CS 354 - Machine Organization & Programming
Tuesday April 16, and Thursday April 18, 2023

Homework hw6: DUE on or before Monday Apr 15

Homework hw7: DUE on or before Monday Apr 22

Project p5: DUE on or before Friday April 19

Project p6: Assigned soon and Due on May 3rd, 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 *` 32 bit machines always 4 bytes long

pointee type used by compiler to determine the scaling factor used in ASM

use leal to compute addresses

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

movl and reference w some register eg (%ecx)

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

leal

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

movl

Recall Casting in C

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

```
... (char *)p + 2
```
scale factor will match type, in this case char *, so add 2 bytes to address of p

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

Function Pointers

What? A *function pointer*

- is a pointer to code
- stores the address of the first byte of a function

Why?

enables functions to be

- pass to or return functions
- lets us store function pointers in arrays - jump tables

How?

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

int (*fptr)(int);           //2. declare function pointer
fptr = func;                //3. assign its function

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

fptr can only take a single argument and return an int

char func2(int, char)
char (*name)(int, char)

(*fptr)(11)

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

if we did a[11] = 55;
mutates caller's stack frame instead

```
int a[5] = {1,2,3,4,5};
printf("%d", a[11]);
```

→ What happens when you execute the code?

probably get some intermittent error, getting data from elsewhere in the stack (not current stack frame)

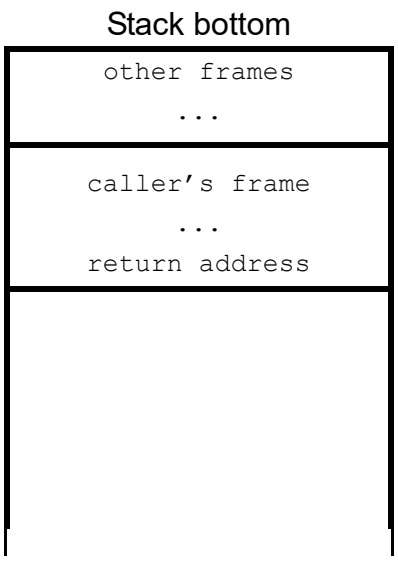
❋ *The lack of bounds checking array accesses is one of C's main vulnerabilities.*

Buffer Overflow can happen on the stack or on the heap

- common way this happens is array out of bounds
- really dangerous on the stack

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

ok if less than 8 characters typed in,
9th character mess with caller's ebp
more than 12 chars, mess w return addr
=> BAD security vulnearbility



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

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

in particular the return address

Stack Smashing

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

if can figure out where machine is going to drop the exploited code, can get machine to jump there to gain control of machine

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 from instruction to instruction

control flow jmp, je (conditional jumps etc),

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

  sequencing

- What control structures result in abrupt changes in the flow of execution?

  repetition, loops, if-then-else, call/return

Exceptional Control Flow

logical control flow

this is what our code does

exceptional control flow

ways the machine can react to exceptional events — eg pushing key on keyboard, arrival of network packet
anything that is urgent / unusual / anomalous events for the machine

event

changing processor state
might be related to our code or could be completely different

processor state

content of our registers, condition flags, signals

Some Uses of Exceptions

process

ask the kernel to do something for us (make exceptional call to the kernel)
Kernel can do on our behalf, I/O, share data, send and receive signals

OS

is the bridge for us to communicate with processes and hardware
switch between running processes
deal with paging (virtual address space)

hardware

will tell us when operations are ready
timer events that will fire and let os decide what to do
packet arrival from the network with interrupt processor

Exceptional Events

What? An *exception*

- is an event that side steps normal flow
- can come from hardware, or software
- abruptly changes the flow of execution

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

asynchronous — eg interrupts, unrelated to current execution flow
such as network packet, it comes when it comes, cant control
phone call is an eg of async interrupt

synchronous — result from executing current instruction, eg dividing a number by zero, eg seg fault

General Exceptional Control Flow

Application | Exception Handler

0. normal flow
   I_0
   I_1

1. exceptional event occurs

2. transfer control to exception handler

3. exception handler runs

4. return control back to one of 3 places:
   - current instruction (cannot finish what its doing)
   - next instruction
   - some place into OS to abort the program

Kinds of Exceptions

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

1. INTERRUPT — primary way hardware gets attention

signal from external device storage network (NICs, timers)
asynchronous
returns to Inext

How? Generally:

