OPerating Systems Laboratory
OPS*Lab*

Programming in the UNIX Environment

(Part 4.2.3)

Prof. Marco Autili

University of L'Aquila

What we have done so far

Chapter 1
Chapter 3
Chapter 4

Introduction

File I/O

Files and Directories

Chapter 5

Standard I/O Library

Chapter 7

Process Environment

- main function
- Process termination
- Environment list

Chapter 8

Process Control

- Creation of new processes
- Program execution
- Process termination

Chapter 10

Signals

- kill
- raise
- alarm
- pause

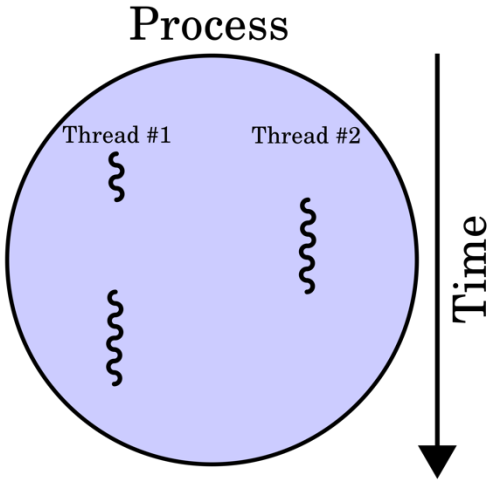
What we are going to do

Chapter 11
Threads

Chapter 15
Interprocess Communication (IPC)

Threads

- A typical "UNIX-like" process can be thought of as having a single thread of control: each process is doing only one thing at a time
- Multiple *threads of control* (or simply *threads*) can be used to perform multiple tasks within the environment of a single process
- At the programming level, all threads within a single process have access to the same process components, such as file descriptors and memory address space
- With multiple threads of control, we can design our programs to do more than one thing at a time within a single process, with each thread handling a separate task



- A single-threaded process with multiple tasks to perform implicitly serializes those tasks, because there is only one thread of control
- With multiple threads of control, the processing of independent tasks can be interleaved by simply assigning a separate thread per task
    - two tasks can be interleaved, without the need of performing any synchronization of the related threads, only if they do not depend on the processing performed by each other

Threads VS Processes

Process	Thread
A process is heavyweight	A thread is a lightweight process also called an LWP
A process has its own memory	A thread shares the memory with the parent process and other threads within the process
Inter-process communication is slower due to isolated memory	Inter-thread communication is faster due to shared memory
Context switching between processes is expensive due to saving old and loading new process memory and stack info	Context switching between threads is less expensive due to shared memory
An application with several processes for its components can provide better memory utilization when memory is scarce. We can assign low priority to inactive processes in the application. This idle process is then eligible to be swapped to disk. This keeps the active components of the application responsive	When memory is scarce, the multi-threaded application does not provide any provision to manage memory

Programming model VS systems

- Some people associate multithreaded **programming** with multiprocessor or multicore **systems**
- However, the benefits of a multithreaded **programming model** can be realized even if your program is running on a uniprocessor system. As long as your program has to block when serializing tasks, you can still see improvements in response time and throughput when running on a uniprocessor, because some threads might be able to run while others are blocked
- The following definition from Wikipedia better clarifies this aspect

> In computer architecture, multithreading is the ability of **a** central processing unit (CPU) (or a single core in a multi-core processor) to provide multiple threads of execution concurrently, supported by the **operating system**

- This approach differs from multiprocessing. In a multithreaded application, the threads share the resources of a single or multiple cores, which include the computing units, the CPU caches, and the translation lookaside buffer (TLB)
- While multiprocessing systems include multiple complete processing units in one or more cores, multithreading aims to increase utilization of a single core by using thread-level parallelism, as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multithreading CPUs and with CPUs with multiple multithreading cores

https://en.wikipedia.org/wiki/Multithreading_(computer_architecture)

Pthreads

POSIX Threads, usually referred to as pthreads, is an execution model that exists independently from a language, as well as a parallel execution model

- It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API

- POSIX Threads is an API defined by the standard POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)

- Implementations of the API are available on many Unix-like POSIX-conformant operating systems such as FreeBSD, NetBSD, OpenBSD, Linux, macOS, Android, Solaris, Redox, and AUTOSAR Adaptive, typically bundled as a library libpthread

