Exceptional Control Flow & OH for Midterm

18600: Introduction to Computer Systems
Recitation 9: Tuesday, October 25th, 2016

Agenda

- Exceptional Control Flow
- Processes
- Signals
- OH for Midterm

Exceptional Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

  Both react to changes in ***program state***

- Insufficient for a useful system:
  Difficult to react to changes in *system state*
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - user hits Ctrl-C at the keyboard
  - System timer expires

- System needs mechanisms for “exceptional control flow”

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - Handler returns to "next" instruction
- Examples:
  - I/O interrupts
    - hitting Ctrl-C at the keyboard
    - arrival of a packet from a network
    - arrival of data from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting Ctrl-Alt-Delete on a PC

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - ***Aborts***
    - unintentional and unrecoverable
    - Examples: parity error, machine check
    - Aborts current program

Processes

- What is a *program*?
  - A bunch of data and instructions stored in an executable binary file
  - Written according to a specification that tells users what it is supposed to do
  - Stateless since binary file is static

Processes

- Definition: A *process* is an instance of a running program.
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory
    - Gives the running program a ***state***
- How are these Illusions maintained?
  - Process executions interleaved (multitasking) or run on separate cores
  - Address spaces managed by virtual memory system
    - Just know that this exists for now; we'll talk about it soon

Processes

- Four basic States
  - Running
    - Executing instructions on the CPU
    - Number bounded by number of CPU cores
  - Runnable
    - Waiting to be running
  - Blocked
    - Waiting for an event, maybe input from STDIN
    - Not runnable
  - Zombie
    - Terminated, not yet reaped

Processes

- Four basic process control function families:
  - fork()
  - exec()
    - And other variants such as execve()
  - exit()
  - wait()
    - And variants like waitpid()
- Standard on all UNIX-based systems
- Don't be confused:
  Fork(), **E**xit(), **W**ait() are all wrappers provided by CS:APP

Process Examples

```
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{

   printf("Parent!\n");
}
```

- What are the possible output (assuming fork succeeds) ?
  - `Child!`
    `Parent!`
  - `Parent!`
    `Child!`
- How to get the child to always print first?

Process Examples

```
int status;
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf("Child!\n");

   exit(0);
}
else{
   waitpid(child_pid, &status, 0);

   printf("Parent!\n");
}
```

- Waits till the child has terminated.
  Parent can inspect exit status of child using 'status'
  - WEXITSTATUS(status)

- Output always:
  `Child!`
  `Parent!`

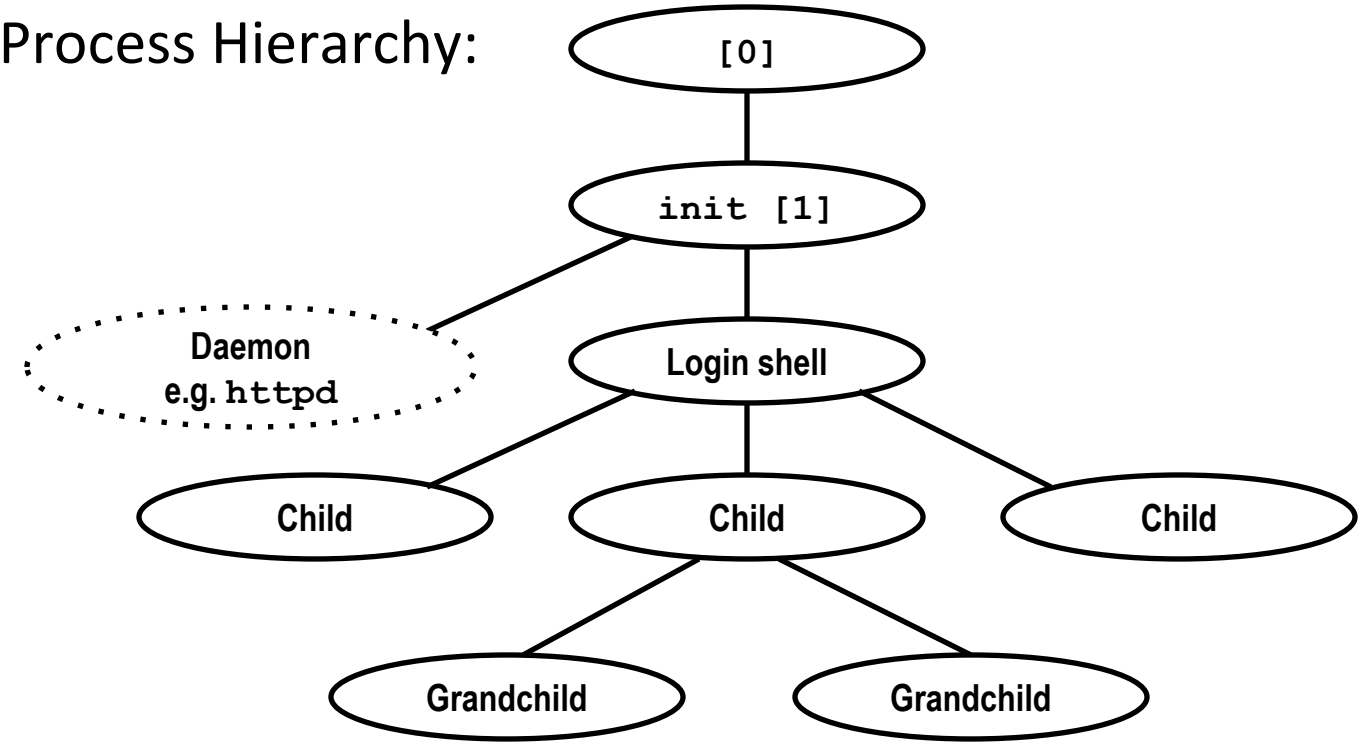
Process Examples

```
int status;
pid_t child_pid = fork();
char* argv[] = {“/bin/ls”, “-l”, NULL};
char* env[] = {…, NULL};

if (child_pid == 0){
   /* only child comes here */

   execve(“/bin/ls”, argv, env);

   /* will child reach here? */
}
else{
   waitpid(child_pid, &status, 0);

   … parent continue execution…
}
```