1. Interrupt occurs
2. Processor finishes current instruction
3. transfer control to appropriate exception handler
4. transfer control back to interrupted process's next instruction

vs. ***polling*** very low latency, but very wasteful from resource standpoint (processor spends alot of time waiting)
eg keeps checking for packets
interrupts are more efficient

2. TRAP — ways to enable process to interact with the OS (eg I/O call)

intentional exception
synchronous
returns to Inext

How? Generally:

1. process sets up some registers and calls INTERRUPT instruction

   `int` instruction IA-32 => int 0x80 or int 128

2. transfer control to the OS system call handler
   stuff to be done before transfer control:
   - eax has the system call number
3. transfer control back to process's next instruction

3. FAULT — potentially recoverable error, eg page fault (process is trying to access some piece of memory in VAM, but OS hasnt allocated yet or page has been swapped out to disk)

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

4. ABORT— OS will terminate your program

nonrecoverable fatal errors if very bad, OS might panic
synchronous
doesn't return

Transferring Control via Exception Table

❋ *Exceptions transfer control to the Kernel.*

Transferring Control to an Exception Handler

1. push address that was executing — could be curr instr for a FAULT or next instr for TRAP
2. push processor state (context switches)

→ What stack is used for the push steps above?
Kernel stack is used

3. do indirect function call executes the appropriate handler
have to go into kernel mode — returns to user mode when its done

indirect function call

EHA = M[R[ETBR] + ENUM * 4] (exception handler address)

ETBR is for exception table base reg

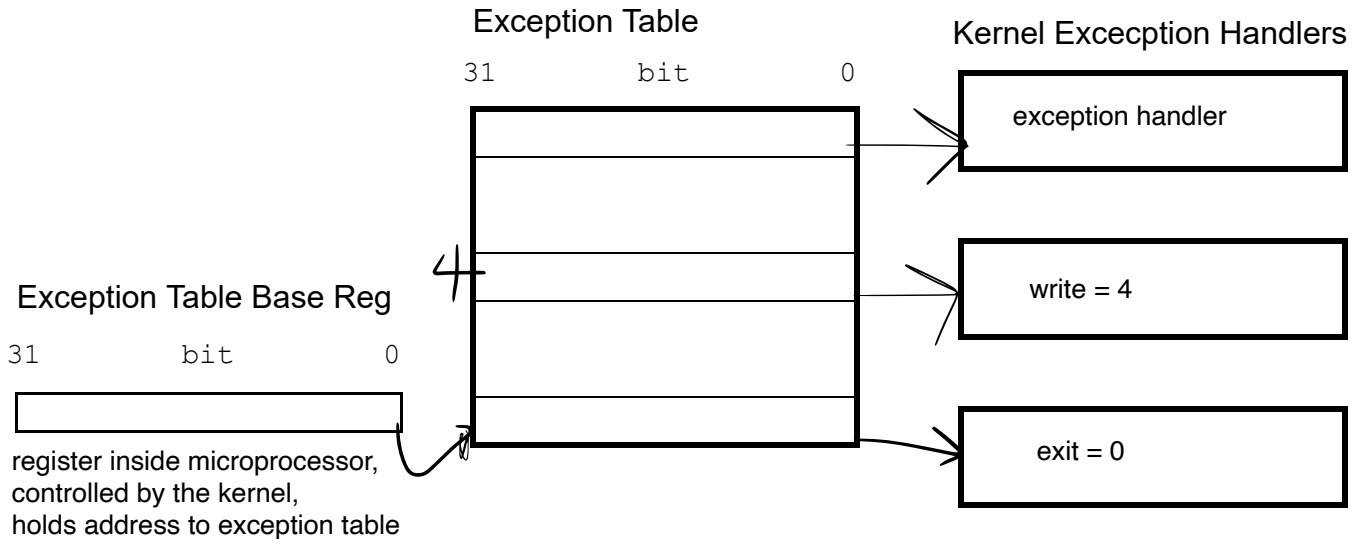
ENUM is for exception number

EHA is for exception handler's address

Exception Table

is a jump table

exception number 8 bit number



Exceptions/System Calls in IA-32 & Linux

Exception Numbers and Types

0 - 31 are defined by processor

- 0 divide by zero
- 13 protection fault (seg fault)
- 14 page fault
- 18 machine check exception (hardware failure)

32 - 255 are defined by OS

- 128 ($0x80) TRAP for system calls

System Calls and Service Numbers

1 exit

2 fork fork — unix gets another process running, one parent calls fork and one child process runs, 2 returns

