CS 354 - Machine Organization & Programming
Tuesday Sept 26 and Thursday Sept 28, 2023

Midterm Exam - Thursday, October 5th, 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**

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, Sept 29

Project p2B: Due on or before next week Friday, Oct 6

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

- use **<stdio.h>** functions: **printf**, **scanf**, **perror**, **sscanf**, **sprintf**, **fopen**, **fclose**, **fgets**, **fputs**
- use predefined file points: **stdin**, **stdout**, **stderr**
- 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

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

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
```

fgets

`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 format specifiers & chars to skip (not read)

format specifiers %d, %f, %c, %s, %p, %i
↳ can use octo or hex input

whitespace input separator " " "\t" "\n"

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 characters to print

("Hello %s", name)
\n to flush buffer

%d decimal
%f float
%c char
%p ptr
%i int

Standard Error

`void perror(const char *str)` allows programmer to choose which output streams

writes formatted error output to the console terminal window

String I/O

`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

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

File I/O in `stdio.h`

Standard I/O Redirection

overwrites

a.out < in_file >out_file

>> append_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
```

`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

if (Fopen(__, __) == NULL) { printf(); exit(1); }

```
int fclose(FILE *stream)
```
flushes the output buffer and then closes the specified `stream`
returns 0, or EOF if error

if (fclose(outfile) != 0) printf("unable to close outfile \n");

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

do not name program cp

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);
   }

   // create buffer (space) to store strings between read & write
   const int bufsize = 257; //WARNING: assumes lines <= 256 chars
   char buffer[bufsize];
           // SAA
                                  room for terminating char
                                                  or EOF

   while ( fgets ( buffer, bufsize, ifp ) != NULL )
           fputs ( buffer, ofp );