- An example of something useful.
- Why is the first arg “/bin/ls”?
- Will child reach here?

Process Examples

- Unix Process Hierarchy:



Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
  - akin to exceptions and interrupts (asynchronous)
  - sent from the kernel (sometimes at the request of another process) to a process
  - signal type is identified by small integer ID's (1-30)
  - only information in a signal is its ID and the fact that it arrived

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., ctl-c from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

Signals

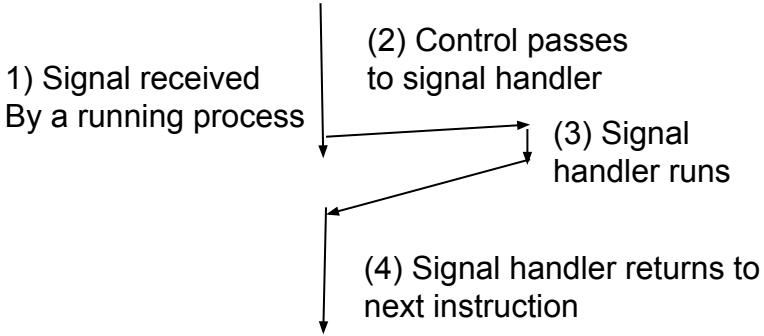
- A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
- Blocking signals
  - Sometimes code needs to run through a section that can't be interrupted
  - Implemented with `sigprocmask()`
- Waiting for signals
  - Sometimes, we want to pause execution until we get a specific signal
  - Implemented with `sigsuspend()`
- Can't modify behavior of SIGKILL and SIGSTOP

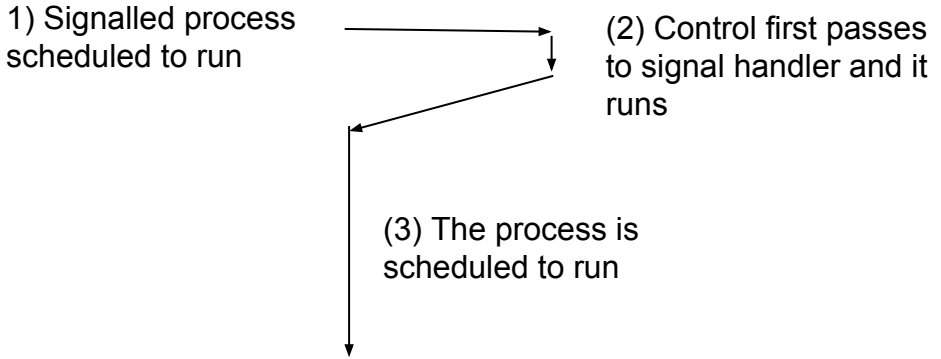
Signals

- Signal handlers
  - Can be installed to run when a signal is received
  - The form is `void handler(int signum){ … }`
  - **Separate** flow of control in the same process
  - Resumes normal flow of control upon returning
  - Can be called **anytime** when the appropriate signal is fired