3 read file 4 write file 5 open file 6 close file

11 execve get OS to run another program, use fork to create another program and use execve to run program

current linux version has 548 of such system calls

Making System Calls

1.) put call number into %eax

2.) parameters to system call are passed through registers

3.) `int $0x80`

System Call Example

calling write — low level system call

3 parameters, first is file descriptor, in unix every open file has a file descriptor, 0 — std input, 1— std output, 2— std error

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

first time open file is file descriptor 3

Assembly Code:

```
   .section .data
   string:
      .ascii "hello world\n"
   string_end:
      .equ len, string_end - string
   .section .text
   .global main
   main:
      movl $4, %eax
      movl $1, %ebx
      movl $string, %ecx
      movl $len, %edx
      int $0x80
      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

is in the data section

is in the data section

system call

ebx is file descriptor to write to

ecx is pointer to data

edx is length of data

} params for write

move 1 into eax, system call for exit

file descriptor of 0 (std input) to ebx

Processes & Context

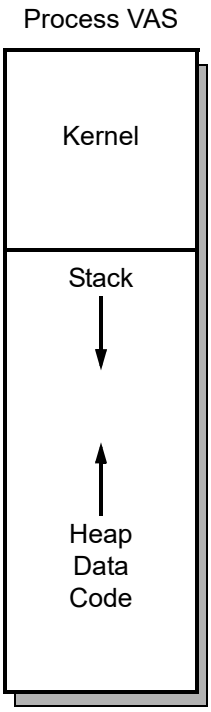
program can be instatiated as a process many times by machine

Recall, a *process*

- is an instance of a running program
- context - the stuff needed to be restarted after interrupting

Why?

abstraction for running programs



Key illusions

1. cpu - to us looks like only process running
2. memory — looks like we have entire space to ourselves (virtual memory)
3. devices — HDD, SSD, Terminal (cannot interact with them, only way is to ask OS to interact with them)

→ Who is the illusionist?

Operating System

Concurrency

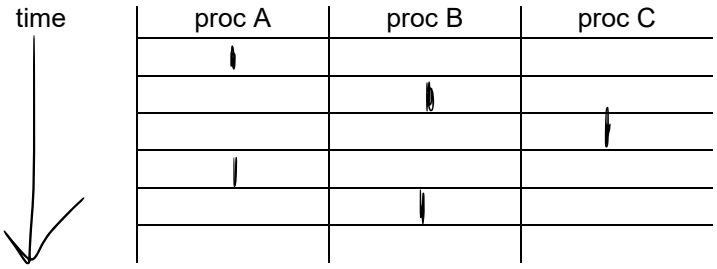
2 types — multiprocessing & multithreading

kernel schedule threads to run,
2 process isolated from each other -> multiprocessing (2 boxes)

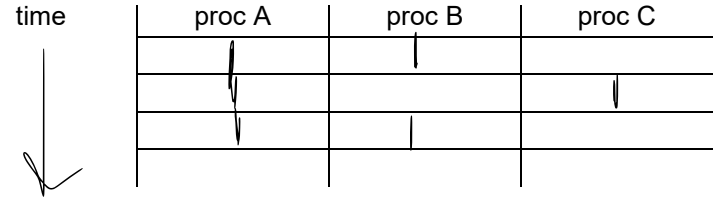
scheduler choosing what runs next

interleaved execution

time slice quantum — 10ms — kernel runs process, sets ~10ms, then timer goes off, kernel receives interrupt then look at which process to run next



parallel execution if 2 CPU cores



User/Kernel Modes

What? Processor *modes* are privilege levels

mode bit single mode bit to define 1 = kernel/supervisor mode, 0 = user mode

kernel mode
- can run any instruction,
