Lecture 8: Unix Pipes and Signals (Feb 10, 2005)

Yap

February 17, 2005

1 ADMIN

- Our Grader will be Mr. Chien-I Liao (cil217@nyu.edu).
- Today's Lecture, we will go into some details of Unix pipes and Signals. This is in preparation of upcoming programming assignments.
- The primary reference for this lecture is the book **Jump Start to C Programming & The Unix Interface**, by Derek Kiong Beng Kee, Prentice Hall (Singapore), 1995. I will refer to this as [Jumpstart].

2 Review

- Q: Give two models for cooperating threads within a process.

  A: One model is manager/worker. The other model is assembly line (or pipeline).

3 Unix System Calls

- A system call looks just like a function call.
- The man pages will tell you which header files to include. E.g., to make the system call `sysinfo`, you can type "man -s 2 sysinfo" and it will tell you to do `#include <sys/systeminfo.h>`.
- Generally, system calls return the value `-1` to indicate failure.
- The most recent error code is stored in the external variable `errno`. This code can be used to index the external string array `sys_errlist` to obtain a concise description of the error. Here is a sample from [Jumpstart]:

```
//============================================
#include <stdio.h>
#include <sys/systeminfo.h>

...
int code = system_call();
if (code == -1) {
extern int errno;
extern char * sys_errlist[];
printf("%s\n", sys_errlist[errno]);
exit(1);
}
...
//============================================
```

- ```
  //============================================
  #include <stdio.h>

  void report_err(char * prefix)
  {
     extern int errno;
     extern int sys_nerr;    // size of array sys_errlist
     extern char *sys_errlist[];

     if (prefix != NULL)
      printf("%s: ", prefix);
     if (0 < errno && errno < sys_nerr)
      printf("%s\n ", sys_errlist[errno]);
     else
      printf("unknown error\n");
     exit(1);
  }
  //============================================
  // THIS CODE COMPILES IN CYGWIN AND SOLARIS
  ```

  You do not need to implement this routine because this code is equivalent to the library function `perror()` [Jumpstart].

4 Redirection of I/O

- Suppose we want to execute ”ls” and then send the output to a file called ”SPOOL”.
- The main process initially forks. Then it waits on its child process.
- Child Process:
  (1) creats a file ”SPOOL”
  ```

(2) make SPOOL the standard output
(3) close current standard output
(4) exec "ls /etc ." (listing 2 dirs)

- Here is the code

```
//============================================
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


main()
{  char *prog = "ls";
   char *argv[] = {"ls", "/etc", ".", NULL};
   int pid;
   printf("Before execute\n");
   pid = fork();
   if (pid==0) {
   /* child */
   int f;
   if ((f=creat("SPOOL", S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH)) == -1)
   perror("SPOOL");
   dup2(f, STDOUT_FILENO);
   close(f); // original not needed after dup2
   execvp(prog, argv);
   perror(prog);
   } else if (pid>0)
   /* wait for child */
   wait(NULL);
   else
   perror("execute");
   printf("After execute\n");
   exit(0);
}
//============================================
// THIS CODE COMPILES and RUNS IN CYGWIN AND SOLARIS
```

5 Using Pipes

- **IO File and their Descriptors**:

  1. All I/O must be done by writing to a file or reading from an abstract concept of an "IO file".

2. These must be created using `creat`, which returns an integer called the "file descriptor".
3. Initially, every process is given 3 such file descriptors: std input, std output, std error. These are numbered 0, 1, 2 or symbolically as `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO`.
4. If you close one of these IO files, then the next `creat` will reuse the lowest available unused integer.

- A pipe is an I/O Buffer associated with 2 IO file. To create a pipe, you call `pipe()`:

```
//============================================
#include <unistd.h>

int pipe(int fildes[2]);
```

  The two file descriptors of the new pipe are stored in `fildes[0]` and `fildes[1]`.

- Reading from `fildes[0]` will access data written into `fildes[1]` on a FIFO basis.
  Conversely, for a bidirectional pipe, reading from `fildes[1]` will access data written into `fildes[0]` on a FIFO basis.

- In Unix shells, you create such pipes by the "—" command. E.g.