Signal Handling

- Running Process:
  - Receipt of a signal triggers a control transfer to a signal handler
  - After it finishes processing, the handler returns control to the interrupted program
- Runnable Process:
  - When the process is next scheduled, the control is first transferred to the signal handler
  - After it finishes processing, the handler returns control to the program





Shell Lab

- Shell Lab is out!
- Due Monday, November 7th
- Read the code we've given you
  - There's a lot of stuff you don't need to write yourself; we gave you quite a few helper functions
  - It's a good example of the code we expect from you!
- Don't be afraid to write your own helper functions; this is not a simple assignment

Shell Lab

- Read man pages. You may find the following functions helpful:
  - `sigemptyset()`
  - `sigaddset()`
  - `sigprocmask()`
  - `sigsuspend()`
  - `waitpid()`
  - `open()`
  - `dup2()`
  - `setpgid()`
  - `kill()`
- Please read the man pages thoroughly to understand what each function does
- Please do not use sleep() to solve synchronization issues.

Shell Lab

- Hazards
  - Race conditions
    - Hard to debug so start early (and think carefully)
  - Reaping zombies
    - Race conditions
    - Handling signals correctly
  - Waiting for foreground job
    - Think carefully about what the right way to do this is

Shell Lab Testing

- Run your shell
  - This is the fun part!
- tshref
  - How should the shell behave?
- runtrace
  - Each trace tests one feature.

Midterm Review

Cache

- Types of caches
  - Direct Mapped (multiple sets, one cache line per set)
  - Set associative (multiple sets, multiple lines per set)
  - Fully associative (one set, multiple lines in that set)

- Types of locality
  - Spatial locality (Access set of adjacent elements successively)
  - Temporal locality (Access same set of elements iteratively)

- Miss rate = $\frac{\#\text{misses}}{\#\text{accesses}}$.

- The better the locality, lower the miss rate

Cache - Hit rate and Locality analysis

Example question:

1. Direct mapped 16 byte data cache with two cache lines.
2. Float requires 4 bytes.
3. Cache is loaded such that X is cache aligned: X[0] is loaded into the beginning of the first cache line

```
float X[8], t = 0;
for(int j = 0; j < 2; j++)
  for(int i = 0; i < 8; i++)
    t += X[i];
```

- Miss rate?
- Does this code exhibit locality? What kind of locality?

Cache - Hit rate and Locality analysis

Example question:

1. Direct mapped 16 byte data cache with two cache lines.
2. Float requires 4 bytes.
3. Cache is loaded such that X is cache aligned: X[0] is loaded into the beginning of the first cache line

```
float X[8], t = 0;
for(int j = 0; j < 2; j++)
  for(int i = 0; i < 8; i++)
    t += X[i];
```

- Miss rate? 50%
- Does this code exhibit locality? What kind of locality? Yes, Spatial locality

Cache - Hit rate and Locality analysis

Now try this:

```
float X[8], float t = 0;
for(i = 0; i < 2; i++)
  for(k = 0; k < 2; k++)
    for(j = 0; j < 4; j++)
      t += X[j + i * 4];
```

- Miss rate?
- Does this code exhibit locality? What kind of locality?

Cache - Hit rate and Locality analysis

Now try this:

```
float X[8], float t = 0;
for(i = 0; i < 2; i++)
  for(k = 0; k < 2; k++)
    for(j = 0; j < 4; j++)
      t += X[j + i * 4];
```

- Miss rate? 25%
- Does this code exhibit locality? What kind of locality? Yes. Spatial, Temporal locality

Example of forwarding from Recitation 6

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Case 2:
  - Which is the earliest stage at which value of r3 is ready ?
  - Which is the latest by which I3 MUST receive updated r3 ?
- Case 3:
  - Which is the earliest stage at which value of r2 is ready ?
  - Which is the latest by which I3 MUST receive updated r2 ?