- can access any memory,
- can communicate with devices

user mode
- some instructions are disabled, eg halt to stop kernel running (only kernel can)
- only allowed to "your" memory allocated by the OS
- devices — have to go through the kernel

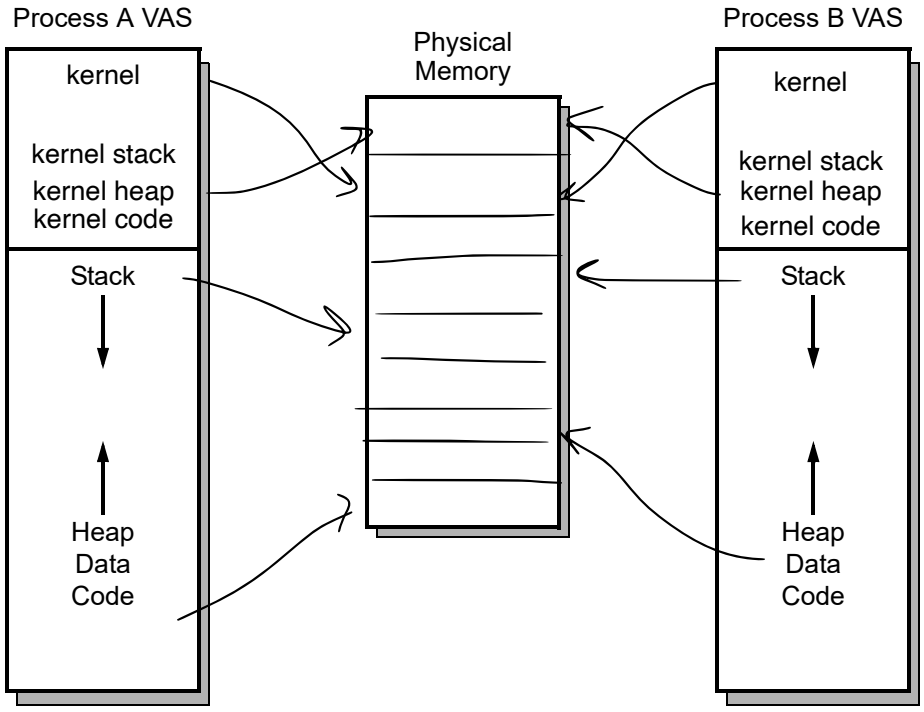
flipping modes

- if sitting in user mode
- some exception will occurs — processor switch from user to kernel mode
- kernel does its work — after figure out what to run next, switch back to user mode before it transfer control back

Sharing the Kernel

only kernel is shared,
diff processes kernel map to same physical memory

OS tries to give least amt of physical memory that it can get away with



Context Switch

What? A *context switch*

- what happens when the OS stops one process and starts another existing process

things to be preserved and recorded:

- - CPU state %eip
  - registers
  - stack pointers
  - process state (what files do you have open)
  - virtual memory info — page table
  - file table

When? only happen when kernel decides it should happen
only can happen after we enter the kernel — either hardware interrupt, timer, etc

Why? how we get the scheduler to run

How?

1. save context/state of running process, decide what process gets to run next
2. restore process state
   return to user mode
3. transfer control back to the restored process

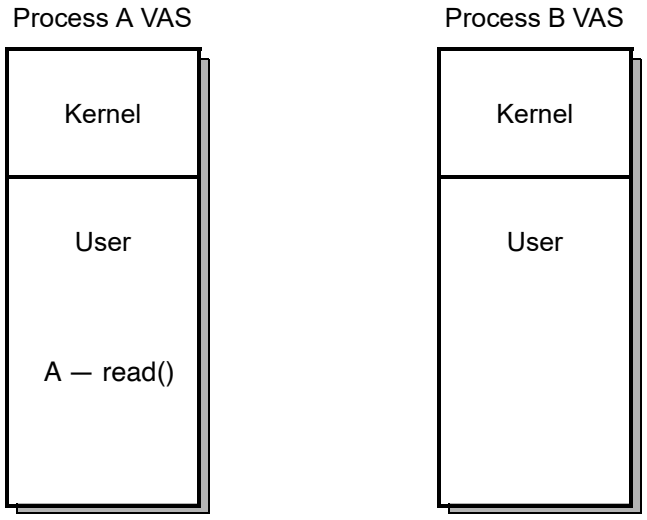
✵ *Context switches* are very expensive
for eg, context switches cost 2 microseconds, and only let program run 10 microseconds => 20% context switching
10ms time slice, gives good ratio of context switch time and running time,
wont notice lag of key in char onto screen

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

when context switch, have alot of L1 and L2 cache data of old program, thrashes these caches

Context Switch Example

Stepping through a `read()` System Call



1. USER Mode running program then eventually call int 0x80
   read() %eax will have 3

2. switch to kernel mode — Direct memory access (some address in physical memory)
   tell device it needs to transfer data, gives responsibility to device controller

3. Initiate Context Switch — Preserve Process A state, restore Process B state

4. Switch to USER Mode

5. Transfer Control back to Process B — eventually the device finishes

6. Raise interrupt — causes machine to go into kernel mode

7. Process A becomes runnable

8. OS decide to context switch — preserves B, restore A