outlineL23-w12T-student

Saturday, December 3, 2022 2:09 PM

outlineL23-w12T-stu...

CS 354 - Machine Organization & Programming
Tuesday November 22, 2022

Homework hw6: DUE on or before Monday Nov 21
Homework hw7: DUE on or before Monday Dec 5
Project p5: DUE on or before Friday Dec 2nd (do before Wed Nov 23)
Project p6: Assigned soon and Due on last day of classes.

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	**Next Week**
finish w11 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 *`

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

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);

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

- *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 instruction of a function

Why?

enables functions to be

- passed to and returned from functions
- stored in arrays, e.g., faster switch logic - jump tables

How?

Return type (fptr) (params)

```
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
```

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;
}
```

```
Enter: [0-add, 1-subtract, 2-multiply]
2
22 * 11 = 242
```


Buffer Overflow & Stack Smashing

Bounds Checking

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

→ What happens when you execute the code?
intermittent error - might print junk, or worse, might crash - seg fault

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

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

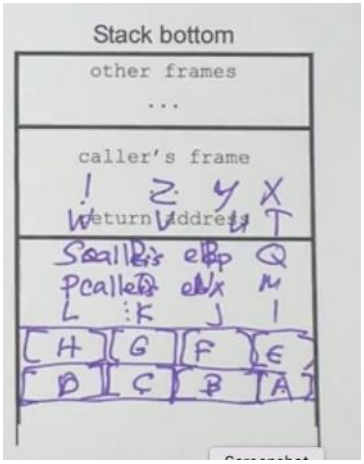
Buffer Overflow

- is exceeding the bounds of an array
- is particularly dangerous with stack allocated arrays

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

What if user inputted alphabet?

Use fgets



❋ *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 Transition from one inst. To another

control flow 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?

Conditional execution, repetition -> selection Function calls / returns as well

Exceptional Control Flow

logical control flow

Normal/ "expected" execution

exceptional control flow

"special" execution (unusual/urgent/anomalies)

event

Change in processor state may or may not be related to

event

Change in processor state may or may not be related to Current instruction

processor state

Internal processor storage
(registers, flags, signals) etc

Some Uses of Exceptions

By process to ask for kernel services

To share info between processes
To send and receive messages (these are both in p6)

By OS to communicate with running processes and hardware

To switch executions among processes
To deal with memory pages

- hardware — Indicates device status

Exceptional Events

What? An *exception*

- An event that sidesteps the logical flow
- Can originate from hardware or software
- An indirect call that abruptly changes flow of execution

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

asynchronous — From event that is unrelated to current instruction

synchronous — Resulting from a current instruction

General Exceptional Control Flow

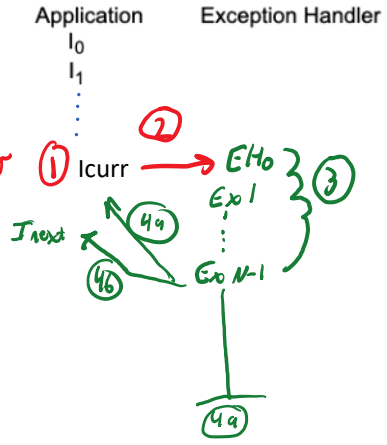
0. normal flow

1. Exception event occurs

2. transfer control to exception handler

3. appropriate exception handler runs

4. Return control to:
   a. Icurr (page fault)
   b. Inext (file I/O)
   c. OS - Aborts (seg-fault)
      default is abort



Kinds of Exceptions

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

1. Interrupt enables devices to signal need for attn

signal from external device
asynchronous
returns to Inext

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

vs. polling Periodically checking devices

2. Trap enables processes to interact with OS

intentional exception
synchronous
returns to Inext

"Sys calls"

How? Generally:
1. Prog. indicates need for O.S. service

   `int` interrupt instruction
2. transfer control to the OS system call handler
3. transfer control back to process's next instruction

3. Fault handle probs with curr inst / page fault / seg fault

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

4. Abort Cleanly ends a process

nonrecoverable fatal errors e.g. hardware error
synchronous
doesn't retur