Exceptional Control Flow

15-213: Introduction to Computer Systems
Recitation 9: Monday, March 18\textsuperscript{th}, 2013

Ian Hartwig
Section F
News

- Midterms are available for pickup at the ECE Course Hub
  - HH 1112
  - Open 9:00am - 5:00pm Monday - Friday

- Shell lab due next Thursday at 11:59pm
Agenda

- Midterm Wrap-Up
- Exceptional Control Flow
- Processes
- Process Examples
- Signals
- Shell lab
Midterm Wrap-Up

- Mid-semester/Midterm grades aren’t curved or adjusted.
- Cutoffs are standard A >= 90, B >= 80, C >= 80, etc.
Midterm Wrap-Up

- Mean: 66
- Median: 67
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”
Exceptional Control Flow

What are the different types of ECF?

- Exceptions
- Process Context Switch
- Signals
- Nonlocal jumps
Exceptional Control Flow

- Exceptions are either synchronous or asynchronous:
  - Asynchronous
    - I/O interrupts
    - Reset interrupts
  - Synchronous
    - Traps
    - Faults
    - Aborts
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
Agenda

- Exceptional Control Flow
- Processes and Signals
- Shell lab
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(), ⚠️Exit(), ⚠️Wait() are all wrappers provided by CS:APP
Processes

**int fork(void)**
- creates a new process (child process) that is identical to the calling process (parent process)
- OS creates an exact duplicate of parent’s state:
  - Virtual address space (memory), including heap and stack
  - Registers, except for the return value (%eax/%rax)
  - File descriptors but files are shared
- Result \(\rightarrow\) **Equal but separate state**

- Fork is interesting (and often confusing) because it is called *once* but returns *twice*
Processes

- int fork(void)
  - returns 0 to the child process
  - returns child’s `pid` (process id) to the parent process
  - Usually used like:

```c
pid_t pid = fork();

if (pid == 0) {
    // pid is 0 so we can detect child
    printf("hello from child\n");
}

else {
    // pid = child’s assigned pid
    printf("hello from parent\n");
}
```
Processes

- **int exec()**
  - Replaces the current process’s state and context
    - But keeps PID, open files, and signal context
  - Provides a way to load and run another program
    - Replaces the current running memory image with that of new program
      - Set up stack with arguments and environment variables
      - Start execution at the entry point
  - Never returns on successful execution
  - The newly loaded program’s perspective: as if the previous program has not been run before
  - More useful variant is **int execve()**
  - More information? man 3 exec
Processes

- **void exit(int status)**
  - Normally return with status 0 (other numbers indicate an error)
  - Terminates the current process
  - OS frees resources such as heap memory and open file descriptors and so on...
  - Reduce to a zombie state
    - Must wait to be reaped by the parent process (or the init process if the parent died)
    - Signal is sent to the parent process notifying of death
    - Reaper can inspect the exit status
Processes

- `int wait(int *child_status)`
  - suspends current process until one of its children terminates
  - return value is the pid of the child process that terminated
    - When `wait` returns a pid > 0, child process has been reaped
    - All child resources freed
  - if `child_status` != NULL, then the object it points to will be set to a status indicating why the child process terminated
  - More useful variant is `int waitpid()`
  - For details: man 2 `wait`
Process Examples

What are the possible output (assuming fork succeeds)?

- Child!
  Parent!
- Parent!
  Child!

How to get the child to always print first?

```c
pid_t child_pid = fork();

if (child_pid == 0){
    /* only child comes here */
    printf("Child!\n");
    exit(0);
} else{
    printf("Parent!\n");
}
```
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 til 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:

```
[0]  
  \______________________________________________
  |                                         |
  v                                         v
init [1]  
  \______________________________
  |                           |
  v                           v
Login shell  
  \______________________________
  |                           |
  v                           v
Daemon e.g. `httpd`  
  \______________________________
  |                           |
  v                           v
Child  
  \______________________________
  |                           |
  v                           v
Child  
  \______________________________
  |                           |
  v                           v
Grandchild  
  \______________________________
  |                           |
  v                           v
Grandchild
```
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

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., ctl-c from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Signals

- Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process.

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as Ctrl-C (SIGINT), divide-by-zero (SIGFPE), or the termination of a child process (SIGCHLD).
  - Another program called the `kill()` function.
  - The user used a kill utility.
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.

- Receiving a signal is non-queueing
  - There is only one bit in the context per signal
  - Receiving 1 or 300 SIGINTs looks the same to the process

- Signals are received at a context switch

- Three possible ways to react:
  - *Ignore* the signal (do nothing)
  - *Terminate* the process (with optional core dump)
  - *Catch* the signal by executing a user-level function called *signal handler*
    - Akin to a hardware exception handler being called in response to an asynchronous interrupt
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 Examples

Signal delivered

Process A

\[ I_{\text{curr}} \]

user code (main)

kernel code

Signal received

\[ I_{\text{next}} \]

user code (main)

user code (handler)

kernel code

Process B

\} context switch

\} context switch
Signal Examples

- Every process belongs to exactly one process group
- Process groups can be used to distribute signals easily
- A forked process becomes a member of the parent’s process group

```
getpgid()
Return process group of current process

setpgid()
Change process group of a process
```
Signal Examples

// sigchld handler installed

pid_t child_pid = fork();

if (child_pid == 0){
  /* child comes here */
  execve(......);
}
else{
  add_job(child_pid);
}

Does add_job or remove_job() come first?

Where can we block signals in this code to guarantee correct execution?

void sigchld_handler(int signum)
{
  int status;

  pid_t child_pid =
    waitpid(-1, &status, WNOHANG);

  if (WIFEXITED(status))
    remove_job(child_pid);
}
Signal Examples

```c
// sigchld handler installed  void sigchld_handler(int signum)

pid_t child_pid = fork();
if (child_pid == 0){ /* child comes here */
    execve(......);
} else{
    add_job(child_pid);
}
```

- Does `add_job` or `remove_job()` come first?
- Where can we block signals in this code to guarantee correct execution?

Block SIGCHLD
```c
int status;
if (child_pid == 0){
    waitpid(-1, &status, WNOHANG);
    execve(......);
} else{
    remove_job(child_pid);
}
```

Unblock SIGCHLD

Unblock SIGCHLD
Shell Lab

- 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 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
Q & A

- For your listening enjoyment:
  http://www.monzy.com/intro/killdashnine_lyrics.html