System Programming

Module 6

EGCI 252
System Programming

Computer Engineering Department
Faculty of Engineering
Mahidol University

Motivation: Pipes

What is a pipe?

- A pipe is an interface between two processes that allows those two processes to communicate (i.e., pass data back and forth)
- A pipe connects the STDOUT of one process (writer) and the STDIN of another (reader)
- A pipe is represented by an array of two file descriptors, each of which, instead of referencing a normal disk file, represent input and output paths for interprocess communication
- Examples:
  - ls | sort
  - ypcat passwd | awk –F: ‘{print $1}’ | sort
  - echo "2 + 3" | bc

Traditional Pipes

- How would you mimic the following command in a program:
  - $ ls /usr/bin | sort
- Create the pipe
- associate stdin and stdout with the proper read/write pipes via dup2() call
- close unneeded ends of the pipe
- call exec()

Pipes the easy way: popen()

- The simplest way (and like system() vs. fork(), the most expensive way) to create a pipe is to use popen():
  - #include <stdio.h>
  - FILE * popen(const char * cmd, const char * type);
  - ptr = popen(“/usr/bin/ls”, “r”);
- popen() is similar to fopen(), except popen() returns a pipe via a FILE *
- you close the pipe via pclose(FILE *);

popen()

- When called, popen() does the following:
  - creates a new process
  - creates a pipe to the new process, and assigns it to either stdin or stdout (depending on char * type)
    - “r”: you will be reading from the executing command
    - “w”: you will be writing to the executing command
  - executes the command cmd via a bourne shell

Pipe - Read from Process : Example

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    FILE *read_fp; char buffer[BUFSIZ + 1];
    int chars_read;
    memset(buffer, ‘\0’, sizeof(buffer));
    read_fp = popen(“uname –a”, “r”);
    if (read_fp != NULL)
    {
        chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
        if (chars_read > 0) { printf(“Output from pipe: \n%s\n”, buffer); }
        pclose(read_fp); exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Pipe – Write to Process : Example

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    FILE *write_fp; char buffer[BUFSIZ + 1];

    sprintf(buffer, “This is a message for writing to a process!\n”);
    write_fp = popen(“od –c”, “w”);
    if (write_fp != NULL)
    {
        fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
        pclose(read_fp); exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Create a low-level pipe

- #include <unistd.h>
- int pipe(int pipefd[2]);
- pipefd represents the pipe, and data written to pipefd[1] (think STDOUT) can be read from pipefd[0] (think STDIN)
- pipe() returns 0 if successful
- pipe() returns –1 if unsuccessful, and sets the reason for failure in errno (accessible through perror())

Low-level Pipes

- Pipes are half duplex by default, meaning that one pipe is opened specifically for unidirectional writing, and the other is opened for unidirectional reading (i.e., there is a specific “read” end and “write” end of the pipe)
- The net effect of this is that across a given pipe, only one process does the writing (the “writer”), and the other does the reading (the “reader”)
- If two way communication is necessary, two separate pipe() calls must be made, or, use SVR5’s full duplex capability (stream pipes)

Low Level Pipes : Example

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    int n_bytes, pipefd[2];
    char buffer[BUFSIZ + 1], data[] = “12345”;

    memset(buffer, ‘\0’, sizeof(buffer));
    if (pipe(pipefd) == 0)
    {
        n_bytes = write(pipefd[1], data, strlen(data)); printf(“%d bytes have been written!\n, n_bytes);
        n_bytes = read(pipefd[0], buffer, BUFSIZ); printf(“%d bytes have been read!\n, n_bytes);
        exit(EXIT_SUCCESS);
    }
    exit(EXIT_FAILURE);
}
```

Low Level Pipes : More Example

```
int main(void) //writer.c
{
      int n_bytes, pipefd[2], pid;
      char buffer[BUFSIZ + 1], data[] = “12345”;

      memset(buffer, ‘\0’, sizeof(buffer));
      if (pipe(pipefd) == 0)
      {
            pid = fork();
            switch (pid)
            {
                  case -1: fprintf(stderr, “Fork failed!\n”); break;
                  case  0: sprintf(buffer, “%d”, pipefd[0]);
                           execl(“reader”, “reader”, buffer, (char *) 0);
                           exit(EXIT_FAILURE);
                  default: n_bytes = write(pipefd[1], data, strlen(data));
                           printf(“%d bytes have been written from %d!\n”, n_bytes, getpid());
            }
      }
      wait(NULL);
      exit(EXIT_SUCCESS);
}
```

Low Level Pipes : More Example (Cont.)

```
int main(int argc, char *argv[]) //reader.c
{
    int n_bytes, fd;
    char buffer[BUFSIZ + 1];

    memset(buffer, '\0', sizeof(buffer));
    sscanf(argv[1], "%d", &fd);
    n_bytes = read(fd, buffer, BUFSIZ);
    printf("%d bytes have been read from %d : %s\n", n_bytes, getpid(), buffer);
    exit(EXIT_SUCCESS);
}
```

Communication via a pipe

- One thing is in common between all the examples we've seen so far:
  - All our examples have had shared file descriptors, shared from a parent processes forking a child process, which inherits the open file descriptors as part of the parent's environment for the pipe
- Question: How do two entirely unrelated processes communicate via a pipe?

Creating FIFOs in a program

- #include <sys/types.h>
- #include <sys/stat.h>
- int mkfifo(const char * path, mode_t mode);
  - path is the pathname to the FIFO to be created on the filesystem
  - mode is a bitmask of permissions for the file, modified by the default umask
- mkfifo returns 0 on success, -1 on failure and sets errno (perror())

Named Pipe : Example

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

#define FIFO_NAME "/tmp/my_fifo"

int main(int argc, char *argv[])
{
    int res, open_mode = 0;
```

```
if (argc < 2)
{
    fprintf(stderr, "Usage: %s <[O_RDONLY,
        O_WRONLY, O_RDWR,
        O_NONBLOCK]>\n", *argv);
    exit(EXIT_FAILURE);
}
argv++;
if (*argv)
{
    if (strncmp(*argv, "O_RDONLY", 8) == 0)
        open_mode |= O_RDONLY;
    if (strncmp(*argv, "O_WRONLY", 8) == 0)
        open_mode |= O_WRONLY;
    if (strncmp(*argv, "O_RDWR", 6) == 0)
        open_mode |= O_RDWR;
    if (strncmp(*argv, "O_NONBLOCK", 10) == 0)
        open_mode |= O_NONBLOCK;
}
```

Named Pipe : Example (Cont.)

```
if (access(FIFO_NAME, F_OK) == -1
{
    res = mkfifo(FIFO_NAME, 0777);
    if (res)
    {
      fprintf(stderr, “Could not create fifo
              %s\n”, FIFO_NAME);
      exit(EXIT_FAILURE);
    }
}
```

```
printf(“Process %d opening FIFO\n”, getpid());
res = open(FIFO_NAME, open_mode);
printf(“Process %d result %d\n”, getpid(), res);
sleep(5);
if (res != -1) (void) close(res);
printf(“Process %d finished\n”, getpid());

unlink(FIFO_NAME);
exit(EXIT_SUCCESS);
}
```

Assignment

- Write a simple chat program using double FIFOs
- Requirements:
  - The program's name must be "chat.c"
  - The program takes one command line argument (i.e., 1 or 2 to indicate the process 1 or process 2)
  - Create two FIFOs (fifo1to2 and fifo2to1)
  - Must be able to concurrently send and receive any messages between two "chat" processes.
  - Use the word "end chat" as a command to end the chat program

End of Module