CS 354 - Machine Organization & Programming
Tuesday April 18, and Thursday April 20, 2023

Homework hw6: DUE on or before Monday Apr 17

Homework hw7: DUE on or before Monday Apr 24

Project p5: DUE on or before Friday April 21st

Project p6: Assigned soon and Due on May 5th, last day of classes. No late day, no Oops on p6.

Last Week

Function Call-Return Example (L20 p7) Recursion Stack Allocated Arrays in C Stack Allocated Arrays in Assembly Stack Allocated Multidimensional Arrays	Stack Allocated Structs Alignment Alignment Practice Unions

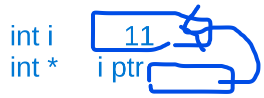
This Week

Pointers Function Pointers Buffer Overflow & Stack Smashing Flow of Execution Exceptional Events Kinds of Exceptions	Transferring Control via Exception Table Exceptions/System Calls in IA-32 & Linux Processes and Context User/Kernel Modes Context Switch Context Switch Example
Next Week: Signals, and multifile coding, Linking and Symbols B&O 8.5 Signals Intro, 8.5.1 Signal Terminology 8.5.2 Sending Signals 8.5.3 Receiving Signals 8.5.4 Signal Handling Issues, p.745	

Pointers

Recall Pointer Basics in C

```
int i = 11;
int *iptr = &i;
*iptr = 22;
```



pointer type `int *`

pointee type used by compiler to determine the scaling factor used in ASM   4(%ebx, %ecx, 2)

pointer value `0x2A300F87`, `0x00000000` (NULL)
address used with addressing modes to specify an effective address in ASM

address of `&i`
& operator, becomes `leal` instr, which just calculates the effective address

dereferencing `*iptr`
* operator, becomes `mov` instr, which accesses mem at the effective address

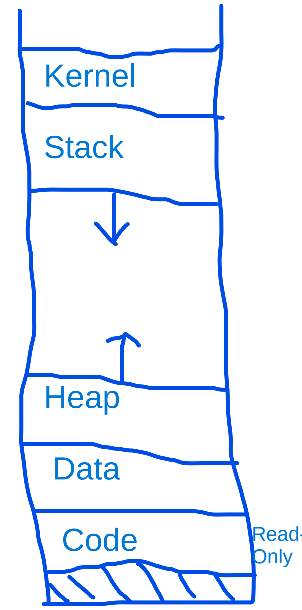
Recall Casting in C

```
int *p = malloc(sizeof(int) * 11);
```
//44 bytes = payload

```
... (char *)p + 2
```

❋ *Casting changes the scaling factor used not the pointer's value.*

Function Pointers



What? A *function pointer*

- a pointer to code
- stores addr of 1st instruction of function

Why?

enables functions to be:

- passed and returned from functions

- stored in arrays, faster switch logic - jump tables

How?

```
int func(int x) { ...}       //1. implement some function

int (*fptr)(int);            //2. declare function pointer, var to store function addr

fptr = func;                 //3. assign its function

int x = fptr(11);            //4. use function pointer
```

int

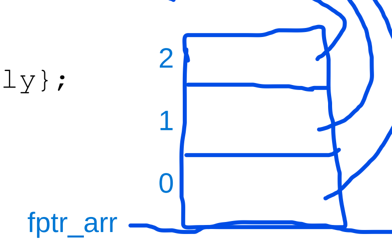
Example

```
#include <stdio.h>

void add      (int x, int y) { printf("%d + %d = %d\n", x, y, x+y); }
void subtract (int x, int y) { printf("%d - %d = %d\n", x, y, x-y); }
void multiply (int x, int y) { printf("%d * %d = %d\n", x, y, x*y); }

int main() {
    void (*fptr_arr[])(int, int) = {add, subtract, multiply};
    unsigned int choice;
    int i = 22, j = 11; //user should input

    printf("Enter: [0-add, 1-subtract, 2-multiply]\n");
    scanf("%d", &choice);
    if (choice > 2) return -1;
    fptr_arr[choice](i, j);
    return 0;
}
```



Buffer Overflow & Stack Smashing

Bounds Checking

```
int a[5] = {1,2,3,4,5}; //index 0 - 4
printf("%d", a[11]);    //gcc "no problem" (no bounds checking)
                        //run-time   undefined behavior
```

→ What happens when you execute the code?

