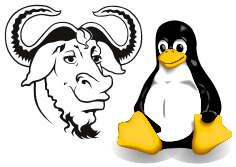

Workshop on Inter Process Communication — Solutions

1 Background

Threads can share information with each other quite easily (if they belong to the same process), since they share the same memory space. But processes have totally isolated memory spaces, and cannot talk to each other easily. However, often we need processes to send information to each other, so the need for Inter Process Communication, or IPC.

We examine a few methods for processes to talk with each other, and try out some of the simpler ones.

2 System Calls

We use a number of system calls today. Each one has its own manual page in *section 2*. For example, to read about the `fork()` system call, see `man 2 fork`. Here are some function prototypes for each system call that we use, copied from the manual pages.

2.1 System Calls for Pipes

The `pipe()` system call sets up a one-way communication channel between two processes. The parameter is an array of two file descriptors. `filedes[0]` is for reading, `filedes[1]` is for writing. These are file descriptors; you have used file descriptors 0, 1 and 2 for standard input, standard output and standard error already. Every open file has a file descriptor, including a pipe. You write to the second file descriptor and read from the first one. See figure 1 on the following page.

```
#include <unistd.h>
int pipe(int filedes[2]);
```

2.2 System Calls for Signals

`signal()` installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function, or either `SIG_IGN` or `SIG_DFL`.

Using a signal handler function for a signal is called "catching the signal". The signals `SIGKILL` and `SIGSTOP` cannot be caught or ignored.

You can see a complete list of signals if you type `kill -l`.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

2.3 Error Handling of System Calls

Note that all the system calls used here return the value −1 when there is an error, and set a global variable called `errno` to an error number. There is a library function called `perror()` which can print an error message when `errno` is set. See `man 3 perror` for more information, and see the example `signal.c` in program 3 on page 4.

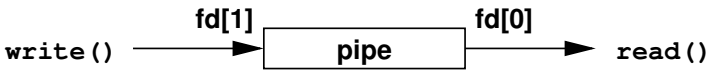

Figure 1: How to read and write a pipe: read from the first, write to the second file descriptor.

3 Interprocess Communication (IPC)

IPC Techniques include:

- pipes, and named pipes (FIFOs)
- sockets
- messages and message queues
- shared memory regions
- signals

All have some overhead

3.1 Pipes and Named Pipes

- Circular buffer, can be written by one process, read by another
  - related processes can use *unnamed pipes*
    - Trivial in shell programming: `ls | grep rpm`
    - Quite easy in C; see program 1 on the following page. The `pipe()` system call takes an array of two integers.
    - A limitation of unnamed pipes is that the two processes must be related (have a common ancestor) — usually, between a parent and child process.
  - unrelated processes can use *named pipes* — sometimes called FIFOs
- Limitation: *one direction* only

3.2 Sockets

- Very similar to network progamming with sockets, but on the same machine
- A little more complicated than pipes, but can send data *both ways*

3.3 Messages

- *Messages* — POSIX provides system calls `msgsnd()` and `msgrcv()`
  - message is block of text with a type
  - each process has a message queue, like a mailbox
  - processes are suspended when attempt to read from empty queue, or write to full queue.

Program 1 A Pipe—simplest IPC. The `pipe()` system call takes an array of two file descriptors; one process reads from the first, the other process writes to the second.

```
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>
#define BUFSIZE 30
int main() {
        int pipe_file_descriptors[2];
        char buf[BUFSIZE];

        pipe( pipe_file_descriptors );

        if ( fork() == 0 ) {
                printf( " CHILD: reading from pipe\n" );
                read( pipe_file_descriptors[0], buf, BUFSIZE );
                printf( " CHILD: read \"%s\"\n", buf );
        } else {
                printf( "PARENT: writing to the pipe\n" );
                write( pipe_file_descriptors[1], "test", BUFSIZE );
                printf( "PARENT: finished writing\n" );
                wait( NULL );
        }
        return 0;
}
```

3.4 IPC — Shared Memory

- *Shared Memory* — a Common block of memory shared by many processes
- Fastest way of communicating
- Requires synchronisation

3.5 IPC — Signals

- Some *signals* can be generated from the keyboard, i.e., Control-C — interrupt (`SIGINT`); Control-\ — quit (`SIGQUIT`), Control-Z — stop (`SIGSTOP`)
- A software *signal* that sends an asynchronous number to a process
- implemented as single bits in a field in the processs table, so cannot be queued
- A process may respond to a signal with:
  - a default action (i.e., terminate)
  - a signal handler function (see `trap` in shell programming notes), or
  - ignore the signal (unless it is `SIGKILL` or `SIGSTOP`)
  - In shell programming, we used `trap` to change response to a signal, as in program 2 on the next page.
  - POSIX introduces many functions to handle sets of signals, but the old ANSI C interface is quite simple—see program 3 on the next page

4 Procedure

1. Take the program `pipe.c` in program 1, compile and execute it.

Program 2 A shell script `signal.sh` that catches the interrupt signal (`SIGINT`), like the C program 3.

```
#! /bin/sh
trap "echo 'Got an Interrupt!'" INT
echo Enter a string:
read string
echo You entered: $string
```

Program 3 A program `signal.c` that catches the Interrupt signal (`SIGINT`), slightly modified from [Beej97].

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#define MAX_LEN 200

void sigint_handler( int sig ) {
        printf( "Got an Interrupt!\n" );
}

int main( void ) {
        char s[MAX_LEN];

        if ( signal( SIGINT, sigint_handler ) == SIG_ERR ) {
                perror( "signal" );
                exit( 1 );
        }

        printf( "Enter a string:\n" );

        if ( fgets( s, MAX_LEN, stdin ) == NULL )
                perror( "fgets" );
        else
                printf( "You entered: %s", s );

        return 0;
}
```

2. Write a list of all the "things that can go wrong" in `pipe.c`:

3. Modify it to add error handling using `perror()`. See section 2.3 on page 2.

4. Modify `pipe.c` to send a large block of data (say a megabyte), and time how long it takes using the shell built-in function `time`. You can use it like this:

   ```
   $ time ./pipe2
   ```

   where `pipe2` is the executable of your modified `pipe.c`.

I suggest that you initialise the buffer in the parent (writing) process using a `for` loop, then send that copy of the buffer to the other (child) process. Add a null character `'\0'` at the end of your buffer before the parent prints it with `printf()`.

5. Run the shell script `signal.sh` in program 2 on the preceding page.

6. Modify it to handle other signals, such as the `TERM`, `HUP` and `QUIT` signals. How do you send these signals to the program?

7. Modify the C program `signal.c` in program 3 on the previous page similarly.

8. Write a simple shell script to send signals to the two programs `signal.sh` and `signal.c`.

References

[Beej97] *Beej's Guide to Unix Interprocess Communication* by Brian "Beej" Hall: `http://www.ecst.csuchico.edu/~beej/guide/ipc/` is the main source for this workshop.

[Ste92] W. Richard Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992, call number in our library: QA 76.76 .063 S754 1992