COMP 4511

Inter-Process Communication
Part 1: Pipe and FIFO

Inter-process Communication (IPC) in Linux

- There are a few methods which can communicate between among processes on the same host
  - Files
  - Pipes (will be discussed in this lab)
  - FIFOs (will be discussed in this lab)
  - Message Queues
  - Shared Memory (will be discussed in the next lab)
  - Unix domain sockets
- For synchronization on the same host
  - Signals (for synchronization)
  - Semaphores (for synchronization)
- For processes on separated hosts
  - Remote Procedure Call (RPC)
  - Socket

Pipes

Introduction

- Piping is a process where the output of one process is made as the input of another process
- Example: Piping in Shell (using the | operator)
  - `ps aux | grep root | sort | less`
    - The `ps aux` command will output a list of running processes and the corresponding information.
    - After that, the output will be treated as an input of `grep`, which is a program to match the given pattern `root`
    - The lines will be sorted in an alphabetical order using the `sort` command
    - The final text result will be displayed using the `less` command

Two types of pipes

- Unnamed pipe
  - A unnamed pipe does not associated with any file
  - Can only be shared by related processes (descendants of a process that creates the unnamed pipe)
  - Pipe is an internal data structure maintained by the kernel
  - Read and write processes are assumed to run concurrently
- Named pipe
  - Like a file (create a named pipe, open, read/write)
  - Can be shared by any process
  - FIFO

Pipe in <stdio.h>

- `popen, pclose`
  - `FILE *popen(const char *command, const char *type);`
  - `int pclose(FILE *stream);`
- `popen` and `pclose` are higher level, invoking the shell to complete the operations
- We should avoid using `popen` and `pclose` for the programming assignments
- Example:

```c
#include <stdio.h>
#define MAXSTRS 5
int main()
{
    int i;
    FILE *pipe_fp;
    char *strings[MAXSTRS] = {"Ann","Dog","Bob","Egg","Cat"};

    pipe_fp = popen("sort", "w"); // create a "sort" pipe
    if (!pipe_fp) {
        perror("popen"); // error handling
        exit(1);
    } else {
        for (i=0; i<MAXSTRS;i++) {
            fputs(strings[i], pipe_fp); // send a string to the pipe
            fputc('\n', pipe_fp); // send the endline character
        }
    }
    pclose(pipe_fp); // close the pipe
    return 0;
}
```

Unnamed pipe

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

- Create a message pipe
  - Anything can be written to the pipe, and read from the other end
  - Data is received in the order it was sent
  - OS enforces mutual exclusion: only one process at a time
  - Accessed by a file descriptor
  - Processes sharing the pipe must have the same parent
- Returns a pair of file descriptors (
  - `pfds[0]` is the read end
  - `pfds[1]` is the write end

Pipe example – Necessary header files

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    // The main program for pipes
}
```

Pipe example (Child => Parent)


```
int main()
{
  int pfds[2];
  char buf[30];
  pipe(pfds); /* Create a message pipe*/
  pid_t pid = fork(); /* 0 (child), non-zero (parent) */
  if ( pid == 0 ) {
    printf("CHILD: writing to pipe\n");
    close(pfds[0]);
    write(pfds[1], "test", 5);
    printf("CHILD: exiting\n");
  } else {
    printf("PARENT: reading from pipe\n");
    close(pdfs[1]);
    read(pfds[0], buf, 5);
    wait(NULL); /* Wait until the child returns*/
    printf("PARENT: read \"%s\"\n", buf);
}
  return 0;
}
```

Pipe example (Parent => Child)


```
int main()
{
  int pfds[2];
  char buf[30];
  pipe(pfds); /* Create a message pipe*/
  pid_t pid = fork(); /* 0 (child), non-zero (parent) */
  if ( pid != 0 ) {
    printf("PARENT: writing to pipe\n");
    close(pfds[0]);
    write(pfds[1], "test", 5);
    wait(NULL); /* Wait until the child returns*/
    printf("PARENT: exiting\n");
} else {
    printf("CHILD: reading from pipe\n");
    close(pdfs[1]);
    write(pfds[0], buf, 5);
    printf("CHILD: read \"%s\"\n", buf);
}
  return 0;
}
```

Duplicating a file descriptor

