves rest on stream

string: sequence of char w null terminating char

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

reads formatted input from the console keyboard

read into these

returns number of inputs stored, or EOF if error/end-of-file occurs before any inputs

End of File

format string contains format specifiers and characters to skip

format specifiers %d, %f float, %p, %s string, %i int

03 octal
0x1b hex
33 dec

must match corresponding dest variable

%d: formats the output as a signed decimal integer.

whitespace input separator (space, tab, newline)

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 display

use \n to flush buffer

Standard Error

alternate output location, terminal window by default

`void perror(const char *str)`

writes formatted error output to the console terminal window

String I/O

can read from and write to string instead of keyboard terminal

scan from string, into the variables

```
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

a.out (executable, presumably requires input)

a.out < input_file

a.out (executable, presumably has output)

a.out > output_file overwrites to output_file

a.out >> output_file appends to output_file

File I/O

File Input

```
fgetc/~~getc~~, ungetc //reads 1 char at a time
fgets            //reads 1 string terminate with a newline char or EOF
```

expects data file to be opened

```
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

same as about but to file instead of terminal keyboard now

```
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

Opening and Closing

"r" read or "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

check for errors

```
int fclose(FILE *stream)
```

flushes the output buffer and then closes the specified `stream`

returns 0, or EOF if error

error if no permission to write/read etc

Copying Text Files

name of file is copy

```
#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];
                          while ( fgets(buffer, bufsize, ifp) != NULL ) {
                            fputs(buffer, ofp);
                          }

                          if (fclose(ifp) == NULL) {
                            print(“error ifp close\n”);
                            exit(1);
                          }

