IPC & Signals

CST 357/457 – Systems Programming
Michael Ruth, Ph.D.
Associate Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Discuss signals in terms of purpose, use, lifecycle and their symbolic identifiers
- Explain a common set of signals including their default operations and events
- Discuss basic signal management including sending signals, catching signals, ignoring, and waiting for signals
- Explain reentrancy as it relates to signals and discuss blocking and restoring signals

IPC

- Inter-process Communication (IPC)
- There are many reasons one process may wish to communicate with another
  - Synchronization of external resources
  - Producer/consumer relationships
    - Server/client… etc

IPC Mechanisms

- There are several methods we will consider:
  - Signals (chapter 10)
  - Pipes (really from I/O but we'll discuss)
  - Queues (message passing)
  - Semaphores
  - Sockets

Signals?

- Signals are software interrupts that provide a mechanism for asynchronous events
  - These events can come:
    - from outside the system
      - EX: User types [CTRL-C] to interrupt processing
  - From activities within a program
    - Divide by zero
- Signals are a primitive form of IPC
  - IPC = InterProcess Communication
- Events occur asynchronously and the program handles them asynchronously
  - Signal ***handlers*** are ***registered*** with the kernel

Signal Lifecycle

- First, a signal is ***raised***
  - AKA ***sent, generated***
- The kernel then ***stores*** the signal
- Finally, when appropriate, the kernel ***handles*** the signal
  - The kernel can perform one of three functions:
    - Ignore
      - No action is taken
    - Catch & Handle
      - Goes to registered function
    - Perform the Default Action
      - Depends on the signal

Signal Identifiers

- Every signal has a symbolic name that starts with the prefix SIG
  - Defined in header named <signal.h>
  - Actually **int**, but always use symbolic name
- You can generate a list of signals on your system by typing kill –l
  - NOTE: kill is used to send signals to programs
    - We'll see this later

Signals (1)

Signal	Description	Default
[redacted]		
SIGALRM	Sent by alarm()	Terminate
[redacted]		
SIGCHLD	Child has terminated	Ignored
[redacted]		
SIGPFE	Arithmetic Exception	Terminate w/CD
[redacted]		
SIGILL	Process Tried to Execute an Illegal Instruction	Terminate w/CD
[redacted]		
SIGIO	Asynchronous IO Event	Terminate/Ignore
[redacted]		

Signals (2)

Signal	Description	Default
[redacted]		
SIGPROF	Profiling Timer Expired	Terminate
[redacted]		
SIGQUIT	User generated the quit character [CTRL-\]	Terminate w/CD
[redacted]		
SIGSTOP	Suspends Execution of the process	Stop
[redacted]		
SIGTERM	Catchable Process Termination	Terminate
[redacted]		
SIGTSTP	User Generated Suspend Character [CTRL-Z]	STOP
[redacted]		

Signals (3)

Signal	Description	Default
[redacted]		
SIGURG	Urgent IO Pending	Ignored
[redacted]		
SIGUSR2	Process Defined Signal	Terminate
[redacted]		
SIGWINCH	Size of controlling terminal window changed	Ignored
[redacted]		
SIGXFSZ	File resource limits were exceeded	Terminate w/CD

Basic Signal Management

- `#include <signal.h?`
- `typedef void (*signalhandler_t)(int)`
- `signalhandler_t signal(int signo, sighandler_t handler)`

- Successful call removes the current action for the given signal and handles the signal with the given handler
- Signal Handler:
  - `void my_handler(int signo)`
- You can also use this function to instruct the kernel to IGNORE or go back to the default handler using special values for handler:
  - SIG_DFL – default
  - SIG_IGN - ignore

Waiting For a Signal

- Useful for debugging purposes, the following mechanism blocks until a signal is received that is handled or terminates
  - `#include <signal.h>`
  - `int pause(void)`

Execution & Inheritance

Signal Behavior	Across Forks	Across Execs
[redacted]		
Default	Inherited	Inherited
[redacted]		
Pending	Not Inherited	Inherited

Mapping Signal Numbers to Strings

- It's often important to get the name rather than the number, so…
  - **`extern const char * const sys_siglist[]`**
    - Actually your best bet!
  - **OR**
  - **`#include <signal.h>`**
  - **`void psignal(int signo, const char *msg)`**

Sending a Signal

- The kill() system call is the basis of the kill utility sends a signal from one process to another:
  - **`#include <sys/types.h>`**
  - **`#include <signal.h>`**
  - **`int kill(pid_t pid, int signo)`**
- In normal use, sends signo to process identified by pid
  - If pid = 0, sends signo to all processes in process group
  - If pid = -1, sends signo to all processes it has permission to send a signal to except itself/init
- On success, returns 0, otherwise returns -1 and sets errno to one of the following:
  - EINVAL (bad signal)
  - EPERM (don't have permission)
  - ESRCH (bad process or a zombie)

Sending Signals with Kill Utility

- **kill [-signal] pid**
  - Sends TERM by default!
- **kill -9 pid**
  - "Guarantees" that the process will die
- Terminal Signals:
  - **<CTRL-C> → SIGINT**
  - **CTRL-\ → SIGQUIT**
  - **CTRL-Z → SIGSTP**

Permissions

- In order to be able to send a signal from one process to another we need permission
  - A process with CAP_KILL (usually root) can send a signal to any process
  - A process without CAP_KILL requires that the effective or real UID equal to the real or saved UID of the receiving process
    - A user can only send signals to processes it owns

Sending a Signal to Yourself

- The raise function allows you to send a signal to yourself
  - **`#include <signal.h>`**
  - **`int raise(int signo)`**

