COMP 530
Introduction to Operating Systems

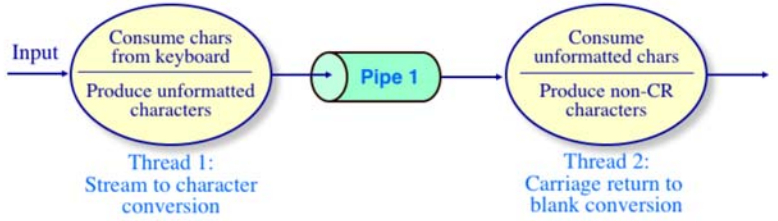
Notes on HW 4
Implementing Producer/Consumer systems
With Message Passing

Kevin Jeffay
Department of Computer Science
University of North Carolina at Chapel Hill
jeffay@cs.unc.edu
October 4, 2017

http://www.cs.unc.edu/~jeffay/courses/comp530

 1

HW 4
A message passing producer/consumer system

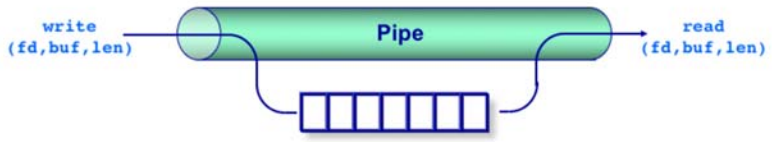


- Re-implement your solution to HW 3 as a pipeline of Unix processes that communicate via message passing using Linux "pipes"
- Major changes to HW3:
  - » Each one of your ST threads will become a Linux process
  - » Each one of your bounded buffers will become a Linux pipe

 2

HW 4

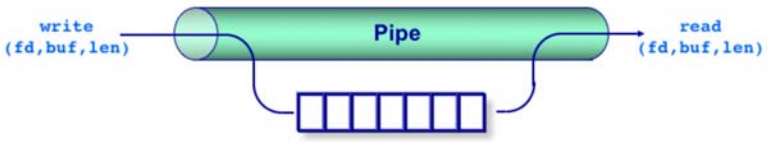
Details —Linux Interprocess Communication (IPC)



- IPC in Linux is like message passing except more general
- The programming abstraction is a "pipe" — a shared, in-memory file
  - » A queue of 4K bytes
  - » Linux *read/write* systems calls used to "pass messages" between processes

HW 4

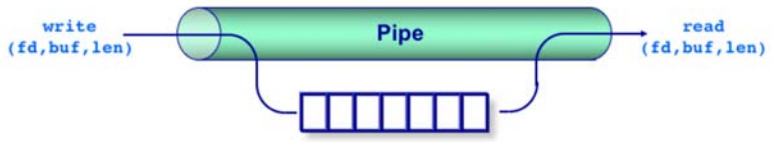
Details —Linux Interprocess Communication (IPC)



- Pipes provide a form of buffered, asynchronous message passing
  - » Data written to one end of the pipe by one process can be read at the other end of the pipe by another process
    - ❖ A reader of the pipe blocks when the queue is empty
    - ❖ A writer to the pipe blocks when the queue is full
  - » Data is handled in a FIFO (first-in-first-out) order
  - » Pipes are unidirectional

HW 4

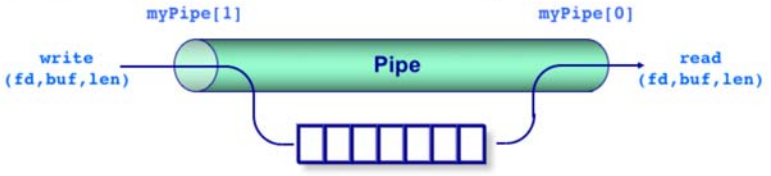
Details —Linux Interprocess Communication (IPC)



- Typically, a process creates a pipe just before it forks one or more child processes
  - » Creating a pipe results in a process receiving a read file descriptor and a write file descriptor
- The pipe is then used for communication between the parent and child process, or between two sibling processes
  - » One process uses the read file descriptor (and ignores the write descriptor) and a second process uses the write file descriptor (and ignores the read descriptor)

Linux Pipes

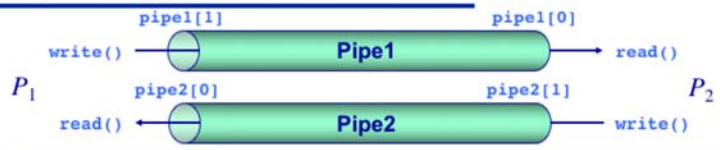
Programming details



- The Linux *pipe*( ) function:
  - » Is declared in the header file *unistd.h*
  - » `int pipe(int myPipe[2])`
    - Creates a pipe and puts the file descriptors for the reading and writing ends of the file (respectively) into myPipe[0] and myPipe[1]
    - If successful, pipe returns a value of 0
    - On failure, -1 is returned
  - » Functions *read*( ) and *write*( ) then used to send/receive messages

Linux Pipes

Programming details



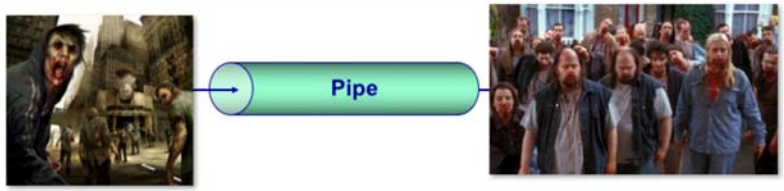

```
main() {
 int pipe1[2], pipe2[2];

 if (pipe(pipe1) == -1  ||  pipe(pipe2) == -1) error(...

 switch (childpid = fork()) {
   case 0:                          /* child */
     close(pipe1[1]);               /* write descriptor for pipe1 */
     close(pipe2[0]);               /* read  descriptor for pipe2 */
     client(pipe1[0],pipe2[1]);     /* start client program       */
   default :                        /* parent */
     close(pipe1[0]);               /* read  descriptor for pipe1 */
     close(pipe2[1]);               /* write descriptor for pipe2 */
     server(pipe2[0],pipe1[1]);     /* start server program       */
     while (wait((int *) 0) != childpid);   /* wait for child   */
   }
}
```


Linux Pipes

More programming details



- You're going to be creating lots of processes in this assignment…
- Make sure to check for zombie processes you've created!
  - » *ps -ef | egrep -e PID -e YOUR-LOGIN-NAME*
  - » *kill pid-number*
- Also, be sure to limit the maximum number of processes you can create