```
#include <unistd.h>
int dup(int oldfd);
```

- Create a copy of an open file descriptor: put new copy in first unused file descriptor
- Returns
  - Return value >= 0: success, returns new file descriptor
  - Return value = -1: error, check value of **`errno`**
- Parameters
  - **`oldfd`**: the open file descriptor to be duplicated

Example: A command-line pipe

ls | wc -l

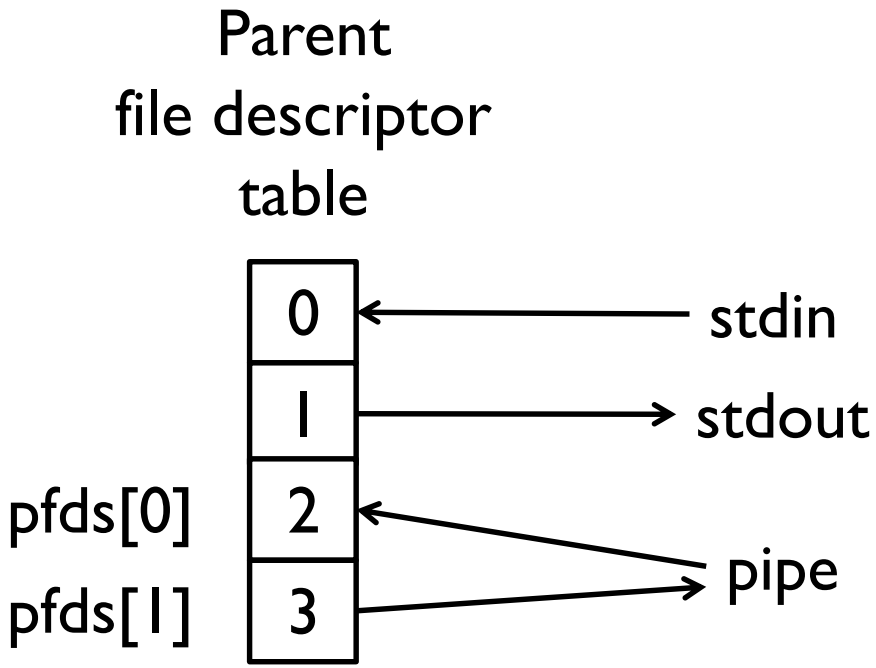
Can we implement a command-line pipe with `pipe()`?

How do we attach the stdout of **ls** to the stdin of **wc**?

Command-line pipe: ls | wc -l

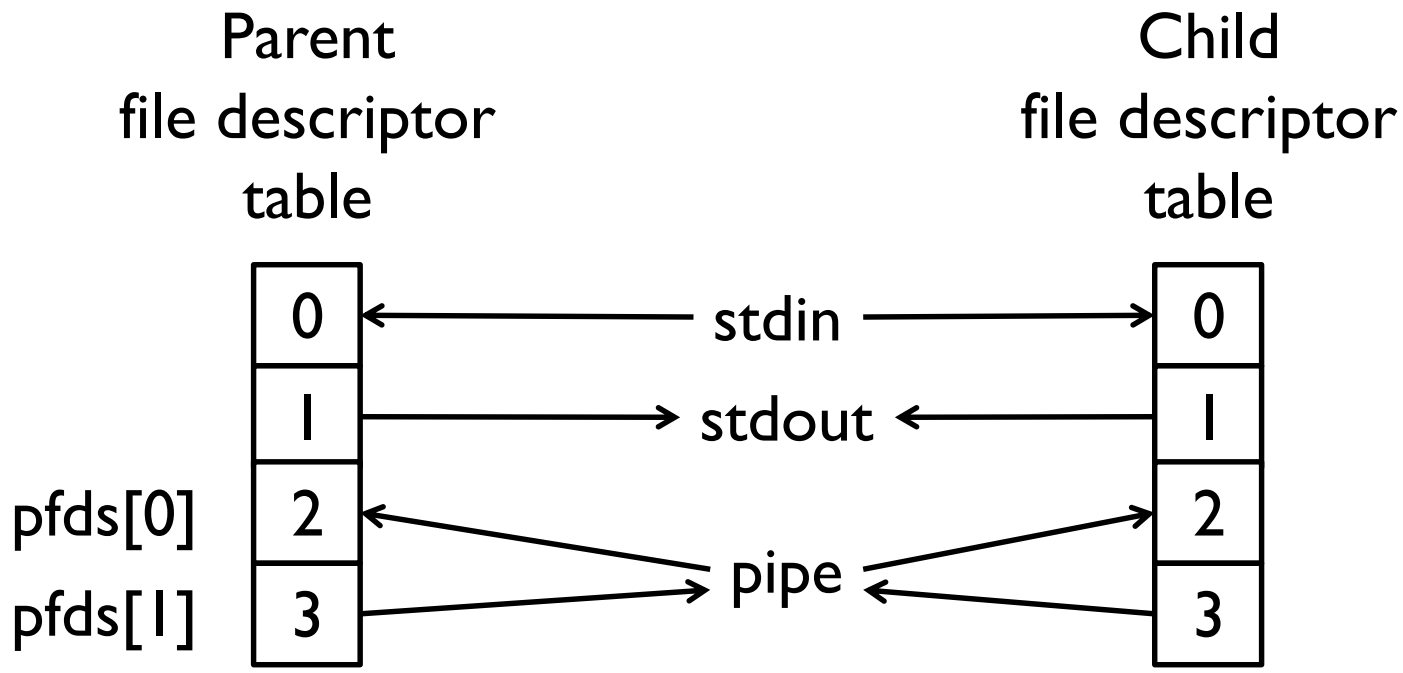
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
  int pfds[2];
  pipe(pfds);
  pid_t pid = fork(); /* 0 (child), non-zero (parent) */
  if ( pid == 0 ) { /* The child process*/
    close(1); /* close stdout */
    dup(pfds[1]); /* make stdout as pipe input */
    close(pfds[0]); /* don't need this */
    execlp("ls", "ls", NULL);
  } else { /* The parent process*/
    close(0); /* close stdin */
    dup(pfds[0]); /* make stdin as pipe output */
    close(pfds[1]); /* don't need this */
    wait(0); /* wait for the child process */
    execlp("wc", "wc", "-l", NULL);
  }
  return 0;
}
```