Example of forwarding from Recitation 6

- Consider the following example

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

- Case 2:
  - Which is the earliest stage at which value of r3 is ready ? MEMORY
  - Which is the latest by which I3 MUST receive updated r3 ? MEMORY
  - Forward from MEMORY stage of I2 to MEMORY stage of I3
- Case 3:
  - Which is the earliest stage at which value of r2 is ready ? EXECUTE
  - Which is the latest by which I3 MUST receive updated r2 ? EXECUTE
  - Forward from EXECUTE stage of I1 to EXECUTE stage of I3
  - Also not late to forward from MEMORY stage of I1 to EXECUTE stage of I3

How the Pipeline looks like after forwarding

```
I1: add r1, r2
I2: mrmovq d(r2), r3
I3: rmmovq r3, d(r2)
```

	Src, Dst	1	2	3	4	5	6	7	8	9
ADD	R1, R2	IF	ID	**EX**	**MEM**	WB				
MRMOVQ	d(R2), R3		IF	ID	**EX**	**MEM**	WB			
RMMOVQ	R3, d(R2)			IF	ID	**EX**	**MEM**	WB		

- The first forwarding is for value of R2 from EX_{add} to EX_{mrmovq}
- The second forwarding is for value of R2 from $MEMORY_{add}$ to EX_{rmmovq}
- The third forwarding is for value of R3 from MEM_{mrmovq} to MEM_{rmmovq}

Superscalar Processing: Important Concepts

- Dynamic branch prediction to reduce control hazards
- Avoid false data hazards using register allocation & renaming
- Reduce pipeline hazards using load forwarding & bypassing
- Dynamic scheduling using reservations stations, reorder buffer

Exploit Illustration

Strcpy Vulnerability

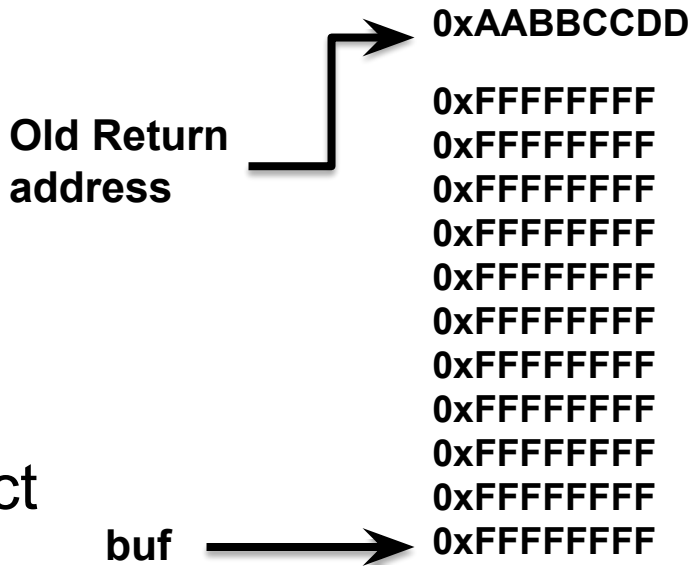
```
int main(int argc, char *argv[]){
    foo(argv[1]);
    ...
}

void foo(char *input){
    char buf[32];
    ...
    strcpy (buf, input);
    return;
}
```

What is the potential issue with this program?

Buffer Overflows

- Exploit *strcpy vulnerability* to overwrite important info on stack
- When this function returns, where will it begin executing?
  - Recall