- DR-DOS and Microsoft Windows implementations also exist: within the SFU/SUA subsystem , which provides a native implementation of a number of POSIX APIs, and also within third-party packages such as pthreads-w32, which implements pthreads on top of existing Windows API

https://en.wikipedia.org/wiki/POSIX_Threads

Thread Identification

- Just as every process has a process ID, every thread has a thread ID
  - unlike the process ID, which is unique in the system, the thread ID has significance only within the context of the process to which it belongs
- Implementations are allowed to use a structure to represent the pthread_t data type, so portable implementations can't treat them as integers
  - the pthread_equal() function must be used to compare two thread IDs
  - a thread can obtain its own thread ID by calling the pthread_self() function

```
#include <pthread.h>

int pthread_equal(pthread_t tid1, pthread_t tid2);
```

Returns: nonzero if equal, 0 otherwise

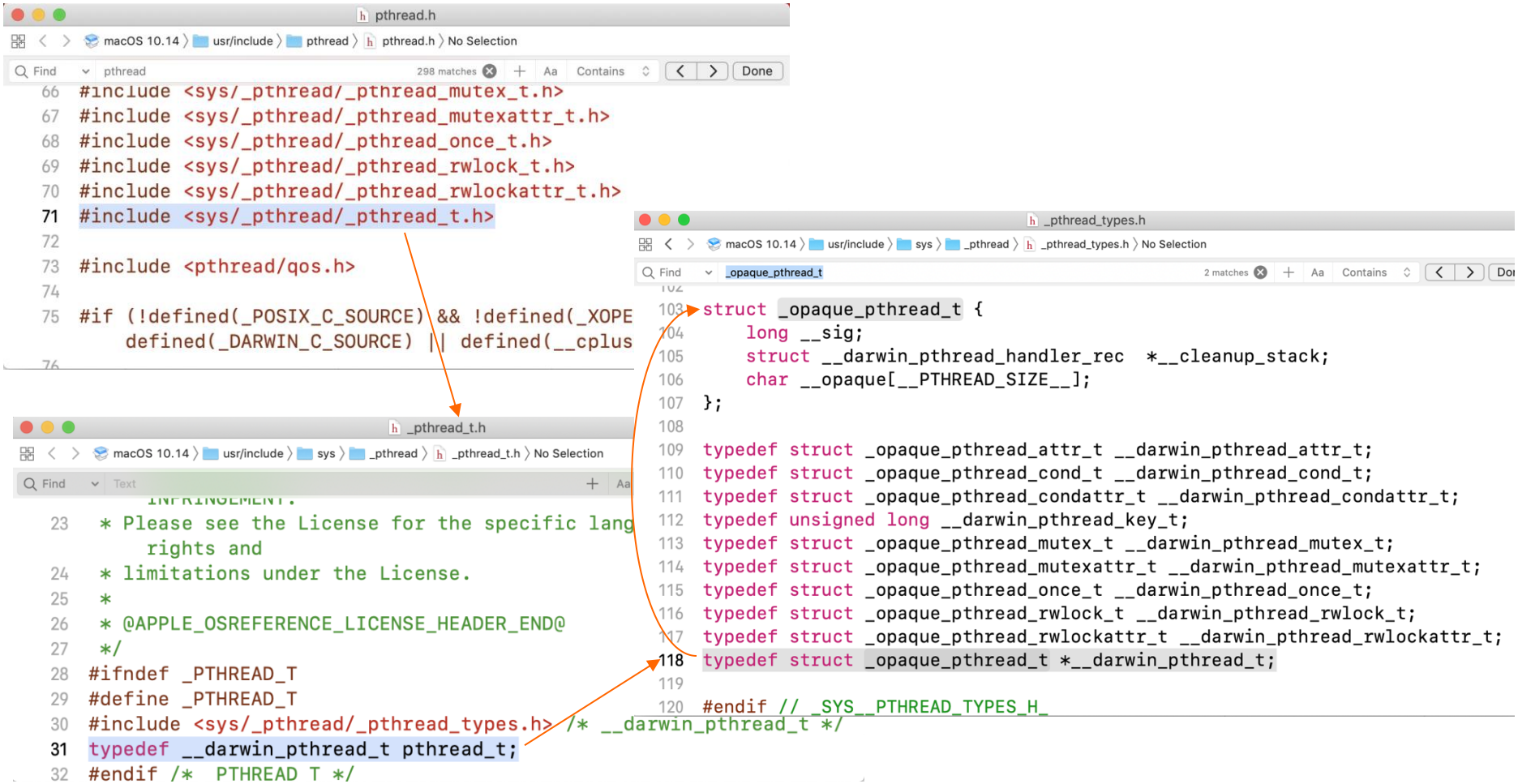
```
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns: the thread ID of the calling thread

Finding pthread_t

In my system, the header file <pthread.h> is in the folder:

/Applications/Xcode.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX.sdk/usr/include/pthread



Work queue example

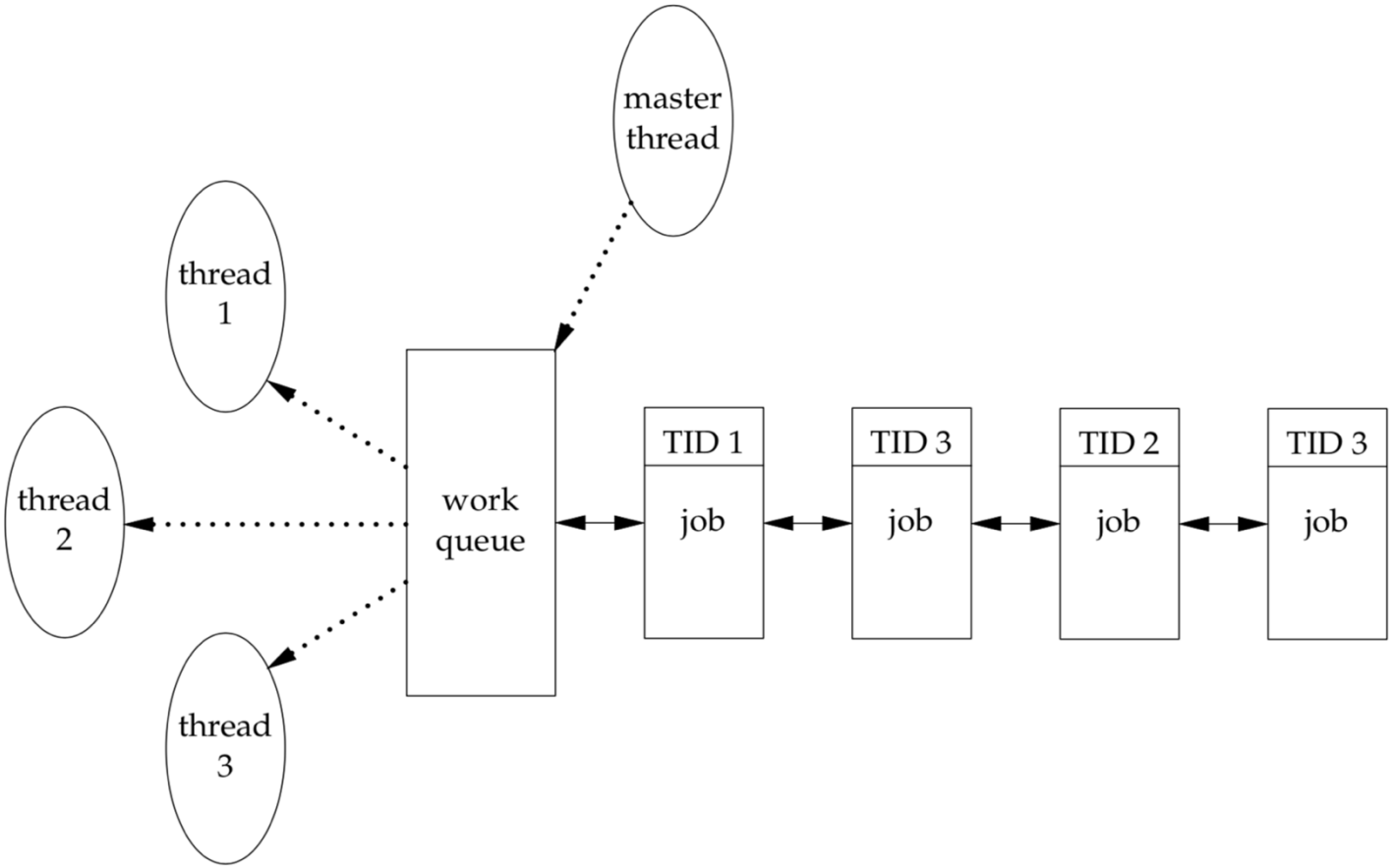


Figure 11.1 Work queue example

Thread Creation