   if ( fclose (ifp) != 0 ) { print error

   if ( fclose ( ofp ) != 0 ) { print error
   return 0;
}
```

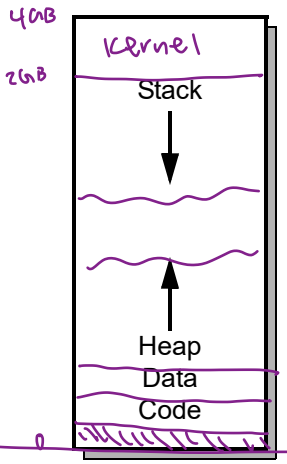
Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

Three Faces of Memory

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

① **Process View = Virtual Memory**

Goal: provide a simple view to programs



virtual address space (VAS):
illusion that each process has its own address space

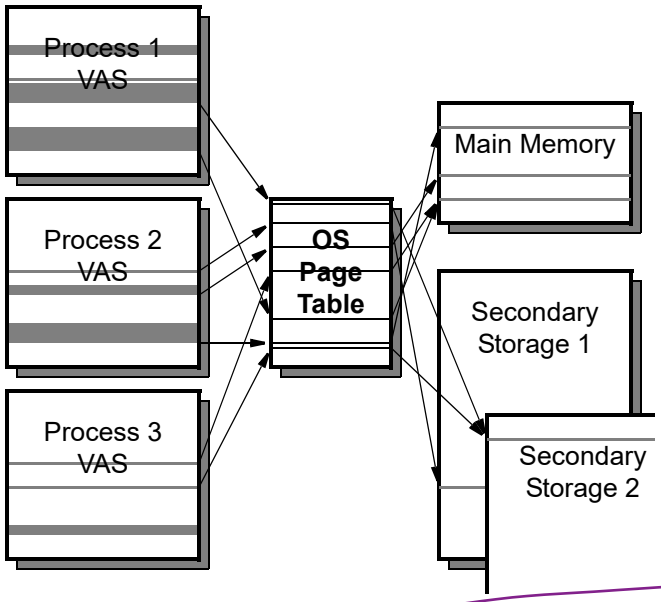
virtual address: simulated address that process generates

② **System View = Illusionist (CS 537)**

Goal: make memory shareable & secure

pages: 4kb unit = 4096 bytes

page table: data st. in O.S. that maps virtual pages to physical pages
- ensures that processes cant interfere w/ each other
- only V pages have P pages

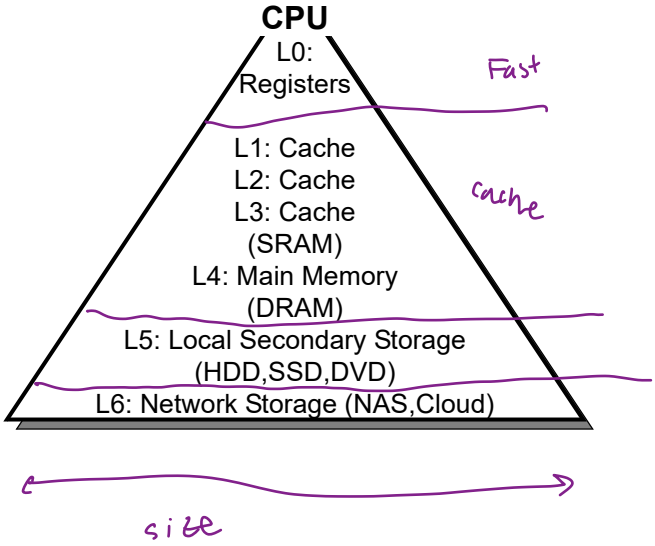


③ **Hardware View = Physical Memory**

Goal: keeping CPU busy

physical address space (PAS):
- multilevel hierarchy
- ensure freq. accessed data is close to CPU

physical address: addr used to access machine mem



Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

Virtual Address Space (IA-32/Linux)

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

4GiB = 11111111111111111111111111111111 = 0xFFFFFFFF

address space:

the range of valid memory addr. for process

process: a running program

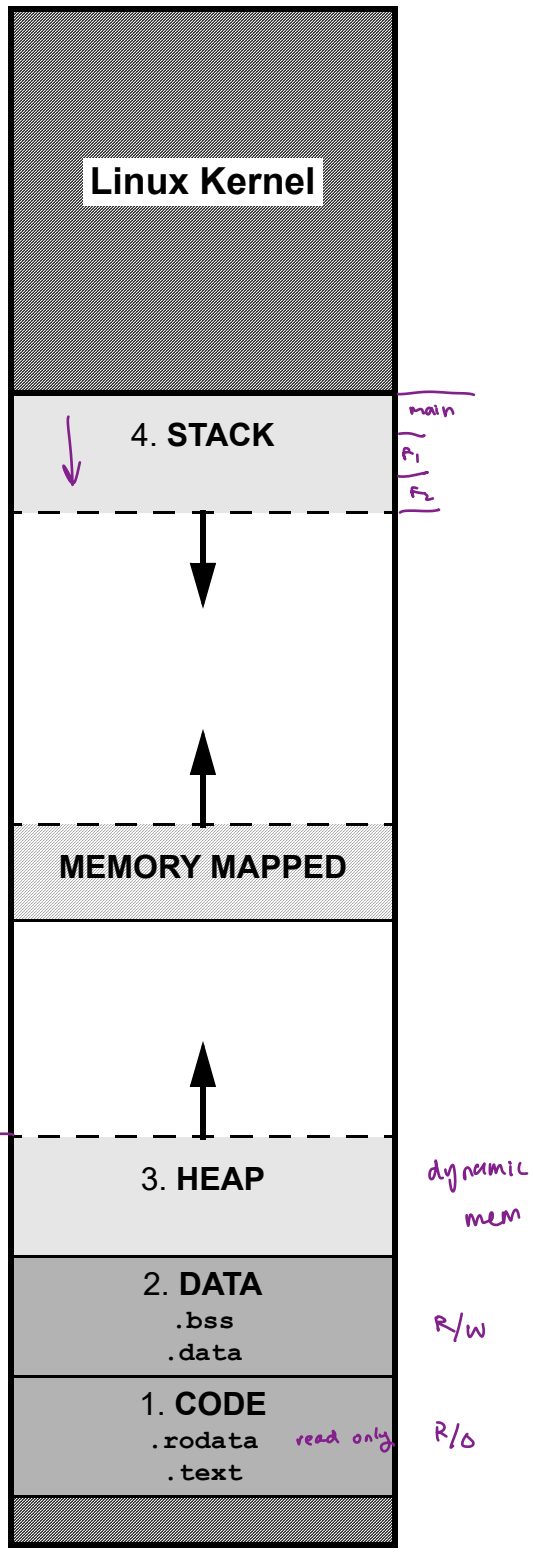
2^{31} = 2GiB

2^{31} = 10000000000000000000000000000000 = 0xC0000000

kernel: mem. resident portions of O.S.

user process: process that is not kernel

- *Every user process* has simple view of memory



magic number

00001000000001001000000000000000 = 0x08048000

0 = 00000000000000000000000000000000 = 0x00000000

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

C's Abstract Memory Model

1. CODE Segment

Contains: the prog. instructions

.text section binary machine code

.rodata section string literals

Lifetime: entire program's execution

Initialization: from executable object file (a.out) by loader EOF

Access: read only

2. DATA Segment

Contains: local variables (global?) and static local vars

Lifetime: entire program's execution

Initialization: from EOF by loader

.data section initialized to non-zero values

.bss section zero or uninit. values

Block started by symbol

Access: read/write

3. HEAP (AKA Free Store)

Contains: mem that is allocated and freed by prog. in runtime

Lifetime: managed by programmer (malloc, calloc, free, realloc)

Initialization: no init. by fault

Access: read/write

4. STACK (AKA Auto Store)

Contains: memory in stack frames, auto alloc'd and free'd by func calls and returns

stack frame (AKA activation record) non static local vars, parameters, temp vars, more ...

Lifetime: from declaration until end of scope

Initialization: none by default

Access: read/write

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

Meet Globals and Static Locals

What?

A *global variable* is

- decl. outside of a func
- accessible to all funcs
- allocated in data segment (not part of stack)

A *static local variable* is

- decl in a func. w/ static modifier : static int count
- accessible only w/in the func (after decl)
- alloc'd in data segment

Why?

for storage that exists for entire prog.

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

How?