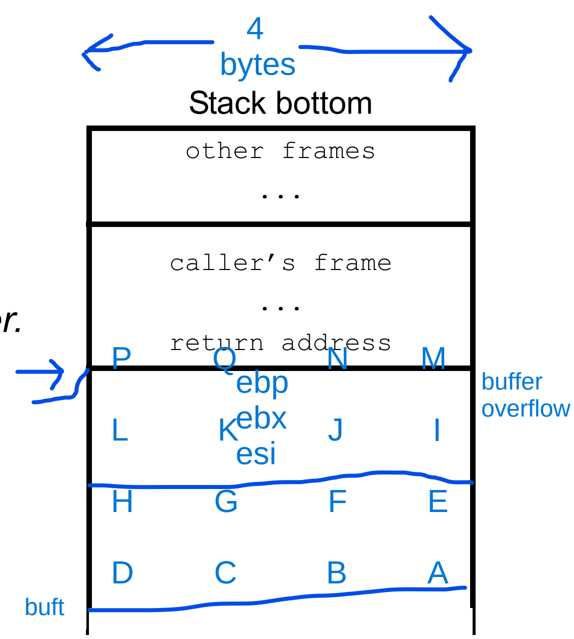
prints junk, 0, seg fault => crash

❋ *The lack of bounds checking array accesses* is one of C's main vulnerabilities.

Buffer Overflow

- exceeding bounds of array
- particularly dangerous for stack-allocated arrays

```
void echo() {
   char bufr[8];
   gets(bufr);
   puts(bufr);
}
```



❋ *Buffer overflow can overwrite data outside the buffer.*

❋ *It can also overwrite the state of execution!*

Stack Smashing

1. Get "exploit code" in
   enter input crafted to be machine instrs

2. Get "exploit code" to run
   overwrite return address with addr of buffer with exploit code

3. Cover your tracks
   restore stack so execution continues as expected

❋ *In 1988 the Morris Worm brought down the Internet using this kind of exploit.*

Flow of Execution

What?

control transfer a transition from one instruction to another

control flow a sequence of control transfers

- What control structure results in a smooth flow of execution? sequential

- What control structures result in abrupt changes in the flow of execution? selection, repetition, function calls & returns

Exceptional Control Flow

logical control flow normal expected execution flow

exceptional control flow special "unexpected" execution

allows systems to react to unusual / urgent / anomalies

event a change in processor state that may or may not be related to current instruction

processor state internal storage elements

regs, flags, signals, etc.

Some Uses of Exceptions

process to ask for kernel services
to share info with other processes
to send / receive messages (signals)

OS to communicate with our process
to switch execution among processes

hardware to indicate device status

Exceptional Events

What? An *exception*

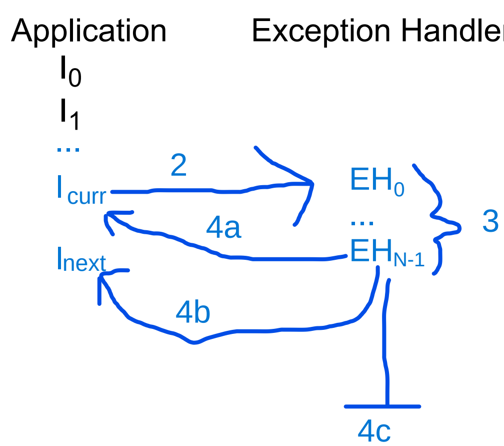
- is an event that side-steps logical flow
- can originate from hardware or software
- an indirect function call that abruptly changes the flow of execution

→ What's the difference between an asychronous vs. a synchronous exception?

asynchronous result from an event that is unrelated to the current instruction

synchronous result from current instruction (e.g. seg fault, divide by 0)

General Exceptional Control Flow



0. normal flow

1. exception occurs

2. transfer control to exception handler

3. appropriate E.H. code runs

4. Return control to:
   a. I_{curr} (page fault)
   b. I_{next}
   c. OS Aborts

Kinds of Exceptions

→ Which describes a ***Trap***? ***Abort***? ***Interrupt***? ***Fault***?