Command-line pipe: ls | wc -l



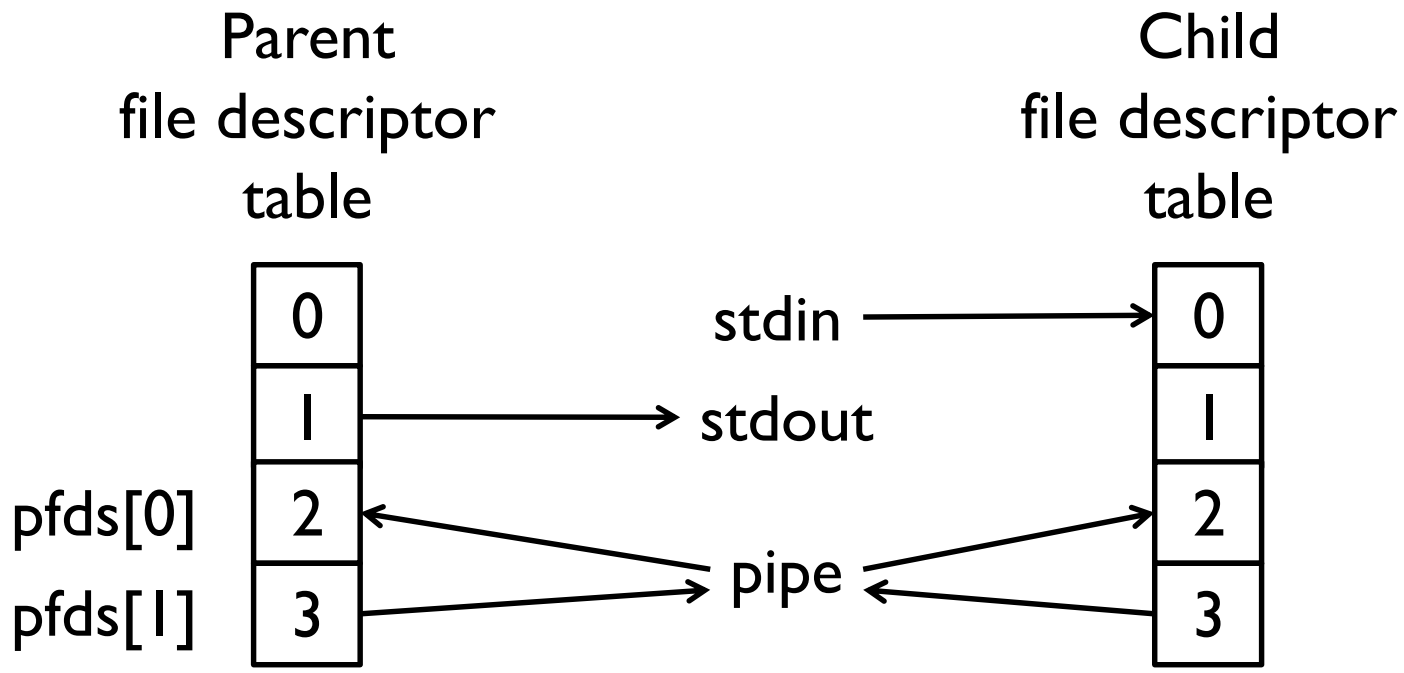

```
pipe(pfds);
```

Command-line pipe: ls | wc -l




```
pipe(pfds);