```
> ls | less
```

  This command spawns two processes and creates a pipe shared by both processes. Process 1 executes the "ls" command, which lists the current directory, but its output is sent to `fildes[1]` of the pipe. Process 2 executes the "less" command, but its input is taken from `fildes[0]`. The output of Process 2 is your screen (the default).

- To use pipes, we exploit two facts: (1) a child process inherits all the files (in particular pipes) of the parent process, (2) calling exec() also does not modify the files.

- Here is a toy example:

```
//============================================
// file: hello_pipe.c from [Jumpstart]
// Synopsis: Simple illustration of pipe
//  (1) Creates a pipe
// (2) Writes "Hello World" into one end
// (3) Reads from the other end
// (4) Write the result of reading to std output.
```

```
//
// REMARK: THIS PROGRAM WORKS ON BOTH CYGWIN and SOLARIS.
// Chee Yap (Feb 10, 2005)

#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{  char *message = "Hello World!\n";
   int length = strlen(message);
   char buf[1024];
   int fdv[2];
   int n;

   if (pipe(fdv) == -1)
   perror("pipe");

   if (write(fdv[1], message, length) != length)
   perror("write to pipe");

   n = read(fdv[0], buf, length);
   if (n != length)
   perror("read from pipe");
   write(STDOUT_FILENO, buf, n);
   exit(0);
}
//==============================================
```

- Why is this a toy? Because pipes usually goes between 2 processes. Here is a less toy program:

```
//==============================================
// file: ls_less.c from [Jumpstart]
// Synopsis: Simple illustration of pipe between 2 processes:
//       Executes "ls" and pipes output to "less".
//  (1) Creates pipe
//  (2) Forks
//  (3) Child: redirect output to pipe, then exec "ls"
//  (4) Parent: redirect input to pipe, then exec "less"
//
// Chee Yap

#include <unistd.h>
```

```
#include <stdio.h>

main()
{

   int pid;
   int pipev[2];

   if (pipe(pipev) == -1)
   perror("pipe");

   pid = fork();
   if (pid == 0) {
   /* child */
   char *argv[] = {"ls", "/etc", ".", NULL};
   dup2(pipev[1], STDOUT_FILENO); // redirect output of ls
   close(pipev[0]); close(pipev[1]);
   execvp("ls", argv);
   perror("ls");
   } else if (pid >0) {
   /* parent */
   dup2(pipev[0], STDIN_FILENO); // redirect input of less
   close(pipev[0]); close(pipev[1]);
   execlp("less", "less", NULL);
   perror("less");
   } else
   perror("fork");

   printf("finish\n");
   exit(0);
}
//==============================================
```

- Pipe can be used for two-way communication, if the 2 processes can carefully coordinate when one will write/read.

  But to provide a more flexible 2-way communication, we can open two pipes, with the convention that one pipe is for communication in one direction, and the other is for the opposite direction.

- Other useful calls:

```
popen(), pclose() -- open and close pipes
getpid(), getppid() -- get process ID and parent process ID
chdir(), fchdir() -- change current directory
getuid(), getgid() -- get REAL user and group IDs
geteuid(), getegid() -- get EFFECTIVE user and group IDs
```

6 Using Signals and Exceptions

- Signals are asynchronous notification of events.

  – this is more efficient than sitting in a loop testing for the occurence of an event.

- Mechanics:
  – there are 20 different signals (with symbolic names like SIGHUP, SIGINT, SIGQUIT, ... corresponding to integers 1, 2, 3, ..., 20).
  – each process can decide on one of three options (”dispositions”) for each signal: ”handle”, ”ignore” or ”default”.
  – To handle the signal means to write your own event handler for the signal.
  – We say a signal is ”caught” if it is ”handled” or takes the ”default” action. (So ”caught” is the opposite of ”ignored”)
  – the system or another process can send signals to any process
  – when a signal is sent to a process, it will be interrupted, its event handler for that signal processed.
  – Under a normal return, the process continues its normal actions. But one possible action is to kill the process.

- Signal 9 (SIGKILL) cannot be ignored or caught. It kills the process receiving it.

- The `signal()` system call is used to change the ”disposition” of the process for a particular signal.

```
#include <signal.h>

void (*signal (int sig, void (*disp) (int) )) (int);
```

  1. The variable `sig` specifies the signal (integer from 1 to 20),
  2. The variable `disp` specifies the disposition, and can be `SIG_DFL`, `SIG_IGN` or a pointer to a function to handle the signal.
  3. If successful, `signal()` returns the previous disposition for this signal.
  4. If unsuccessful, it returns `SIG_ERR` and sets `errno`.

- The declaration of signal above is a remarkably complex construction!

$$void\ \underbrace{(\ *\ \underbrace{signal\ (\underbrace{int\ sig}_{fa}\ ,\ \underbrace{void\ (\ *\ disp\ )\ (int)}_{fb}\ )}_{fc}\ )}_{fd}\ (int); \tag{1}$$

1. Fragment fa describes the `sig` and fragment fb `disp` parts.
2. `sig` is an int
3. `disp` isa pointer to a handler function which takes an int argument, returning void.
4. Combining fa, fb with *signal*, we get fragment fc.
5. If we now simplify the original declaration be substituting fc, we get

   $$void\ (\ *\ fc)\ (int); \tag{2}$$

   THAT IS, is is a pointer to a handler.
6. Thus, this is our return type.
7. To check that this is the correct type, note that (2) is just like fragment fc!