

Week 8

Inter-process Communication

School of Information Technology and Electrical Engineering
The University of Queensland

Outline

- Inter-process communication (IPC)
 - Signals (revisited)
 - File-based IPC – pipes
 - Semaphores
 - Brief mention of some other mechanisms
 - Shared memory
- Credits:
 - Bryant and O'Halloran, "Computer Systems: A Programmer's Perspective"
 - Silberschatz et. al, "Operating Systems concepts"
 - Rochkind, "Advanced UNIX Programming"

2

Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system
 - Kernel abstraction for exceptions and interrupts
 - Sent from the kernel (sometimes at the request of another process) to a process
 - Different signals are identified by small integer ID's
 - The only information in a signal is its ID and the fact that it arrived

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

Signal Concepts

- Sending a signal
 - 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 divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
 - Another process has invoked the `kill()` system call to explicitly request the kernel to send a signal to the destination process

4

Signal Concepts (cont)

- Receiving a signal
 - A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal
 - Three possible ways to react:
 - Ignore the signal (do nothing)
 - Terminate the process
 - *Catch* the signal by executing a user-level function called a *signal handler*
 - Akin to a hardware exception handler being called in response to an asynchronous interrupt.

5

Signal Concepts (cont)

- A signal is *pending* if it has been sent but not yet received
 - There can be at most one pending signal of any particular type.
 - Important: Signals are not queued
 - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded.
- A process can *block* the receipt of certain signals.
 - Blocked signals can be delivered, but will not be received until the signal is unblocked.
- A pending signal is received at most once.

6

COMP2303
COMP7306

Signal Concepts

- Kernel maintains pending and blocked bit vectors in the context of each process.
 - pending – represents the set of pending signals
 - Kernel sets bit k in pending whenever a signal of type k is delivered.
 - Kernel clears bit k in pending whenever a signal of type k is received
 - blocked – represents the set of blocked signals
 - Can be set and cleared by the application using the `sigprocmask()` function.
- Note – some signals can not be blocked

7

COMP2303
COMP7306

Signal Example

- To be presented in class

8

COMP2303
COMP7306

Unix Files

- A Unix *file* is a sequence of m bytes:
 - $B_0, B_1, \dots, B_k, \dots, B_{m-1}$
- All I/O devices are represented as files:
 - `/dev/dsk/c1t1d0s3` (/usr disk partition)
 - `/dev/tty2` (terminal)
- Even the kernel is represented as a file:
 - `/dev/kmem` (kernel memory image)
 - `/proc` (kernel data structures)

9

COMP2303
COMP7306

Unix File Types

- Regular file
 - Binary or text file.
 - Unix does not know the difference!
- Directory file
 - A file that contains the names and locations of other files.
- Links
 - Symbolic links to other files
- Character special and block special files
 - Terminals (character special) and disks (block special)
- FIFO (named pipe)
 - A file type used for interprocess communication
- Socket
 - A file type used for network communication between processes

10

COMP2303
COMP7306

Unix I/O

- The elegant mapping of files to devices allows kernel to export simple interface called Unix I/O.
- Key Unix idea: All input and output is handled in a consistent and uniform way.
- Basic Unix I/O operations (system calls):
 - Opening and closing files
 - `open()` and `close()`
 - Changing the *current file position* (seek)
 - `lseek`
 - Reading and writing a file
 - `read()` and `write()`

11

COMP2303
COMP7306

Opening Files

- Opening a file informs the kernel that you are getting ready to access that file.


```
int fd; /* file descriptor */
if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```
- Returns a small identifying integer *file descriptor*
 - `fd == -1` indicates that an error occurred
- Each process created by a Unix shell begins life with three open files associated with a terminal:
 - 0: standard input
 - 1: standard output
 - 2: standard error

12

How the Unix Kernel Represents Open Files

- Two descriptors referencing two distinct open disk files. Descriptor 1 (stdout) points to terminal, and descriptor 4 points to open disk file.

Part of process control block

Info in stat struct

13

File Sharing

- Two distinct descriptors sharing the same disk file through two distinct open file table entries
 - E.g., Calling `open()` twice with the same filename argument

14

How Processes Share Files

- A child process inherits its parent's open files. Here is the situation immediately after a `fork()`

15

I/O Redirection

- Question: How does a shell implement I/O redirection?
 - `unix> ls > foo.txt`
- Answer: By calling the `dup2(oldfd, newfd)` function
 - Copies (per-process) descriptor table entry `oldfd` to entry `newfd`

16

I/O Redirection Example

- Before calling `dup2(4, 1)`, stdout (descriptor 1) points to a terminal and descriptor 4 points to an open disk file.

17

I/O Redirection Example (cont)

- After calling `dup2(4, 1)`, stdout is now redirected to the disk file pointed at by descriptor 4.

18

COMP2303
COMP7306

Break

19

COMP2303
COMP7306

Useful UNIX

- wc
- tee

20

COMP2303
COMP7306

Pipes

- Easy to create using a shell...
- Examples
 - ls | more
 - who | wc -l
 - Number of login sessions on machine
 - who -q will report something similar
 - who | cut -d " " -f1 | sort | uniq | wc -l
 - Number of distinct users
- Output of one program (standard output) is input to next (standard input)

21

COMP2303
COMP7306

Bidirectional Pipes

- Pipes can be bidirectional in most modern OSs
 - Unidirectional in early UNIXes
 - Not portable though – often best to create two unidirectional pipes
- Can't create bidirectional pipes using shell!
 - Individual processes can create bidirectional pipes

22

COMP2303
COMP7306

pipe() System Call

- *To be shown in class*

23

COMP2303
COMP7306

Connecting processes

- Q: How can you create a pipe between two arbitrary processes?
- A: You can't - process can't pass a meaningful file descriptor to another process
- How to do it then?
 - Create pipe and then fork() – processes will share file descriptors
- Processes communicating via pipes must therefore be related, e.g.
 - parent, child
 - siblings of a common parent (e.g. as in shell)
 - grand-parent, grand-child
 - etc

24

COMP2303
COMP7306

Pipe example

- *To be provided*

25