fork();
```

Command-line pipe: ls | wc -l

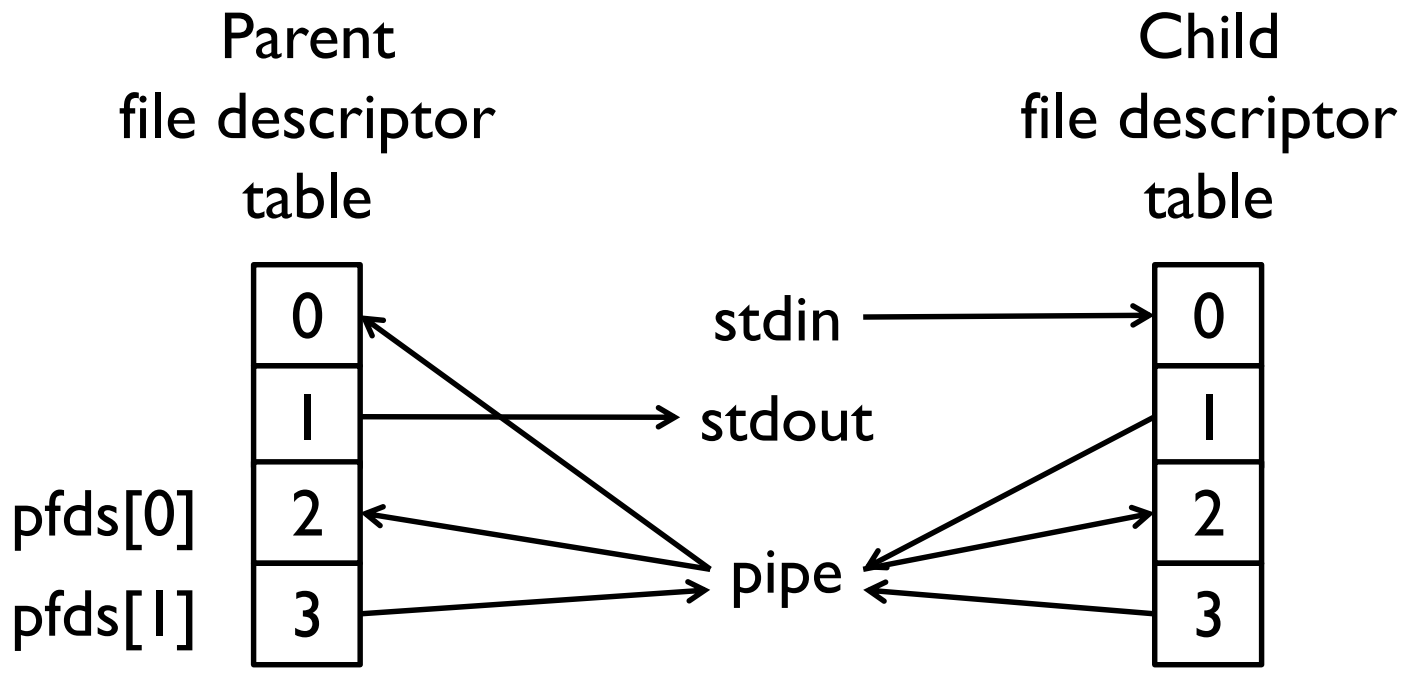



```
pipe(pfds);
fork();

close(0);
```

```
close(1);
```

