Pipes

- Pipes are a way to allow processes to communicate with each other
  - Pipes implement one form of IPC (Interprocess Communication)
  - This allows synchronization of process execution
- There are two kinds of pipes
  - named pipes
  - unnamed pipes

Pipes (cont’d)

- Pipes are uni-directional
  - They can only transfer data in one direction
  - If you want two processes to have a two-way conversation, you must use two pipes

Pipes Implement a FIFO

- A FIFO (First In, First Out) buffer is like a queue or a line at a movie theater
- Elements are added at one end of the queue and exit the other end in the same order
- There is no way for any individual element to move ahead of another
Multiple Inputs

- It is possible to have multiple feeders of a pipe but there is NO guarantee of order beyond first in, first out

Pipes (cont'd)

- Traditional implementation of pipes uses the file system for storage
  - This allows processes to communicate even though they don’t know what processes are at the other end of the pipe

Pipes (cont'd)

- Unnamed pipes can only be used between processes that are children of the process that initiated the pipe
- Named pipes solve this problem - any process can communicate with another using named pipes

Unnamed Pipes

- Unnamed pipes are frequently used in Unix shells
- `cat myfile | grep key | sort | lpr`
Unnamed Pipes (cont’d)

• The parent process (the shell or shell script that creates the pipe) also spawns the child processes that access the pipe
  – *cat*, *grep*, *sort*, and *lpr* in this case
  – Note: the shell or script process that sets up the pipes CANNOT access the pipes itself!

• Processes using unnamed pipes must be closely related
• Normally the two processes represent a parent-child pair or a child-child pair from the same parent
• A parent process must create the pipe before it creates the child process(es) that will be using the pipe

Unnamed Pipes (cont’d)

• The actual pipe is created by the following UNIX system call:
  ```c
  int fd[2];
  if (pipe(fd) == -1) /* pipe creation failed */
  else /* a pipe was created */
  /* pipe creation failed */
  else
  /* a pipe was created */
  • Like all UNIX system (kernel) calls, "pipe" returns the int value of "-1" if it fails

• Assuming "pipe" does not fail, it fills the two element int array passed as an argument with the FD entries for the two ends of the pipe.
  – fd[0] contains the FD for the read end of the pipe
  – fd[1] contains the FD for the write end of the pipe
Unamed Pipes (cont'd)

• At this point the pipe exists but is not very useful as both ends are connected to the same process.

  fd[0]

  Process "A"

  fd[1]

  Direction of data flow

  Result of call to pipe

• And what you have is a shared pipe that is actually bidirectional for the moment.
• However, since the synchronization controls only support unidirectional pipes, you actually have a very dangerous situation!

Unnamed Pipes (cont'd)

• Next, the two processes must decide on the direction of the data flow in the pipe. Then, they each close one end of their shared pipe to make it unidirectional. (The pipe currently has four ends!)

• If the pipe represents a flow from process "A" to process "B", then
  – Process "A" must: close(fd[0]);
  – Process "B" must: close(fd[1]);
Named Pipes

• Named pipes can be accessed by any process that “knows the name”
• Named pipes appear in the user’s directory list

$ ls -l
prw-r--r-- 1 tugcu 0 Mar 27 19:33 mypipe

Named Pipes (cont’d)

• Like any other file, umask determines the initial permissions.
  – To prevent the potential for disturbances by other users, it is a good idea to remove the read/write permissions for Group and Other if they are normally provided by your umask

Named Pipe Creation

• Named pipes are created using the mknod or the mkfifo commands
  $ mkfifo name
  or
  $ mkfifo –m mode name
  $ mknod name p
• Make sure you remove (rm) your pipes after use!

Using Named Pipes

• First, create your pipes
  $ mknod pipe1 p
  $ mknod pipe2 p
  $ mknod pipe3 p
• Then, attach a data source to your pipes
  $ ls -l >> pipe1 &
  $ cat myfile >> pipe2 &
  $ who >> pipe3 &
Using Named Pipes (cont’d)

• Then, read from the pipes with your reader process
  $ cat < pipe1 | lpr
  $ spell < pipe2
  $ sort < pipe3
• Finally, delete your pipes
  $ rm pipe[1-3]

Using Pipes for Synchronization

• # process1
  mkfifo -m 600 outbound
  while true
do
  read msg
  echo "$msg rcvd by proc1"
  while test $-s outbound
do
    done
  echo Hi1 >> outbound
done < inbound

• # process2
  if test $-p inbound
  then
    echo Hi2 >> inbound
  fi
  while true
do
  read msg
  echo "$msg rcvd by proc2"
  while test $-s inbound
do
    done
  echo Hi2 >> inbound
done < outbound

Example #1: Parent Writes to Child

1 /* pipe.c - example of a pipe */ 2 3 #include <stdio.h> 4 #include <stdlib.h> 5 #include <sys/types.h> 6 #include <sys/wait.h> 7 #include <unistd.h> 8 #include <fcntl.h> 9 10 int main(int argc, char **argv) 11 { 12     pid_t id; 13     int count; 14     char buf[100]; 15     int fildes[2]; 16 17     pipe(fildes); 18 19     id = fork(); 20 (cont’d)
Example #1: Parent Writes to Child (cont’d)

Output:
# cc -o pipe pipe.c
# ./pipe
buf[ ] = Hello child!
#

Example 2: Parent Writes to Sort Program

```c
1 /* pipe-exec.c - use a pipe to send "input" to another program */
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <sys/wait.h>
6 #include <unistd.h>
7 #include <fcntl.h>
8 int main(int argc, char **argv) {
9     pid_t id;
10     int count;
11     int fildes[2];
12     pipe(fildes);
13     if ((id = fork()) == 0) {
14         /* we are in the child */
15         close(0); /* close stdin */
16         dup(fildes[0]); /* put read end of pipe into stdin index */
17         close(fildes[1]); /* close "write" end of pipe */
18         if (execl("/bin/sort", "sort", NULL) < 0) {
19             printf("execl failed
");
20             exit(1);
21         } else {
22             /* we are in the parent */
23             /* close "read" end of pipe */
24             close(fildes[0]);
25             /* send some data to sort */
26             write(fildes[1], "bbb
", 4);
27             write(fildes[1], "aaa
", 4);  
28             /* close write end to tell sort we are done */
29             close(fildes[1]);
30             id = wait(NULL); /* wait for child */
31             return(0);
32         }
33     } else { /* we are in the parent */
34         /* put read end of pipe into stdin index */
35         close(fildes[0]);
36         /* close "write" end of pipe */
37         close(fildes[1]);
38         return(0);
39     }
30 }
```

Output: # cc -o pipe-exec pipe-exec.c # ./pipe-exec
aaa
bbb
ttt