CS24: Introduction to Computing Systems

Spring 2015

Lecture 19

Last Time

- Introduced UNIX signals
  - A kernel facility that provides user-mode exceptional control flow
- Allows many hardware-level exceptions to be exposed to application processes
  - Timer events (**`SIGALRM`**), invalid memory access (**`SIGSEGV`**), illegal instruction execution (**`SIGILL`**), etc.
- Also allows processes to send signals to each other
  - e.g. terminal sends **`SIGINT`** (Ctrl-C) to your program
  - e.g. kernel can send **`SIGKILL`** to a runaway process

Reentrant Functions

- A signal handler can interrupt *any other code* in the program
  - …*including function calls that are in progress!*
- Signal handlers must only use <u>reentrant functions</u>
  - Functions that can be invoked *multiple times concurrently*, without causing errors
    - i.e. multiple overlapping logical flows through the function
  - Frequently, code in a signal handler will interrupt code in the main program that is using the exact same functions
- Example: **`malloc()`** is <u>not</u> reentrant!
  - Updates large, complex data structures within the heap
  - Two calls to **`malloc()`** can easily stomp on each other!
  - Must not use **`malloc()`** within a signal handler!
    (Or, any other function that calls **`malloc()`**!)

Reentrant Functions and State

- A reentrant function must not manage state across multiple function calls
  - Concurrent invocations will cause state to be corrupted
  - No global state, and no static local variables!
- Example:

```
void do_task() {
  /* Variable remains across function calls! */
  static int count = 0;

  ...  /* Do some important task. */
  count++;
  if (count > 100) {
    count = 0;
    periodic_cleanup();
  }
}
```

  - Static local variables are retained across multiple function calls (like a global variable visible only within the function)

Reentrant Functions and State (2)

- Concurrent invocations of **`do_task()`** will not update **`count`** properly!

```
void do_task() {
  /* Variable remains across function calls! */
  static int count = 0;

  ...  /* Do some important task. */
  count++;
  if (count > 100) {
    count = 0;
    periodic_cleanup();
  }
}
```

Logical Control Flow	**Signal Handler**
Read count into register	
	Read count into register
(…interrupted…)	Add 1 to register
	Write reg back to count
Add 1 to register	
Write reg back to count	

- <u>Not</u> reentrant; *must not* be called from a signal handler

Reentrant Functions and Mutexes

- Can't fix this problem with a mutex or semaphore
  - First function invocation locks the mutex before accessing the persistent state, but is then interrupted
    - *…but it still holds the lock!*
  - A second, concurrent function invocation interrupts the first one
    - Can't acquire the lock because it's already held!
    - Second invocation will be blocked by the first one!
  - Entire system will grind to a halt. Whee.
- Locks can be used in a *non-blocking* manner
  - e.g. "try to acquire the lock, but return immediately with a failure code if someone else holds the lock"
  - *(May limit the usefulness of your signal handler…)*

Guaranteed Reentrant Functions

- A signal handler can only use reentrant functions
- Ideally, the handler will also be reentrant
  - Particularly in cases where one handler handles multiple signals
- Specific UNIX functions are guaranteed to be reentrant by the standard. For example:
  - **`alarm()`**, **`pause()`**, **`signal()`**
  - **`mkdir()`**, **`chdir()`**, **`rmdir()`**, **`chmod()`**, **`chown()`**
  - **`open()`**, **`close()`**, **`read()`**, **`write()`**
  - **`fork()`**, **`execve()`**, **`exit()`**, **`kill()`**, **`wait()`**
- Many others are not guaranteed to be reentrant
  - e.g. **`malloc()`**, **`free()`**, **`printf()`**, etc.
  - Must avoid using these functions in signal handlers!

Using Non-Reentrant Functions

- If non-reentrant functions will <u>never</u> be called concurrently, can use them in signal handlers…
  - …but it's *extremely* difficult to guarantee this, especially in context of maintenance and upgrades!