Command-line pipe: ls | wc -l

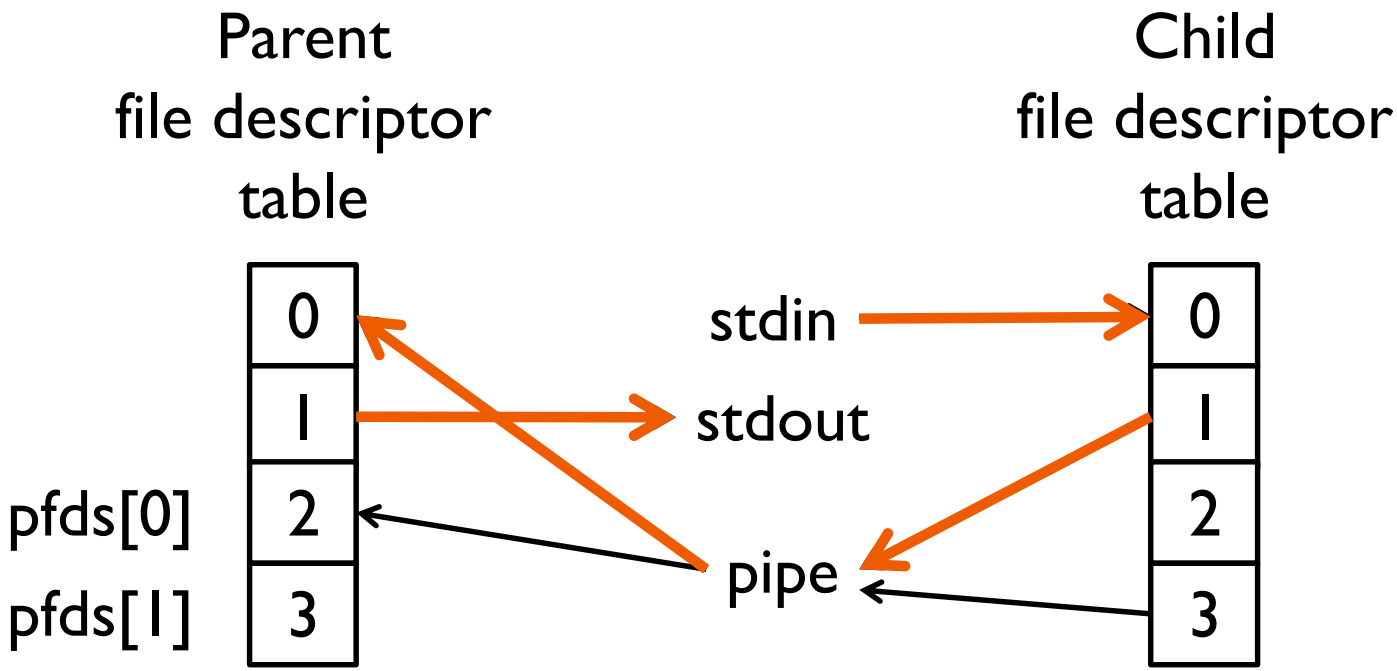



```
pipe(pfds);
fork();

close(0);
dup(pfds[0]);
```

```
close(1);
dup(pfds[1]);
```

