CS 354 - Machine Organization & Programming
Tuesday November 21, 2023

Thanksgiving Break: There is no TA Consulting or Peer Mentoring, from 4pm on Wednesday Nov 22 through the weekend until Sunday Nov 26.

Deb will still have regular schedule of office hours this week and next week.

Homework hw6: DUE on or before Monday Nov 20

Homework hw7: DUE on or before Monday Nov 27

Project p5: DUE on or before Friday Nov 24 (do before Wed Nov 23)

Project p6: Assigned soon and Due on last day of classes.

Learning Objectives

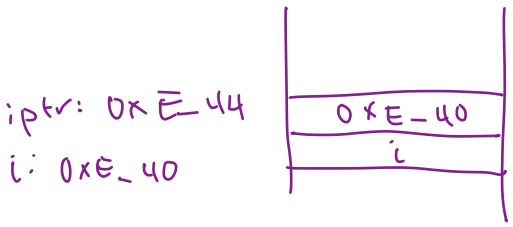
- Understand when and how to use function pointers for selecting which function at runtime
- Identify when buffer overflow occurs and be able to eliminate the chance for buffer overflow
- Identify exceptional control flow in C programs
- Understand the default behavior and to define new behaviors for exceptional events
- Trace the control flow that occurs when an exception occurs.
- Name and describe four categories of Exceptions in C.

This Week	**Next Week**
Pointers Function Pointers Buffer Overflow & Stack Smashing Flow of Execution Exceptional Events Kinds of ExceptionsTransferring Control via Exception Table THANKSGIVING BREAK	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 *`

pointer type is used by compiler to det. scale factor used in ASM

pointer value `0x2A300F87`, `0x00000000` (NULL)

addr used w/ addr mode to specify eff. addr in ASM

address of `&i`

becomes a leal instr

dereferencing `*iptr`

becomes a mov instr

Recall Casting in C

implicit cast from (void *) to (int *)

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

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

explicit cast of p to (char *) - changes S.F. to 1

- *Casting changes* the scale factor used not the ptr values

Function Pointers

What? A *function pointer*

- a ptr to code
- stores addr of 1st instr of func

Why?

enables functions to be

- passed and returned from other func
- store func ptrs in arrays
  - faster switch logic
  - jump table

How?

```
int func(int x) { ...}      // 1. impl func

int (*fptr)(int);           // 2. decl func ptr

fptr = func;                // 3. assign ptr to func

int x = fptr(11);           // 4. use ptr as func
```

Example

```
#include <stdio.h>

void add     (int x, int y) { printf("%d + %d = %d\n", x, y, x+y); }
// 1.
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() {                                                  // 3.
// 2.   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;
// 4.  fptr_arr[choice](i, j);
     return 0;
}
```

Buffer Overflow & Stack Smashing

Bounds Checking — C doesn't do it, programmers must

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

→ What happens when you execute the code?

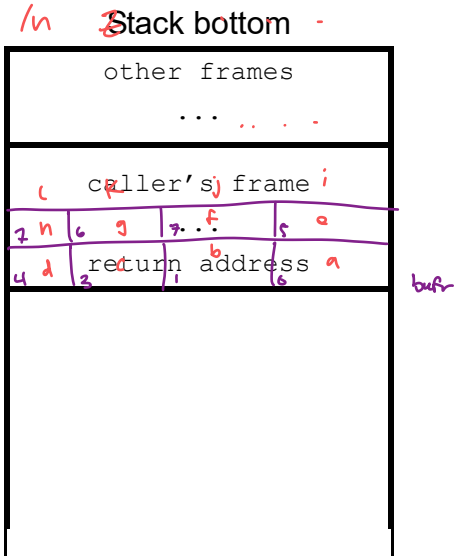
junk seg fault worse?

❋ *The lack of bounds checking array accesses* is a known C vulnerability

Buffer Overflow

- when we exceed bounds of arrays
- dangerous for SAA

```
void echo() {
   char bufr[8];
   gets(bufr);   // should use fgets
   puts(bufr);
}
```



❋ *Buffer overflow can overwrite* data outside buffer

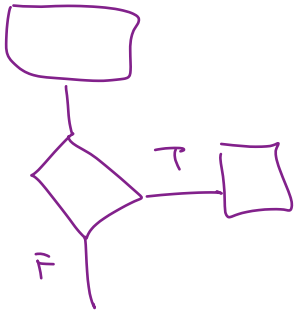
❋ *It can also overwrite* the "state" of exec

Stack Smashing

1. Get "exploit code" in enter input crafted to be mach. inst
2. Get "exploit code" to run overwrite ret addr
3. Cover your tracks restore stack so exec. continue as expected

❋ *In 1988 the Morris Worm* brought down Internet

Flow of Execution



What?

control transfer transition from one inst to next

control flow a sequence of control transitions

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

  sequential

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

  selection, repitition, func calls, return

Exceptional Control Flow

logical control flow normal execution that follows prog logic

exceptional control flow special exec. that enables system to react unusual / urgent anomolous events

event a change in processor state that may/may not be related to curr. inst

processor state processor's intenal mem. elements

- registers, CC / flags, signals

Some Uses of Exceptions

process
- ask for kernel svc
- share info w/ other proc
- send & receive signals

OS
- comm. w/ processes and hardware
- switch execution among proceses

hardware
- deal w/ mem pages - "page faults"
- indicate device status - ready, error, end

Exceptional Events

What? An *exception*

- is event that side-steps usual logical flow
- can originate from HW or SW
- an indirect func call that abruptly change flow of exec

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

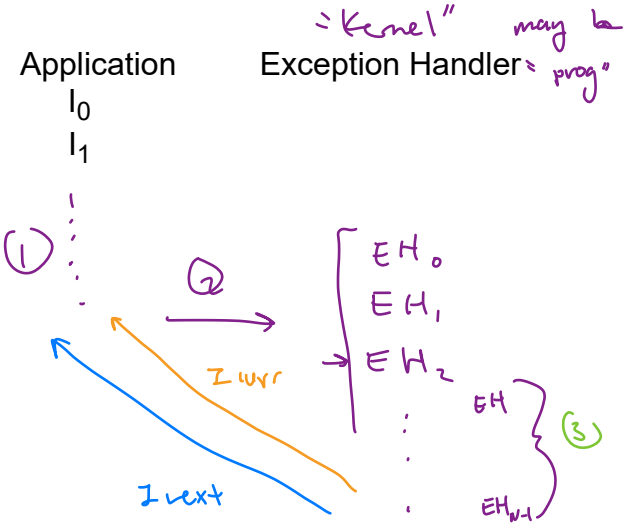
asynchronous event unrelated to current instr

synchronous event related to curr instr

General Exceptional Control Flow

0. normal flow

1. exception event occurs
2. control transfers to approp. exception handlers
3. run approp. exception handler
4. return control to :
   a. I curr (page fault)
   b. I next (file I/O)
   c. OS - abort (seg fault?)



Kinds of Exceptions

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

1. Interrupt - enables device to signal needs attn.
signal from external device
asynchronous
returns to Inext

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

vs. polling
↑ SW periodically check device

2. Trap - enables proc to interact w/ OS
intentional exception
synchronous
returns to Inext

How? Generally:
1. proc indicates need for OS svc
`int` interrupt inst.
2. transfer control to the OS system call handler which exec. requested svc
3. transfer control back to process's next instruction

3. Fault - handle "prob" w/ current inst
potentially recoverable error
synchronous
might return to Icurr and re-execute it
- page fault
- seg fault - default crash

4. Abort! - cleanly end a process
nonrecoverable fatal errors
synchronous
doesn't return