```
#include <stdio.h>
int g = 11;    // global var : data - .data

void f1(int p) {  // p is in stack
   static int x = 22; // static local : data - .data
   x = x + p * g;
   printf("%d\n", x);
}

int main(void) {
   f1(g);
   g = 2;
   int g = 1; // non static local. stack , shadows a global var g
   f1(g);                                  can no longer access global
   return 0;
}
```

shadowing: when local var blocks access to global w/ same name

❋ *Avoid shadowing; don't use the same identifier*

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

Where do I live?

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

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

int gus = 14; // global : data - .data
int guy; // global : data - .bss

int madison(int pam) {  // stack

    static int max = 0; // static local: data- .bss
    int meg[] = {22,44,88}; // SAA
    int *mel = &pam; // stack
    max = gus--;
    return max + meg[1] + *mel;
}

                  params on stack
int *austin(int *pat){  //stack

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

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

   int vic[] = {33,66,99}; // SAA
   int *wes = malloc(sizeof(int)); // stack
   *wes = 55;          if static ... = malloc
   guy = 66;                entire array in data
   free(wes);
   wes = vic;
   wes[1] = madison(guy);
   wes = austin(&gus);
   free(wes);
   printf("Where do I live?");  // string literal : code - .ro data
   return 0;
}
```

ari: on stack but → ptr to heap

❋ *Arrays, structs, and variables* can live in Data, Heap, or Stack

ptr vars can store any addr, but if you deref. an addr outside process mem. seg., then seg fault

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler

Linux: Processes and Address Spaces

Process and Job Control

- Linux is multi-tasking O.S. where you can run multiple processes at same time

ps list a snap shot of usr processes

jobs lists process started from command line

& put process in background

ctrl+z suspends the running process

bg - put suspended process in background
fg - brings process to foreground

ctrl+c stop running foreground process

top table of resource usage

Program Size

size <executable or object_file> size a.out

display size of prog's mem segment

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

Virtual Address Space Maps

- Linux enables you to see Virtual Addr space (mem map) of each process

```
$pmap <pid_of_process>

$cat /proc/<pid_of_process>/maps

$cat /proc/self/maps
```

/proc: Virtual file system that reveals kernel data in ASCII

[ ] Lec 001 9:30am TR [ ] Lec 002 1:00pm TR 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.

Copyright © 2016-2023 Jim Skrentny, Deb Deppeler