Command-line pipe: ls | wc -l




```
pipe(pfds);
fork();

close(0);
dup(pfds[0]);
close(pfds[1]);
execlp("wc", "wc", "-l", NULL);
```

```
close(1);
dup(pfds[1]);
close(pfds[0]);
execlp("ls", "ls", NULL);
```

Duplicating a file descriptor

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

- Create a copy of an open file descriptor: put new copy in specified location (…after closing **`newfd`**, if it was open)
- Returns
  - Return value >= 0: success, returns new file descriptor
  - Return value = -1: error, check value of **`errno`**
- Parameters
  - **`oldfd`**: the open file descriptor to be duplicated

Command-line pipe: ls | wc –l (using dup2)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main() {
  int pfds[2];
  pipe(pfds);
  pid_t pid = fork(); /* 0 (child), non-zero (parent) */
  if ( pid == 0 ) { /* The child process*/
    close(1); /* close stdout */
    dup2(pfds[1],1); /* make stdout as pipe input */
    close(pfds[0]); /* don't need this */
    execlp("ls", "ls", NULL);
  } else { /* The parent process*/
    close(0); /* close stdin */
    dup2(pfds[0],0); /* make stdin as pipe output */
    close(pfds[1]); /* don't need this */
    wait(0); /* wait for the child process */
    execlp("wc", "wc", "-l", NULL);
  }
  return 0;
}
```

Lab exericse: Add pipe support to the simple shell

- Based on the simple shell exercise (`myshell_skeleton.c`) in the previous lab, implement a pipe operation:
  - `myshell> [command A] | [command B]`
- In the current stage, you are NOT required to support multi-level pipe as follows:
  - `myshell> [command A] | [command B] | [command C]`

FIFOs

FIFOs

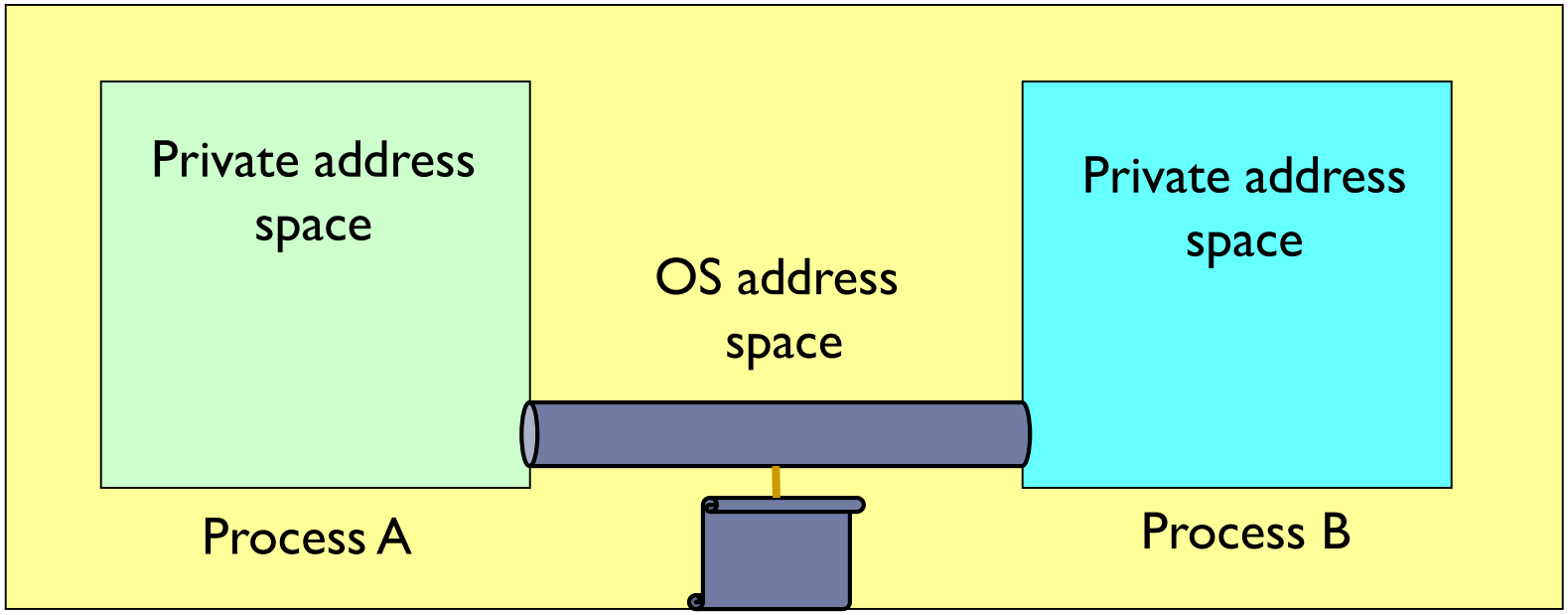
- A pipe disappears when no process has it open
- FIFOs = named pipes
  - Special pipes that persist even after all processes have closed them
  - Actually implemented as a file