```
#include <pthread.h>

int pthread_create(pthread_t *restrict tidp,
                   const pthread_attr_t *restrict attr,
                   void *(*start_rtn)(void *), void *restrict arg);
```

Returns: 0 if OK, error number on failure

- The memory location pointed to by *tidp* is set to the thread ID of the newly created thread when pthread_create() returns successfully
  - The *attr* argument is used to customize various thread attributes (more on that in Section 12.3) but for now, we'll set this to NULL to create a thread with the default attributes
- The newly created thread starts running at the address of the *start_rtn* function. This function takes a single argument, *arg*, which is a typeless pointer
  - If you need to pass more than one argument to the *start_rtn* function, then you need to store them in a structure and pass the address of the structure in *arg*
- When a thread is created, there is no guarantee which will run first: the newly created thread or the calling thread

Thread Termination

A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine. The return value is the thread's exit code.
2. The thread can be canceled by another thread in the same process.
3. The thread can call `pthread_exit`.

call to pthread_cancel()

```
#include <pthread.h>

void pthread_exit(void *rval_ptr);
```

The *rval_ptr* argument is a typeless pointer, similar to the single argument passed to the start routine. This pointer is available to other threads in the process by calling the `pthread_join` function.

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **rval_ptr);

                                        Returns: 0 if OK, error number on failure
```

The calling thread will block until the specified thread calls `pthread_exit`, returns from its start routine, or is canceled. If the thread simply returned from its start routine, *rval_ptr* will contain the return code. If the thread was canceled, the memory location specified by *rval_ptr* is set to `PTHREAD_CANCELED`.

Printing the thread ID

- Although there is no portable way to print the thread ID, the test program chap11-threads/threadid.c is useful to gain some insight into how threads work

- The program chap11-threads/threadid.c creates one thread and prints the process and thread IDs of the new thread and the initial thread

- This example has two peculiarities, which are necessary to handle races between the main thread and the new thread:

    1. The first is the need to sleep in the main thread. If it doesn't sleep, the main thread might exit, thereby terminating the entire process before the new thread gets a chance to run

    2. It is not by chance (and it should not surprise) that the new thread obtains its thread ID by calling pthread_self() instead of reading it out of shared memory or receiving it as an argument to its thread-start routine...

Thread functions VS process functions

- Similarities between the thread functions and the process functions

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

Play with the examples in chap11-threads

What next?

Chapter 11
Threads

Chapter 15
Interprocess Communication (IPC)

IPC

- "Up to now", the only way for processes to exchange information is by passing open files across a fork or an exec or through the file system

- We'll now describe other techniques for processes to communicate with one another

InterProcess Communication (IPC)

- IPC restricted to processes on the same host
  - pipes, FIFOs, message queues, semaphores, shared memory