Reentrancy

- A signal can come from anywhere while the process is executing code.
  - It could be in the middle of something delicate that would leave the system in an inappropriate state
- Signal handlers should never assume they know where or what the system was doing and should be very careful about:
  - Manipulating global variables (or use)
- What about system calls that use buffers? or use files? Allocate memory?
  - Some functions are clearly not reentrant
  - If a signal arrives in the middle of a nonreentrant operation, and the signal handler invokes the nonreentrant code, chaos…
- A ***reentrant function***, then, is a function that does not manipulate static data, must manipulate only stack data or data supplied by the call, and must NOT call nonreentrant functions

Signal Handlers & Reentrancy

- Since we don't know anything about the code that sent us to the handler, we must only ensure that the handler itself does not use functions that are not reentrant
  - There is a list on page 349-350
- We'll learn to block signals to ensure that we don't receive signals at critical times
  - Critical times though depend on reentrancy

Signal Sets

- A datatype which represents multiple signals

- #include <signal.h>
  - `int sigemptyset(sig_set_t *set);`
    - Sets the set to an empty set (init)
  - `int sigfillset(sig_set_t *set);`
    - Initializes the signal set so that all signals are included
  - `int sigaddset(sig_set_t *set, int signo);`
    - Adds a signal to the set
  - `int sigdelset(sig_set_t *set, int signo);`
    - Deletes a signal from the set
  - `int sigismember(const sig_set_t *set, int signo);`
    - Determines if signal exists in set

Signal mask

- The signal mask of a process is the set of signals that are *blocked* from delivery to that process
- **`#include <signal.h>`**
- **`int sigprocmask(int how, const sigset_t *restrict set, sigset_t *restrict oset)`**
  - oset is a non-null pointer, the current signal mask is returned through it
  - if set is non-null, what happens is dependent on how variable which could be one of:
    - SIG_BLOCK – block the signals on the list
    - SIG_UNBLOCK – unblock the signals on the list
    - SIG_SETMASK – set the mask to the list

Pending Signals

- To retrieve a list of signals pending for this process:
  - **`int sigpending(sigset_t *set)`**
    - The result is stored in set

Waiting for a Set of Signals

- Allows a process to temporarily change its signal mask and then wait until a signal is raised that either terminates or is handled
  - **`#include <signal.h>`**
  - **`int sigsuspend(const sigset_t *sigmask)`**
    - If a signal terminates the process, call never returns
    - If a signal is raised and handled, call returns -1
- A common use of sigsuspend is to retrieve signals that might have arrived but were blocked during a critical region
  - Process uses sigprocmask to block, then sigsuspend

Other Forms of IPC

- Signals are an
  - Event notification mechanism (only notifies other processes of state of this process)
- The other forms of IPC differ in that they:
  - Can actually communicate rather freely (send information back and forth at will)

Pipes

- Pipes are the oldest form of UNIX IPC
  - They have two limitations:
    - half-duplex communication
    - Can only be used between processes that share a common ancestor
  - Pipes are still the most commonly used form of IPC
- There are really two mechanisms for dealing with Pipes in C
  - Formatted Pipes (we only care about this one)
  - Low Level Pipes

Formatted Pipes

- **`FILE *popen(char *command, char *mode)`**
  - Executes the function specified by command.
  - It creates a pipe between the ***calling program and the executed command***, and returns a pointer to a stream that can be used to either read from or write to the pipe.
  - ensures that any streams from previous calls that remain open in the parent process are closed in the new child process

Formatted Pipes and Direction

- If the file mode is "r"
  - The stdout of cmd string is the "input"
  - The file pointer (fp) reads the "output"
- If the file mode is "w"
  - The stdin of cmd string is the "output"
  - The file pointer (fp) writes the "input"

Closing a Pipe

- `int pclose(FILE *stream);`
  - Closes stream opened by popen
  - Waits for the command to terminate
  - Returns termination status (or -1 on failure)

popen example 1

```
int main(void) {
   int  cntr;
   FILE *pipe_fp;
   char *strings[5] = {"echo","bravo","alpha",
                       "charlie","delta"};

   if (( pipe_fp = popen("sort", "w")) == NULL)        {
      perror("popen"); exit(1);
   }
   for(cntr=0; cntr<MAXSTRS; cntr++) {
      fputs(strings[cntr], pipe_fp);
      fputc('\n', pipe_fp);
   }
   pclose(pipe_fp);

   return(0);
}
```

popen example 2

```
int main(void) {
        FILE *pipein_fp, *pipeout_fp;
        char readbuf[80];

        if (( pipein_fp = popen("ls", "r")) == NULL)  {
                perror("popen"); exit(1); }
        if (( pipeout_fp = popen("sort", "w")) == NULL) {
                perror("popen"); exit(1); }
        /* Processing loop */
        while(fgets(readbuf, 80, pipein_fp))
                fputs(readbuf, pipeout_fp);

        /* Close the pipes */
        pclose(pipein_fp);
        pclose(pipeout_fp);

        return(0); }
```


Final Note

- Since popen() uses the shell all shell expansion characters are available for use!
- In addition, more advanced techniques such as redirection can be used:
  - `popen("ls ~scottb", "r");`
  - `popen("sort > /tmp/foo", "w");`
  - `popen("sort | uniq | more", "w");`

Summary

- Discussed signals in terms of purpose, use, lifecycle and their symbolic identifiers
- Explained a common set of signals including their default operations and events
- Discussed basic signal management including sending signals, catching signals, ignoring, and waiting for signals
- Explained reentrancy as it relates to signals and discuss blocking and restoring signals

Questions?