                          if (fclose(ofp) == NULL)
                            print(“error ofp close\n”);
   return 0;              exit(1);
}                         }
```

copy src_filename dst_filename

Three Faces of Memory

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

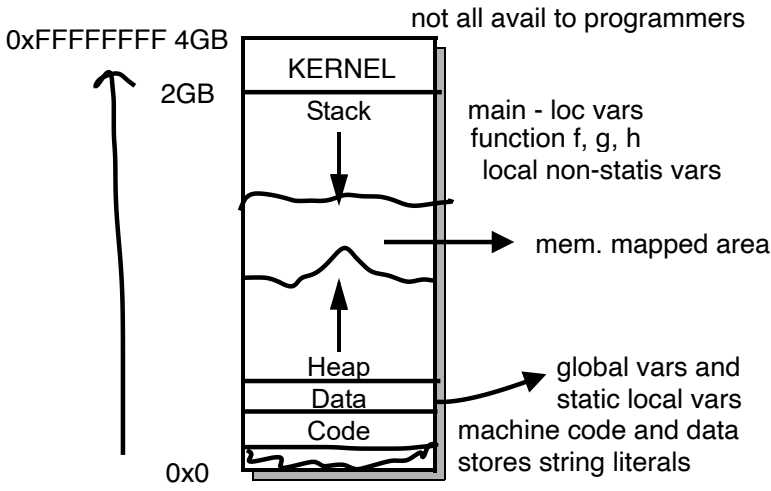
Process View = Virtual Memory

Goal: provide simple view

virtual address space (VAS):

illusion by OS that each process has its own contiguous memory block

virtual address: simulated addr a process generates



System View = Illusionist (CS 537)

Goal: make memory sharable and secure

pages: 4KB units of storage

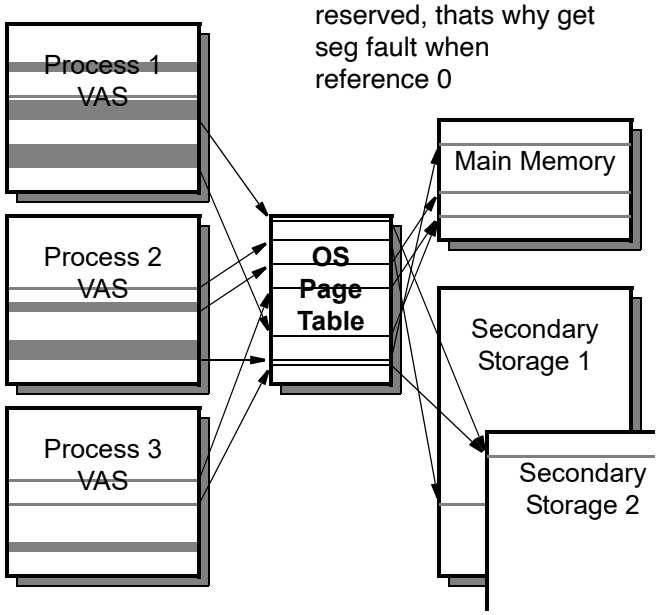
fixed-size determined by OS, contiguous

page table:

OS data structure

maps Virtual pages to Physical pages

keeps process from interfering with other processes



Hardware View = Physical Memory

Goal: keep CPU busy

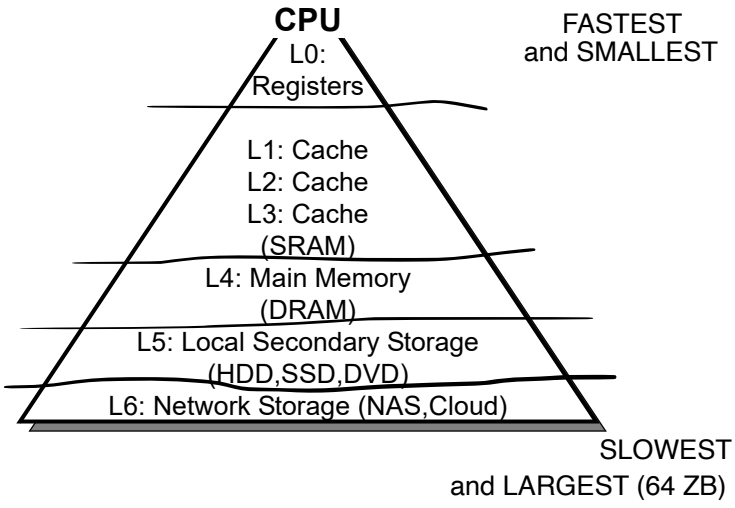
physical address space (PAS):

multi-level hierarchy

frequently accessed data in memory closest to CPU

physical address:

actual address used to access memory



Virtual Address Space (IA-32/Linux)

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

4GB = 11111111111111111111111111111111 = 0xFFFFFFFF

address space:
range of valid memory addresses for processes

process: a running program

Windows IA-32 has 2GB of OS, user gets 2GB

11000000000000000000000000000000 = 0xC0000000

kernel: memory resident portion of the OS

user process: process that is not kernel

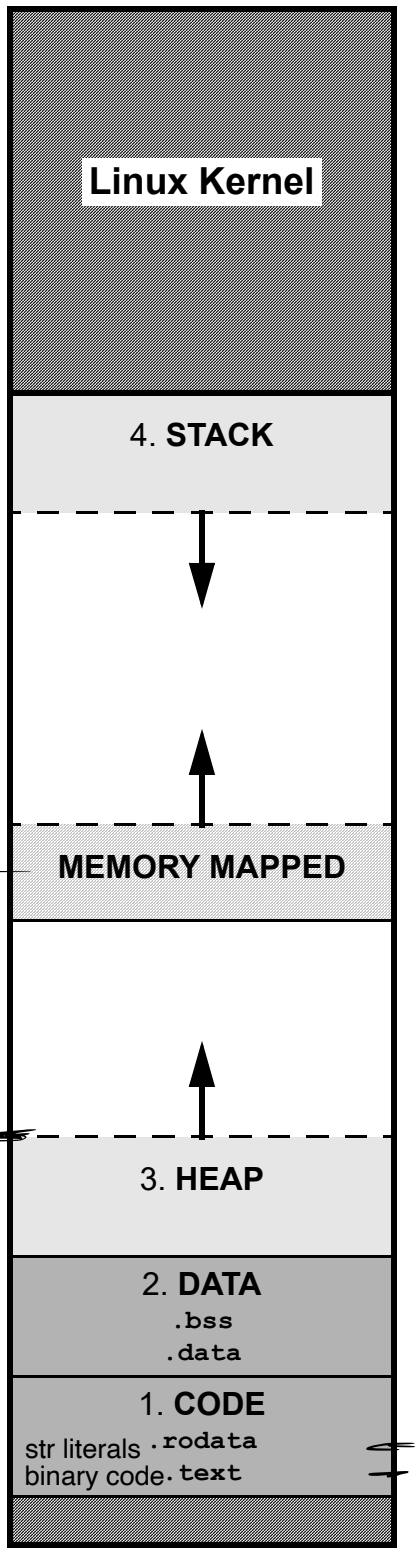
- *Every user process* has this simple view of memory
  makes coding easier

Dynamic Linked Libraries (DLL), File I/O, large memory requests

magic number - where code segment starts

00001000000001001000000000000000 = 0x08048000

00000000000000000000000000000000 = 0x00000000



binary addresses | hex addresses | management unit

C's Abstract Memory Model

1. CODE Segment

Contains: program code — binary machine code

.text section machine code

.rodata section str literals

Lifetime: entire program's execution

LOADER put code in memory before start executing

Initialization: by LOADER from executable object file (before execution begins)

Access: READ ONLY access

2. DATA Segment

Contains: global variables and static locals

Lifetime: entire program's execution

Initialization: by LOADER from executable object file before execution

.data section variables that are initialized to non-zero values

.bss section variables that are not initialized or initialized to zero

Block started by 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 — free

can malloc in one func, can free in another func

Initialization: none by default

Access: read/write

4. STACK (AKA Auto Store)

Contains: memory in stack frames — automatically allocated and freed by code generated by compiler

stack frame (AKA activation record) non-static local , parameters, temp local vars (i,j,k)

int i = 11;
non-static

Lifetime: from when its declared till end of scope

Initialization: none by default

Access: read/write

Meet Globals and Static Locals

What?

A *global variable* is

- declared outside of any function
- accessible to all functions in the source file
- allocated in the DATA segment

A *static local variable* is

- declared inside a function with a static modifier
- accessible only within the function
- allocated in the DATA segment

Why?

for storage that exists during entire runtime

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

How?

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

               non-static local
void f1(int p) {
   static int x = 22;  static local variable
   x = x + p * g;
   printf("%d\n", x);
}

int main(void) {
   f1(g);
   g = 2;
   int g = 1;    non-static local, shadowed global variable g
   f1(g);
   return 0;
}
```