- IPC between processes on the same host or on different hosts
  - sockets

PIPEs

Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems

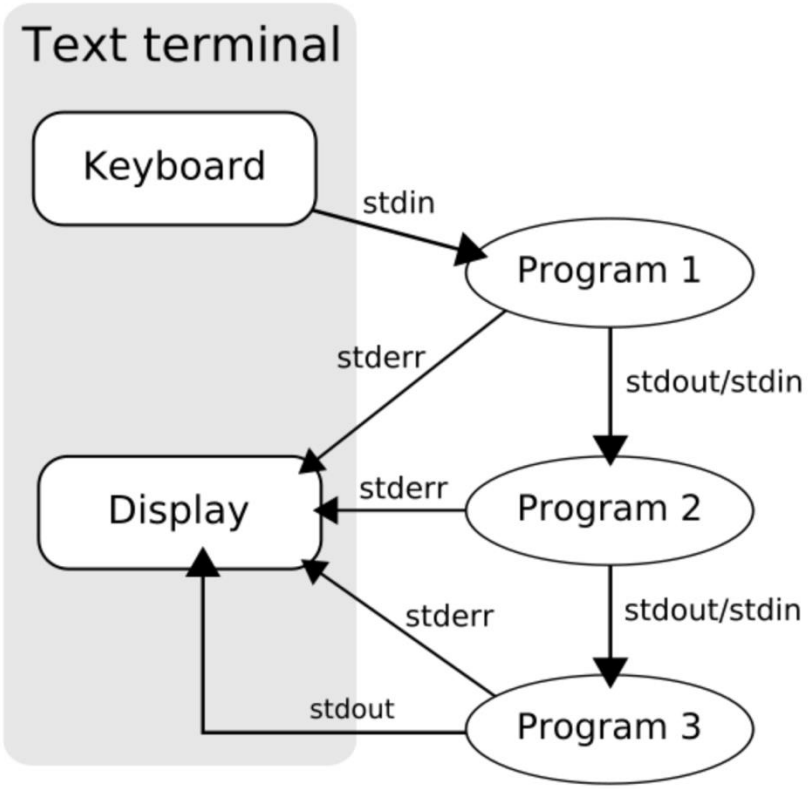
- Pipes have two limitations
  1. Historically, they have been half duplex (i.e., data flows in only one direction). Some systems now provide full-duplex pipes, but for maximum portability, we should never assume that this is the case
  2. Pipes can be used only between processes that have a common ancestor. Normally, a pipe is created by a process, that process calls fork, and the pipe is used between the parent and the child
- FIFOs (Section 15.5) get around the second limitation, and UNIX domain sockets (Section 17.2) get around both limitations.
  - In other words, a *FIFO is a special file* similar to a pipe, but instead of being an anonymous and "temporary connection", a FIFO has a name or names like any other file. Processes open the FIFO by name in order to communicate through it.
- Despite the above limitations, half-duplex pipes are still the most commonly used form of IPC
  - Remember that every time you type a sequence of commands in a pipeline for the shell to execute it, the shell creates a separate process for each command and links the standard output of one process to the standard input of the next using a pipe

Pipeline (from Part 2)

In Unix-like computer operating systems, a pipeline is a sequence of processes chained together by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) to the next one

The exit status of a pipeline is the exit status of the last process in the pipeline

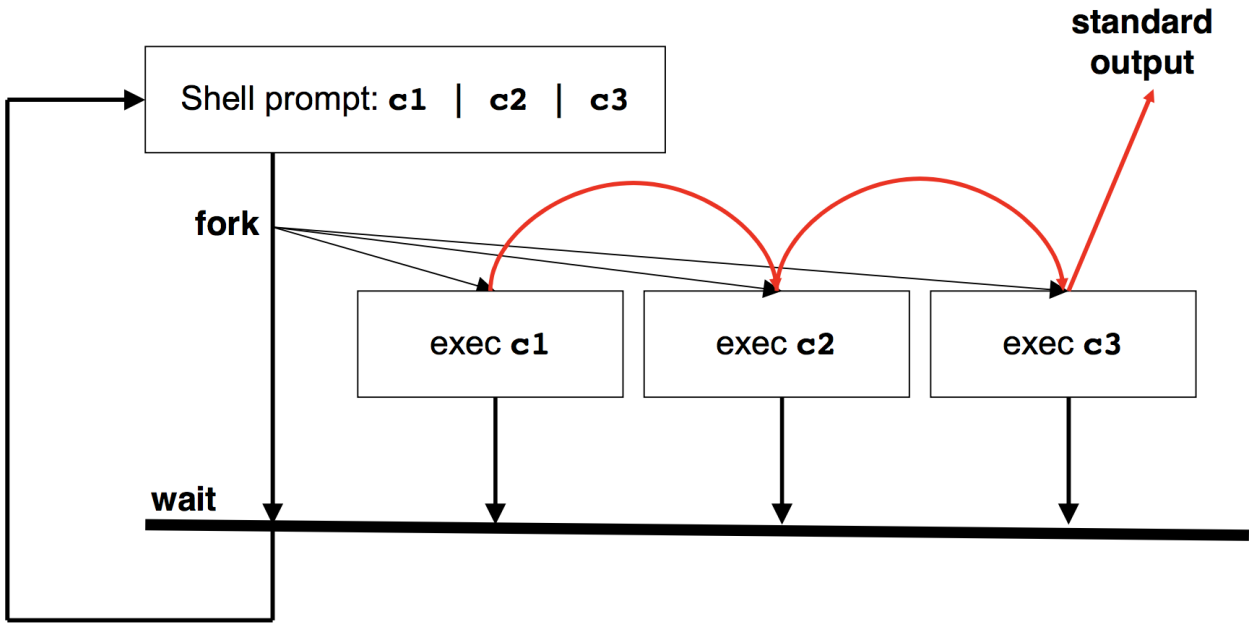
The shell waits for the termination of all the processes in the pipeline before returning the prompt



https://en.wikipedia.org/wiki/Pipeline_(Unix)