- **Don't tempt fate! ☺ Program defensively.**

Signal Handler Operations

- Usually, handlers perform very simple operations
  - Avoids the dangers of non-reentrant code!
- Set a flag to record that signal occurred, then return
  - Main loop checks the flag every iteration
  - When flag becomes set, main loop handles the signal then
  - Main loop code can do whatever it wants, since signal handler won't be doing anything complicated
- Or, simply restart running the program at an appropriate location
  - e.g. an error signal handler can restart a server at the beginning of the program
  - Use something like **`setjmp()`** and **`longjmp()`**… but there's a caveat…

Pending Signals

- For each process, the kernel manages two bookkeeping variables for signal handling
- **`pending`** – a bit-vector of signals that are currently pending for the process
  - These signals have been sent to the process, but haven't yet been received or handled by the process
- Each kind of signal has <u>one bit</u> assigned to it
  - Multiple signals of a particular type sent to a process will not necessarily all be received!
  - If a particular type of signal is already pending, and then is sent again, the second signal is <u>dropped</u>!
    - The **`pending`** flag for that signal is already set, so it can't be sent again

BLOCKED SIGNALS

- The kernel also keeps a **`blocked`** bit-vector for each process
  - Again, each type of signal has a bit assigned to it
  - If a particular type of signal is blocked then it won't be delivered to the process
  - Can definitely have a pending signal that is also blocked!
- When the kernel calls a signal handler on a process, that type of signal is automatically blocked
  - Generally, signal handlers don't need to worry about being interrupted by the same kind of signal again
- Example: a process with a **`SIGINT`** handler
  - First **`SIGINT`** received causes **`SIGINT`** handler to be called
  - Also causes **`SIGINT`** to become blocked for the process!
  - If another **`SIGINT`** occurs during handler execution, it is recorded in **`pending`** bit-vector, but it is not delivered!

Blocked Signals (2)

- When a signal handler returns, the blocked signal-type is automatically unblocked
  - When handler returns, signals of that type can begin being delivered again
  - In the case of a blocked pending **SIGINT**, it will subsequently be delivered to the process…
- Several functions for manipulating these signal bit-vectors

```
int sigpending(sigset_t *set)
```

  - Returns current set of pending signals for the process

```
int sigprocmask(int how,
    const sigset_t *set, sigset_t *oldset)
```

  - Manipulates the set of blocked signals for the process

  - Several other functions too! See CS:APP §8.5.6

Signals and `setjmp()/longjmp()`

- When a signal handler is called, that signal type is automatically blocked for the process
- When the handler returns, the signal type is unblocked again
- What happens if we **`longjmp()`** from inside a signal handler?
- **`setjmp()`** records:
  - Execution state and registers, specifically the caller's instruction pointer and the stack pointer
  - *Does not record any other data beyond this!*
- If we **`longjmp()`** from signal handler, it will not unblock the blocked signal!

SIGNALS AND `setjmp()/longjmp()` (2)

- Example code:

```
static jmp_buf restart_env;

void restart_handler(int sig) {
    /* Just restart the program! */
    longjmp(restart_env, 1);
}

int main(int argc, char **argv) {
  signal(SIGHUP, restart_handler); /* SIGHUP restarts */

  if (setjmp(restart_env))  /* Returns 1 if restarting! */
    fprintf(stderr, "Signal received, restarting!");

  load_config(...);
  while (!quit) {
    ...  /* Handle incoming requests, or something. */
  }
}
```

This code only handles one **SIGHUP**! Signal is blocked when first **SIGHUP** handled, and is never delivered again.