Output
143
145

shadowing: when local variable blocks access to global variable

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

Where do I live?

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

highlighted is memory allocations

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

int gus = 14;     // global variable, in DATA segment, .data file
int guy;          // global variable, in DATA segment, .bss file

int madison(int pam) {        // non-static local variable, in STACK segment

    static int max = 0;       // static local variable, in DATA segment, .bss file
    int meg[] = {22,44,88};   // non-static local variable, in STACK segment      SAA
    int *mel = &pam;          // non-static local variable, in STACK segment
    max = gus--;
    return max + meg[1] + *mel;
}

int *austin(int *pat){        // non-static local variable, in STACK segment
                              // (parameters treated like local variables non-static)
    static int amy = 33;      // static local variable, in DATA segment, .data file
    int *ari = malloc(sizeof(int)*44); ari: // non-static local variable, in STACK segment
    gus--;          memory ari points to is on heap
    *ari = *pat;
    return ari;
}

int main(int argc, char *argv[]) {    argc: // non-static local variable, in STACK segment
                                      argv: // non-static local variable, in STACK segment
   int vic[] = {33,66,99};            // non-static local variable, in STACK segment
   int *wes = malloc(sizeof(int));    wes: // non-static local variable, in STACK segment
   *wes = 55;  memory wes points to is on heap
   guy = 66;
   free(wes);
   wes = vic;
   wes[1] = madison(guy);
   wes = austin(&gus);
   free(wes);                         string literals in CODE segment
   printf("Where do I live?");
   return 0;
}
```

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

if array of strings, array still on stack but string literals on the code segment

pointers can store any address but can get SEG FAULT if try to dereference to address without permission

Linux: Processes and Address Spaces

program when running what can i do with program while running

Process and Job Control

in reality not much

- Linux is multi-tasking OS, where you can run multiple processes concurrently

`ps` list snapshot of user processes, for everyone “ps -e”

`jobs` list only user processes, from command line

`&` put process in background

ctrl+z suspend running process

`bg` put suspended process in background

`fg` bring a process to foreground

ctrl+c stop running foreground process

top display table of resource usage processes

Program Size

size <executable or object_file> display size of program’s memory segments

.rodata .text .data .bss

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

CODE DATA

size in decimal and hex

Virtual Address Space Maps

- Linux enables you to see Vitual Address Space (mem map) of each process

`$pmap <pid_of_process>`

`$cat /proc/<pid_of_process>/maps` magic number, stack, libraries, Virtual Dynamic

`$cat /proc/self/maps` notice heap

/proc: virtual file system that reveals KERNEL data in ASCII text from that can be read by progs

cat /proc/loadavg

SPEC CODES EF 10	UW LOGIN NAME	Last, First Name (as in email and on scantron)

Computer Sciences 354

Midterm Exam 1 Primary

Thursday, October 5th, 2023

60 points (15% of final grade)

Instructor: Debra Deppeler

1. **RECORD Special Codes for EF on scantron. Ask Proctor if there are not 2 digits.**.
2. **PRINT your UWNET ID (login name not photo id number) in box above**.
3. **PRINT Last, First Name in box above**.
4. **SCANTRON Fill in all fields and their bubbles on the scantron form (must use #2 pencil on scantron form).**
   (a) LAST NAME field - left align last name as given in room email
   (b) FIRST NAME field - left align first five letters of your first name as given in email
   (c) IDENTIFICATION NUMBER your UW Student WiscCard ID number
   (d) SPECIAL CODES E - write and fill-in bubble for your exam version number 1
   (e) SPECIAL CODES F - write and fill-in bubble for your room number 0
5. **FILL IN BUBBLES FOR ALL IDENTIFICATION FIELDS and for SPECIAL CODES COLUMNS E and F.**
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 may 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	2 pt Simple Choice	20
II	12	3 pt Multiple Choice (+bonus)	36
III	4	Survey	4
	28	Total	60

Assumptions unless instructions explicitly state otherwise:

+ addresses and integers are 4 bytes unless explicitly stated otherwise.
+ code questions are about C std=gnu99 and IA-32 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 off and put away all notes and electronic devices and wait for the proctor to signal the start of the exam.