1. Interrupt

signal from external device
asynchronous
returns to Inext

async

How? Generally:
1. Device signals interrupt
2. Finish current instruction
3. transfer control to appropriate exception handler
4. transfer control back to interrupted process's next instruction

vs. *polling* - where software periodically checks devices
(synchronous)

2. Trap - enable process to interact with OS

intentional exception
synchronous
returns to Inext

How? Generally:
1. process indicates need for OS service

assembly instr
that means
INTERRUPT,
NOT integer

`int` execute interrupt instruction

2. transfer control to the OS system call handler

which "calls" executres requested svc

3. transfer control back to process's next instruction

sync

3. Fault

potentially recoverable error
synchronous
might return to Icurr and re-execute it

4. ABORT - cleanly ends a process

nonrecoverable fatal errors
synchronous
doesn't return

Transferring Control via Exception Table

❋ *Exceptions transfer control to the Kernel.*

Transferring Control to an Exception Handler

1. push return address

2. push interrupted process's context state so it can be restarted

   context state - state of registers, etc.

→ What stack is used for the push steps above?

Kernel's stack

3. do indirect function call which executes appropriate exception handler

indirect function call uses exception table to determine what function to execute

EHA = M[R[ETBR] + ENUM]

ETBR is for exception table base reg

ENUM is for exception number

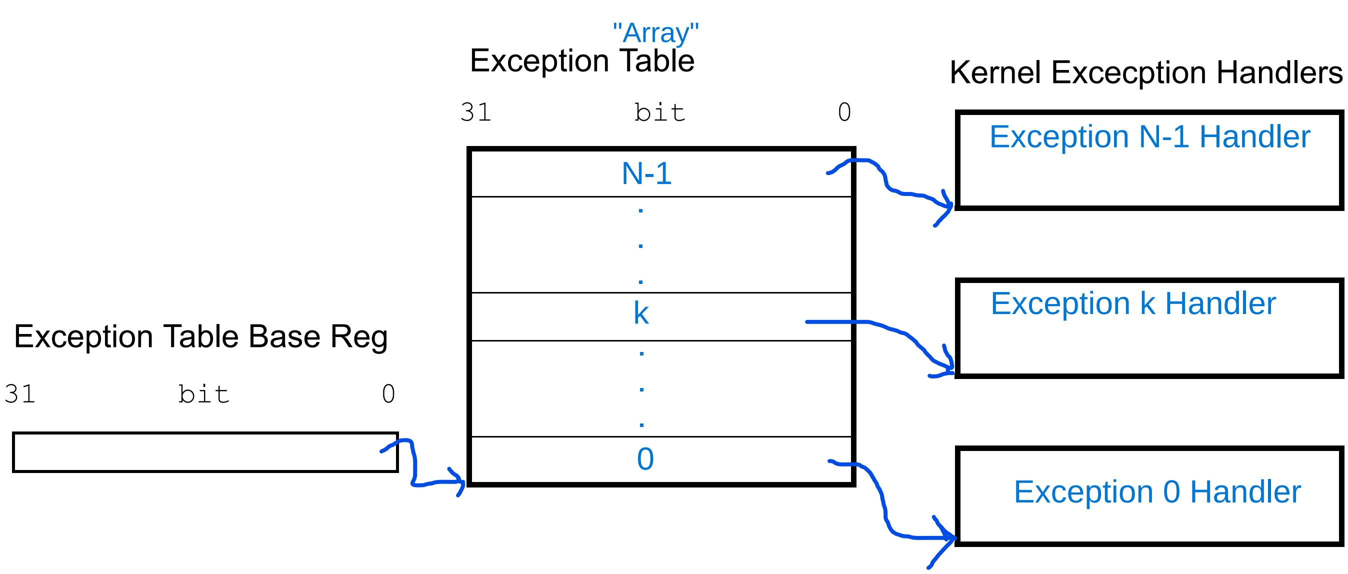
EHA is for exception handler's address

Exception Table

is a "jump" for exceptions that's allocated in memory by the OS on system boot

exception number

a unique non-negative integer associated with each type of exception



Exceptions/System Calls in IA-32 & Linux

Exception Numbers and Types

0 - 31 are defined by processor

0 - divide by zero
13 - protection fault
14 - page fault
18 - machine check

***don't memorize numbers for exams (except maybe 0), remember what types are/do

32 - 255 are defined by OS

128 ($0x80) = system call

System Calls and Service Numbers

*1 exit
2 fork
whole row->*3 read file   4 write file   5 open file   6 close file] file IO
11 execve