`sigsetjmp()` AND `siglongjmp()`

- When using long-jumps from signal handlers, must use <u>**`sigsetjmp()`**</u> and <u>**`siglongjmp()`**</u>
  - Nearly identical to **`setjmp()`** and **`longjmp()`**…
- **`sigsetjmp()`** also records blocked-signal mask

  ```
  int sigsetjmp(sigjmp_buf env, int savesigs)
  ```
  - If **`savesigs`** is true, blocked-signal state is saved
- **`siglongjmp()`** restores the blocked-signal mask

  ```
  void siglongjmp(sigjmp_buf env, int val)
  ```
  - If corresponding **`sigsetjmp()`** set **`savesigs`** to true, then blocked-signal state is also restored here

Signals and Long-Jumps

- In general, long-jumps from signal handlers are useful in a very limited set of circumstances
  - Must ensure that it's actually safe to *never return* to the code that was interrupted!
  - e.g. don't leave files in a corrupted state; don't interrupt **`malloc()`** or other non-atomic operations
- Setting a flag is generally the safest approach
  - Signal handler sets a flag and then returns
  - Main loop checks the flag regularly, and responds accordingly
- Can only long-jump from a signal handler in certain circumstances
  - e.g. when interrupted operation is atomic, or when signal indicates that normal operation was aborted

Summary: Signals

- Signals share many common traits with hardware exception handling
  - A user-mode version of hardware exceptions
  - Signal handlers must be aware of reentrancy issues, just like hardware exception handlers
  - When a signal handler is invoked, that signal type is blocked until the handler returns
    - Very similar to hardware interrupts and `eflags` register
- Signals allow us to leverage exceptional control flow in user-mode programs
  - Enables powerful techniques in server programming
  - Are used in most widely-used server programs
    - Web-servers, email servers, DNS servers, databases, etc.

The Unix Process Model

- UNIX has a simple but powerful process model
- Every process has a context…
  - Kernel must be able to uniquely identify the context of each running process
- Each process has a unique "process ID" (PID)
  - `pid_t getpid()` returns the caller's process ID
  - `pid_t` is simply an integer
- Every process is started by some other process
  - All processes form a hierarchy
- Each process also has a parent process ID
  - `pid_t getppid()` returns PID of the parent process
  - Parent process must be a running process
    - (More on this in a moment…)

THE `init` PROCESS

- Process 1 is the **`init`** process
  - The ancestor of all processes on the UNIX system
  - Started as the last step of kernel boot sequence
- Responsible for starting various sets of processes to support different operating system “runlevels”
  - Each runlevel represents a set of services or capabilities provided by the system
- Use **`init`** to switch runlevels: **`init`** ***`runlevel`***
  - Runlevel 0 tells the system to shutdown
  - Runlevel 1 is single-user mode
  - Runlevel 2 is multi-user mode with limited networking
  - Runlevel 3 is full multi-user mode
  - Runlevel 5 starts X11 server for graphical logins
  - Runlevel 6 tells the system to restart

Process States

- Generally, processes are in one of these states:
- Running
  - The process is currently executing on the CPU, or is waiting to be executed
- Stopped
  - The process is suspended (e.g. by Ctrl-Z from keyboard), and will not be scheduled for execution until it is resumed
  - A process enters this state when it receives **`SIGSTOP`**, **`SIGTSTP`**, **`SIGTTIN`**, or **`SIGTTOU`** signals
    - **`SIGSTOP`** – stop signal not from the terminal (keyboard)
    - **`SIGTSTP`** – stop signal from terminal (i.e. Ctrl-Z from keyboard)
    - **`SIGTTIN`** – background process tries to read from terminal
    - **`SIGTTOU`** – background process tries to write to terminal
  - When process is resumed, changes back to Running state

Process States (2)

- Terminated
  - The process stops executing permanently
  - Occurs when:
    - The process calls **`exit()`** or returns from **`main()`**
    - The process receives a signal whose default action is to terminate
- A process can terminate by calling **`exit()`**
  - **`void exit(int status)`**
  - (Or, return an integer value from **`main()`** function)
  - The exit status of the process is recorded for retrieval by other processes
- **`exit()`** supports *exit-handlers*
  - Functions to perform tasks at process-termination
  - **`int atexit(void (*function)(void))`** registers an exit-handler for the process

PROCESS GROUPS

- Every process also belongs to a process group
  - Another **`pid_t`** value associated with each process
  - Signals can be sent to entire groups of processes

