INTERPROCESS COMMUNICATION (IPC)

Ken Wong
Washington University, St. Louis
kenw@arl.wustl.edu
www.arl.wustl.edu/~kenw

UNIX IPC

<table>
<thead>
<tr>
<th>IPC Type</th>
<th>POSIX.4</th>
<th>SVR5</th>
<th>4.4BSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipes (Half Duplex)</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIFOs (Named Pipes)</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Stream Pipes (Full Duplex)</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>Named Stream Pipes ()</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>Message Queues ()</td>
<td>•</td>
<td>•</td>
<td>*</td>
</tr>
<tr>
<td>Semaphores</td>
<td>•</td>
<td>•</td>
<td>*</td>
</tr>
<tr>
<td>Shared Memory ()</td>
<td>•</td>
<td>•</td>
<td>*</td>
</tr>
<tr>
<td>Sockets</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>Streams</td>
<td></td>
<td>•</td>
<td></td>
</tr>
<tr>
<td>Remote Procedure Call (RPC)</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
</tbody>
</table>

OVERVIEW

- IPC mechanisms allow processes to communicate and synchronize their actions
- Forms of IPC
  - Direct Communication
    - SEND (P, M): Send a message M to process P
    - RECEIVE (Q, M): Receive a message M from process Q
  - Indirect Communication (i.e., through mailboxes or ports)
    - SEND (A, M): Send a message M to mailbox A
    - RECEIVE (A, M): Receive a message M from mailbox A
- Some Issues
  - How do we name processes and mailboxes?
  - Can we share mailboxes? If so, who gets access to the mailbox?
  - How do we efficiently implement the communication (and synchronization)?

SHELL SCRIPT PIPE

- Script x
  
  y | z &
sleep 20
wait $!

  # y and z are shell scripts which just sleep
  # $! is PID of most recently backgrounded command

- Output of ps -l while x is sleeping

  F UID PID PPID TIME COMMAND
  8 1024 17273 17271 0:03 -tcsh
  8 1024 17648 17273 0:00 /bin/sh ./x
  8 1024 17649 17648 0:00 /bin/sh ./z
  8 1024 17650 17649 0:00 /bin/sh ./y
  8 1024 17651 17650 0:00 sleep 20
  8 1024 17652 17648 0:00 sleep 20
  8 1024 17653 17649 0:00 sleep 20
UNIX PIPE EXAMPLE (1)

... Requisite includes and "myinc.h" ...
int
main (void) {
  ... Variable definitions ...
Pipe(fd);
  pid = Fork();
  if (pid > 0) { // parent
    close (fd[0]); // close head of pipe
    Write (fd[1], msg, msgsz); // write msg to pipe tail
  } else { // child
    close (fd[1]); // close tail of pipe
    n = Read (fd[0], line, msgsz); // read msg from pipe head
    printf ("%s\n", line); // write msg to stdout
  }
return 0;
}

UNIX PIPE EXAMPLE (2)

• myinc.h

  static inline int Pipe(int fd) {
    int rc;
    if ((rc=pipe(fd)) < 0) ... Pipe Error ...
    return rc;
  }
  static inline pid_t Fork(void) {
    pid_t pid;
    if ( (pid = fork()) < 0) ... Fork Error ...
    return pid;
  }
  static inline ssize_t Read(int fd, void *buf, ssize_t n) {
    if (read(fd, buf, n) != n) ... Read Error ...
    return n; // doesn’t handle EOF
  }

UNIX PIPE EXAMPLE (3)

• myinc.h (Continued)

  static inline ssize_t Write(int fd, const void *buf, ssize_t n) {
    if (write(fd, buf, n) != n) ... Write Error ...
    return n; // doesn’t handle EOF
  }

• Other Includes: <unistd.h>, <stdio.h>

• Variable Definitions
  int n, fd[2];
  pid_t pid;
  char line[MAXLINE];
  char *msg = "hello\n";
  const int msgsz = sizeof(msg);