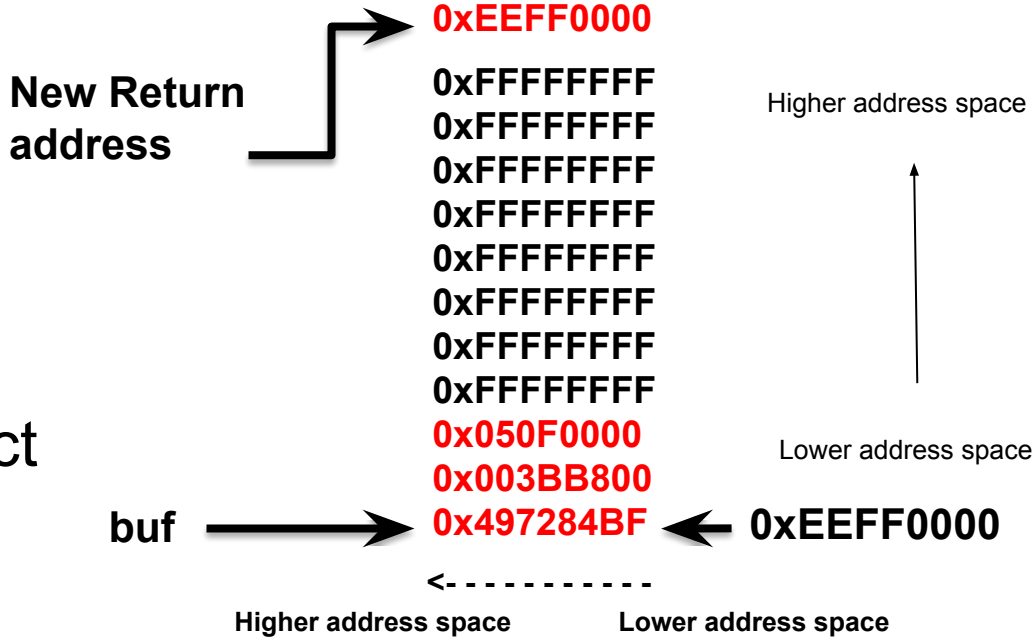
    ```
    ret:pop %rip
    ```
- What if we want to inject new code to execute?



Buffer Overflows

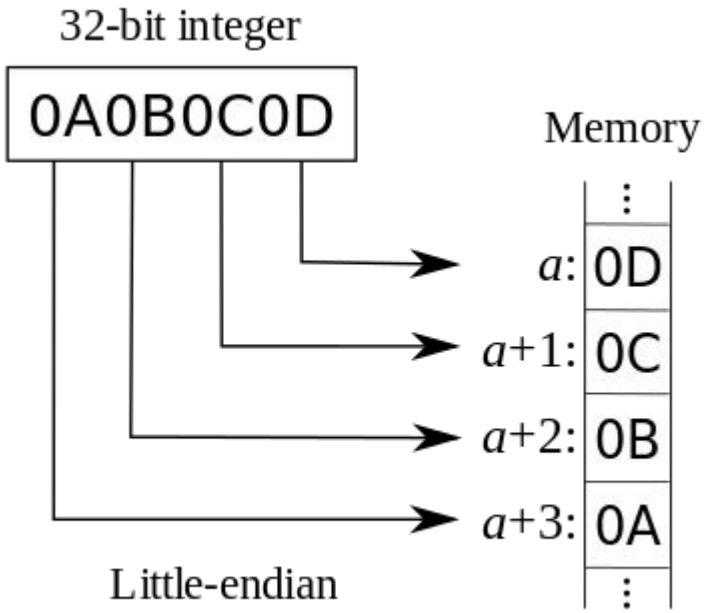
- Exploit *strcpy vulnerability* to overwrite important info on stack
- When this function returns, where will it begin executing?
  - Recall

    `ret:pop %rip`
- What if we want to inject new code to execute?



Endianness

- When printing a value, the byte at the highest memory address is printed first (**high** to **low** bytes)
- When exploiting strcpy or other string operations, you are writing from **low** to **high** bytes
  - Why we tell you to reverse bytes when writing a value (like an address) with a string operation
  - Does not mean actual value is changed



Few tips for buffer overflow questions

- If we print the value at address 0xEEFF0000, what is the output ?

Few tips for buffer overflow questions

- If we print the value at address 0xEEFF0000, what is the output ? **497284BF**
- How does the ‘printf’ statement know what to print and how many values to print ?

Few tips for buffer overflow questions

- If we print the value at address 0xEEFF0000, what is the output ? **497284BF**
- How does the ‘printf’ statement know what to print and how many values to print ?
  - Arguments to ‘printf’ statement include:
    - The address of the format specifier(Eg: “%x”/”%d”)
    - The value to be printed

Few tips for buffer overflow questions

- If we print the value at address 0xEEFF0000, what is the output ? **497284BF**
- How does the ‘printf’ statement know what to print and how many values to print ?
  - Arguments to ‘printf’ statement include:
    - The address of the format specifier(Eg: “%x”/”%d”)
    - The value to be printed
- For buffer overflow attacks, the source string will be passed on the command line:
  - ./program mystring ←-- when main starts, mystring address will be located in argv[1]
  - Recall: argv address contained in %rsi

Open Office Hour