Making System Calls

1.) put service number in %eax

2.) put system call arguments in next registers (except %esp)

3.) `int $0x80` (interrupt with system call)

System Call Example

```
#include <stdlib.h>
int main(void) {
   write(1, "hello world\n", 12);
   exit(0);
}
```

1: stdout
"hello world\n": char string
12: # of chars to print

Assembly Code:

DATA

```
.section .data
string:
   .ascii "hello world\n"
string_end:
   .equ len, string_end - string
```

CODE

```
.section .text
.global main
main:
```

Code	
`movl $4, %eax`	put 4 in eax - "write file"
`movl $1, %ebx`	put 1 in ebx - 1st arg, "stdout"
`movl $string, %ecx`	put address of string in ecx - 2nd arg
`movl $len, %edx`	put length in edx - 3rd arg
`int $0x80`	- GO! DO SYSTEM CALL to write to stdout
`movl $1, %eax`	put 1 in eax - "exit"
`movl $0, %ebx`	put 0 in ebx - 1st (only) arg
`int $0x80`	-GO! DO SYSTEM CALL to exit with code 0

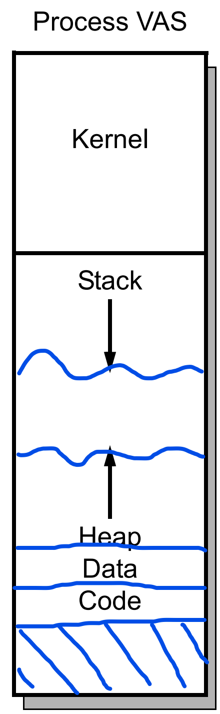
Processes & Context

Recall, a *process*

- is an instance of a program (executing)
- has context ("state")   the info to restart process

Why?

easier to treat each process as single entity
as if it's running by itself



Key illusions - each process has its own:
1. CPU
2. memory - (VAS)
3. devices -

→ Who is the illusionist? OS

Concurrency - multi-threading, multiprocessing, multi-tasking

combined execution of 2 or more processes

scheduler - kernel code to switch among processes

interleaved execution - 1 CPU among all processes

time slice - (a.k.a quantum) ~ 1-10 ms   interval before switching to a process

time	proc A	proc B	proc C
A	↓		
B		↓	
A	↓		
B		↓	
C			↓
A	↓		

1 CPU

parallel execution - multiple core

time	proc A	proc B	proc C
A	↓		
A,B	↓	↓	
A,B,C	↓	↓	↓
A,B,C	↓	↓	↓
A,C	↓		↓

3 CPU

User/Kernel Modes

What? Processor *modes* are different privilege levels

kernel = 1
user = 0

mode bit indicates privilege level

1 kernel mode can execute any instruction
can access any memory
can access any device

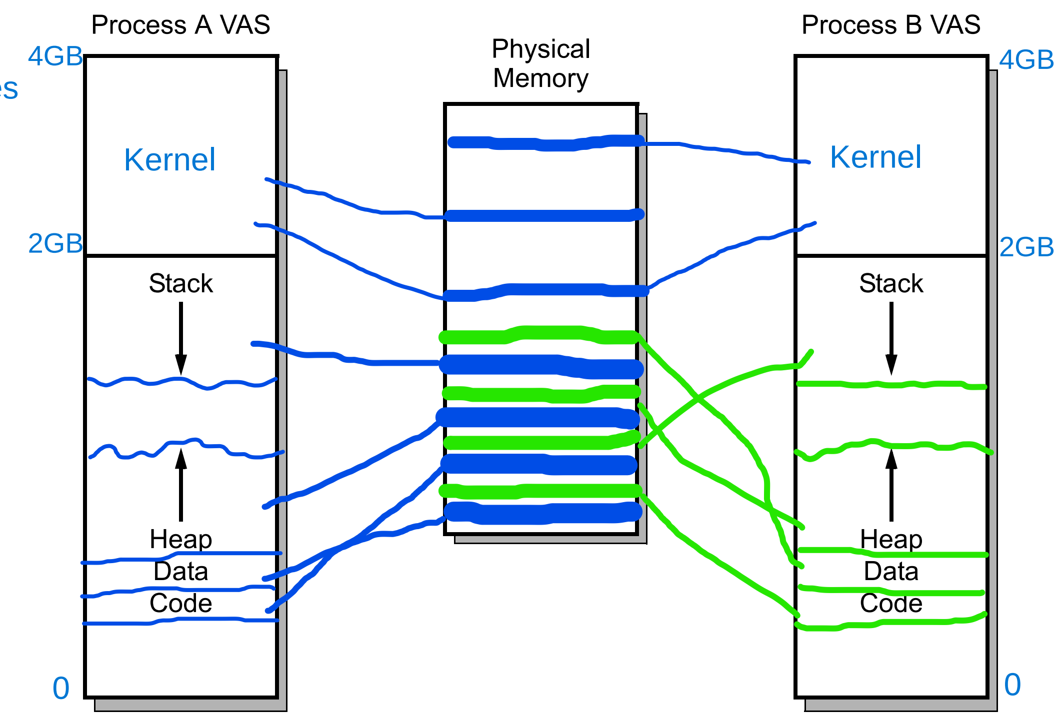
0 user mode can execute some instructions
can access some memory
access to devices via OS