    `int killpg(pid_t pgrp, int sig);`
- Every process group has a leader
  - The leader's PID is equal to the group's PID
  - **`pid_t getpgid(pid_t pid)`**
    - Reports the process-group ID of the specified process
  - **`pid_t getpgrp()`**
    - Reports the process-group ID of the calling process
  - **`int setpgid(pid_t pid, pid_t pgid)`**
    - Sets the process-group ID of the specified process
  - **`pid_t setpgrp()`**
    - Sets the caller's process-group ID to its own PID

Starting a Process

- A process can start a child process with **`fork()`**
  - **`pid_t fork()`**
- **`fork()`** is called once, but returns twice!
  - In parent process, return-value of **`fork()`** is PID of newly spawned child process
    - No other way for the parent to find out the child's PID
  - In child process, return-value of **`fork()`** is 0
- Common usage pattern:

```
pid_t child;

child = fork();
if (child == 0) {
    ...  /* Do child-process stuff. */
}
else {
    ...  /* Do parent-process stuff. */
}
```

Starting a Process (2)

- The child process is an *identical duplicate* of the parent process
  - Main difference is that child process has a different process ID, and a different parent-process ID
  - Child is in same process-group as the parent process
- Parent and child processes have identical registers, memory, and stack contents
  - Separate copies, but identical contents
- Address-spaces of both processes are identical
  - Kernel provides illusion of exclusive access to memory
- Parent and child processes execute concurrently
  - Kernel also provides illusion of exclusive access to CPU
  - Logical control flows of parent and child proceed separately
  - Execution is interleaved as determined by the kernel

Starting a Process (3)

- Parent and child processes also have the same open files!
  - Child can also read/write any files that parent had open when the process was forked
    - e.g. standard input, standard output
  - Very important behavior – used to set up "pipes" between processes for inter-process communication

Simple `fork()` Example

```
#include <unistd.h>
#include <stdio.h>
int main() {
    pid_t child_pid;
    int x = 1;

    child_pid = fork();
    if (child_pid == 0) {  /* Child code. */
        x++;
        printf("Hello from child!   x = %d\n", x);
    } else {               /* Parent code. */
        x--;
        printf("Hello from parent!  x = %d\n", x);
    }

    return 0;
}
```

Simple `fork()` Example (2)

- Both parent and child process print to stdout…
- Compile and run the program:

```
[user@host:~]> gcc -Wall -o procs procs.c
[user@host:~]> ./procs
Hello from parent!  x = 0
Hello from child!   x = 2
[user@host:~]>
```

- Output from both processes is sent to stdout
  - Parent's stdout is sent to the console…
  - When child is spawned, stdout from parent is copied
- Both processes have identical memory contents
  - Both have variable x at the same memory address
  - Child sees x = 1, since x was 1 in parent process

Zombies!!!

- A child process doesn't immediately go away when it terminates
  - Child process terminates with some status value…
  - Parent process may need to find out the child's status
- A terminated child process is called a <u>zombie</u>
  - The process is dead, but it hasn't yet been reaped
  - (Seriously. That's the terminology.)
- Parent processes reap zombie children by calling:

  `pid_t wait(int *status)`
    - Waits for some child process to terminate

  `pid_t waitpid(pid_t pid, int *status, int options)`
    - Waits for a specific child process to terminate
    - Can also wait on children in a process-group, or all children
  - Both report an error if calling process has no children

Reaping Zombie Children

- Status value includes several important details
  - Did the child process terminate normally?
    - i.e. via a call to **`exit()`** or a return from **`main()`**
  - Did the child process terminate because of a signal?
    - e.g. **`SIGINT`** or **`SIGKILL`**
  - Is the child process actually stopped (suspended) instead of being terminated?
- Several macros to extract these details

  `WIFEXITED(status)` returns true if child exited normally
    - **`WEXITSTATUS(status)`** returns the actual exit status

  `WIFSIGNALED(status)` – returns true if the child process was terminated by a signal
    - **`WTERMSIG(status)`** returns signal number that terminated it

  `WIFSTOPPED(status)` – returns true if process was stopped instead of being terminated
    - **`WSTOPSIG(status)`** returns signal number that stopped it

Next Time

- Continue exploring the UNIX process API