Pipeline execution order (from Part 2)

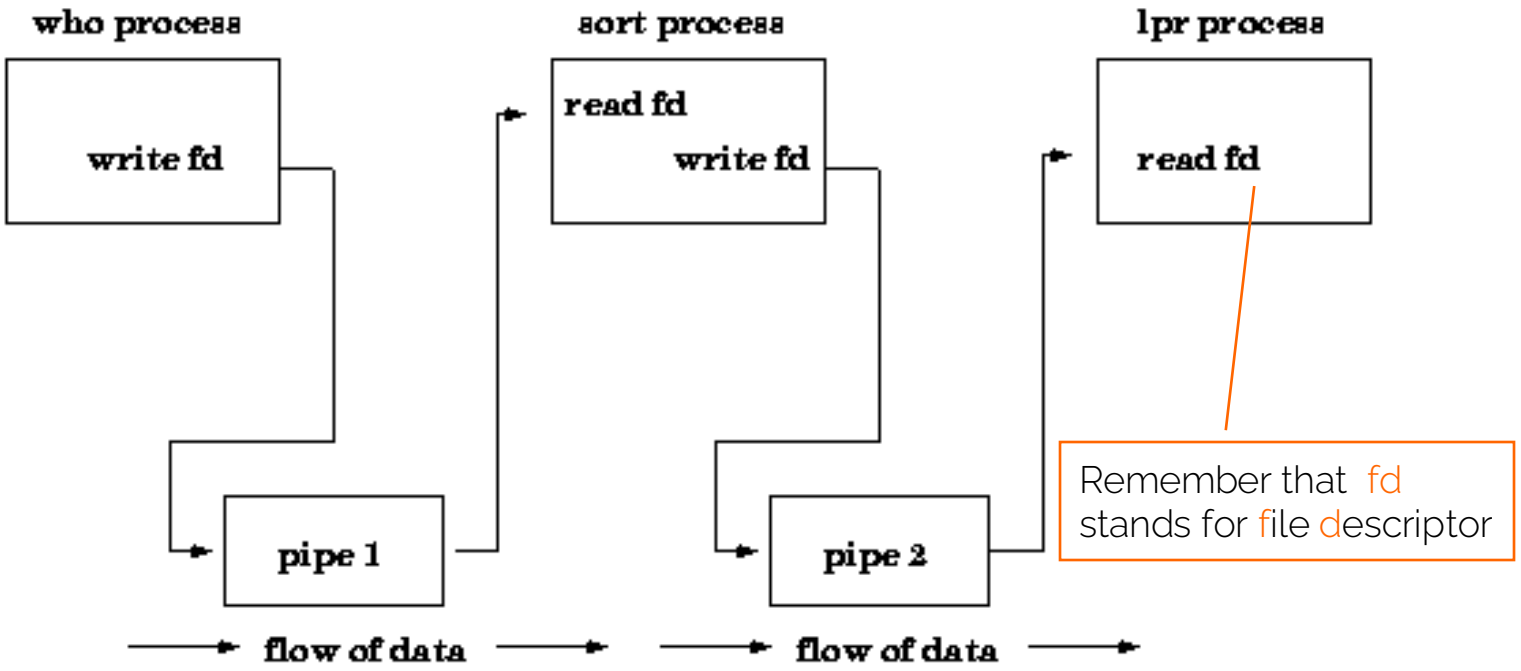
- The order the processes are run does not matter and is not guaranteed
- The shell first creates the pipe, the conduit for the data that will flow between the processes, and then creates the processes with the ends of the pipe connected to them
- For example, the first process that is run may block waiting for input from the second process, or block waiting for the second process to start reading data from the pipe. These waits can be arbitrarily long and do not matter
- Whichever order the processes are run, the data eventually gets transferred and everything works



Pipeline (from Part 2)