flipping modes

- Start in user mode
- Only exception can switch from user to kernel mode
- In kernel exception Handler can switch to user mode

Sharing the Kernel

Key part of OS
+ shared by all processes
+ mem. resident



Context Switch

What? A *context switch*

- When OS switches from one running process to another

- requires preservation of process's context

  1. CPU state
  2. user stack %ebp %esp
  3. kernel stack
  4. kernel data structures
     a. Page Table
     b. process table
     c. file table

When?
happens as result of exception (including interrupts) when kernel needs to switch to another process

Why?
enables exceptions to be processed

How?

1. save context of current process

2. restore context of process we are changing to

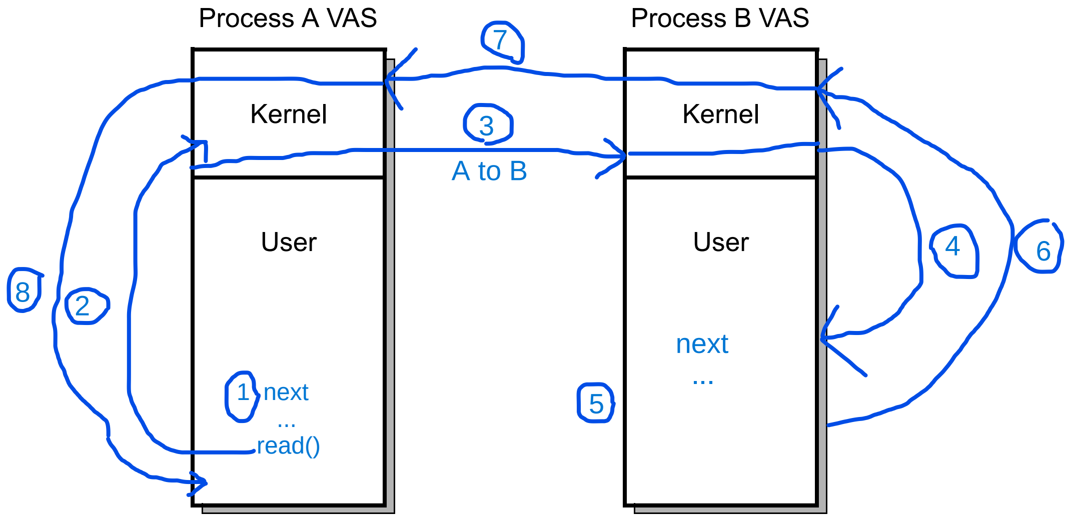
3. transfer control to next process

✵ *Context switches* are very expensive!

→ What is the impact of a context switch on the cache? Negative
"cache pollution"

Context Switch Example

Stepping through a `read()` System Call



1. Process A is running in USER MODE . . . get to read()

int 0x80 (128)

2. Switch to kernel mode . . . run exception handler for svc #3

3. In kernel mode . . . DO CONTEXT SWITCH

   save procA
   restore procB
   transfer to B

4. Switch to user mode

5. In user mode in process B
   interrupt occurs from HW done reading into mem

6. Switch to kernel mode

7. IN kernel mode -- do context switch to A

   save B
   restore A
   transfer to A

8. Switch to user mode -- continue to proc A