UNIX PIPE MOVIE (1)
Parent creates pipe

User Process

Pipe

OS Kernel
**UNIX PIPE MOVIE (2)**

Parent forks child process

![Diagram of UNIX PIPE MOVIE (2)](image)

**UNIX PIPE MOVIE (3)**

Parent (child) closes head (tail) of pipe

![Diagram of UNIX PIPE MOVIE (3)](image)

**UNIX PIPE:**

- **Limitations**
  - They are half-duplex (data flows in one direction)
  - Can only be used between 2 processes that have a common ancestor
    - Process A creates a pipe
    - A calls 'fork' to create a child B
    - A pipe is used between processes A and B
    - i.e., the shell command "A | B"
  - **Signature**
    - #include <unistd.h>
    - int pipe (int FileDes[2]);
  - Returns: 0 if OK; -1 if error
  - FileDes[0] is open for reading
  - FileDes[1] is open for writing

**PIPE OPERATION**

- **Reading from a pipe whose write end has been closed**
  - 'read' returns 0 to indicate end of file after all data has been read
- **Writing to a pipe whose read end has been closed**
  - Generates a SIGPIPE signal
  - 'write' returns an error with 'errno' set to EPIPE if we ignore the signal or catch it and return from the signal handler (See strerror(3C) or perror(3C))
- **The constant 'PIPE_BUF' specifies the kernel's pipe buffer size**
  - Interleaving of messages from multiple writers will occur if we try to send more than 'PIPE_BUF' (512 in POSIX, 5120 in Solaris 2.7) bytes in a message
- **write is atomic if the size is less than 'PIPE_BUF' bytes**
- **There are NO message boundaries**
  - 'n = read (fd, buff, 100)' will attempt to read 100 bytes and could return less than 100
FIFO EXAMPLE (CLIENT/SERVER)

UNIX FIFOs (Named Pipes)

- A FIFO allows unrelated processes to exchange data
  - A FIFO is a type of file
  - Creating a FIFO is similar to creating a file

Signature

```c
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *PathName, mode_t Mode);
```

- `PathName` contains a string which is the name of the FIFO
- `Mode` is the mode of the FIFO and is the same as a file mode
- Returns: 0 if OK; -1 on error

MKFIFO AND FILE MODE

- Specifies the access permission of the file

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Permission</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>S_IRUSR</td>
<td>user-read</td>
<td>(0400; i.e., r--)</td>
</tr>
<tr>
<td>S_IWUSR</td>
<td>user-write</td>
<td>(0200; i.e., w--)</td>
</tr>
<tr>
<td>S_IXUSR</td>
<td>user-execute</td>
<td>(0100; i.e., x--)</td>
</tr>
<tr>
<td>S_IRGRP</td>
<td>group-read</td>
<td></td>
</tr>
<tr>
<td>S_IWGRP</td>
<td>group-write</td>
<td></td>
</tr>
<tr>
<td>S_IXGRP</td>
<td>group-execute</td>
<td></td>
</tr>
<tr>
<td>S_IXOTH</td>
<td>other-execute</td>
<td>(0001)</td>
</tr>
<tr>
<td>S_IWOTH</td>
<td>other-write</td>
<td></td>
</tr>
</tbody>
</table>

- Usage:

  ```c
  rc = mkfifo (fifoPipe0, S_IRUSR | S_IRGRP); // usr+group read
  rc = mkfifo (fifoPipe0, 0440); // usr+group read
  rc = mkfifo (fifoPipe1, S_IRUSR | S_IWUSR); // usr read+write
  ```

FIFO OPERATIONS

- After creating a FIFO using `mkfifo`, normal file I/O functions can be used on the FIFO

  - `open`
  - `write`: Maximum message is `PIPE_BUF` bytes long
  - `read`
  - `close`
  - `unlink`

  - The Default Operation

    - An open for only reading will block until some other process opens the FIFO for writing.
    - An open for only writing will block until some other process opens the FIFO for reading.
    - The nonblocking flag (O_NONBLOCK) affects what happens in the above two cases.