```
#include <sys/types.h>
#include <sys/stat.h>

/* Somewhere inside your main program… */
int status;
status = mkfifo("/home/cnd/mod_done",
    /* mode=0644 as the second parameter */
  S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH);
```

Communication over of FIFO

- First **open** blocks until second process opens the FIFO
- Can use **O_NONBLOCK** flag to make operations non-blocking
- FIFO is persistent: can be used multiple times
- Like pipes, OS ensures atomicity of writes and reads



FIFOs

- It can be used between unrelated processes
- When a FIFO is created, it must be opened with syscall **`open()`** or stream operations such as **`fopen()`**, but you cannot use **`lseek()`**
- You cannot open FIFO with "read+write" mode
- It has to be open at both ends simultaneously before you can proceed to do any input or output operations on it

```
/* server */
writefd = open(FIFO1, O_WRONLY);
readfd = open(FIFO2,  O_RDONLY);

/* client */
readfd = open(FIFO1,  O_RDONLY);
writefd = open(FIFO2, O_WRONLY);
```

Can we change the order of open sequences?

FIFO example: producer-consumer

- Producer:
  - Writes to FIFO
- Consumer:
  - Reads from FIFO
  - Outputs data to file
- FIFO ensures atomicity of write

FIFO example

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
int main (int argc, char *argv[])
{
  int requestfd;

  if (argc != 2) {
    /* name of consumer fifo on the command line */
    fprintf(stderr, "Usage: %s fifoname > logfile\n", argv[0]);
    return 1;
  }

   /* Note: Continue in the next slide! */
```

FIFO example

```
/* create a named pipe to handle incoming requests */
if ((mkfifo(argv[1], S_IRWXU | S_IWGRP| S_IWOTH) == -1)
      && (errno != EEXIST)) {
  perror("Server failed to create a FIFO");
  return 1;
}

/* open a read/write communication endpoint to the pipe */
 if ((requestfd = open(argv[1], O_RDWR)) == -1) {
  perror("Server failed to open its FIFO");
  return 1;
}

/* The remaining part of the program
   Write to pipe like you would to a file */
```