Example: who | sort | lpr

A pipe to send to the default printer (lpr) the ordered (sort) list of all users currently logged on (who)



pipe function

```
#include <unistd.h>

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

Returns: 0 if OK, –1 on error

Two file descriptors are returned through the *fd* argument: *fd[0]* is open for reading, and *fd[1]* is open for writing. The output of *fd[1]* is the input for *fd[0]*.

Originally in 4.3BSD and 4.4BSD, pipes were implemented using UNIX domain sockets. Even though UNIX domain sockets are full duplex by default, these operating systems hobbled the sockets used with pipes so that they operated in half-duplex mode only.

POSIX.1 allows for implementations to support full-duplex pipes. For these implementations, *fd[0]* and *fd[1]* are open for both reading and writing.

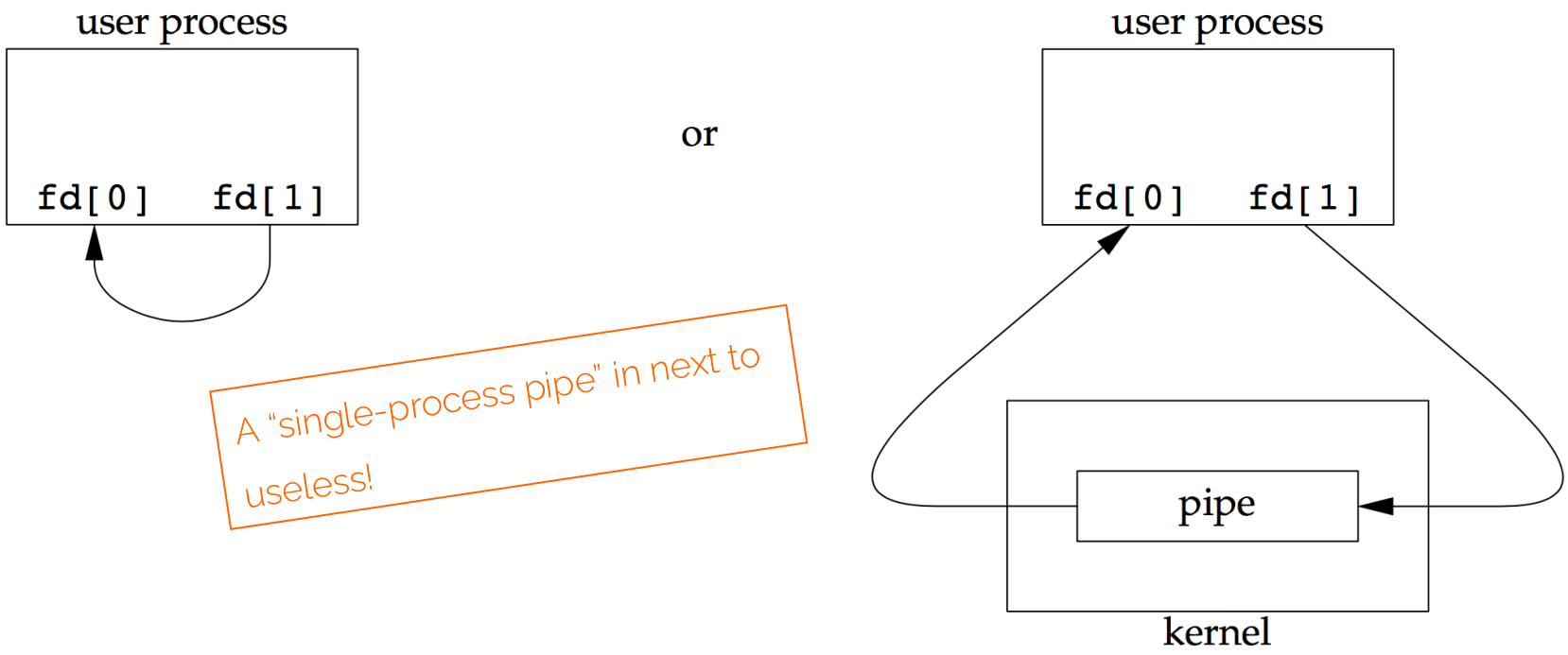
The `fstat` function (Section 4.2) returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.

POSIX.1 states that the `st_size` member of the `stat` structure is undefined for pipes. But when the `fstat` function is applied to the file descriptor for the read end of the pipe, many systems store in `st_size` the number of bytes available for reading in the pipe. This is, however, nonportable.

Half-duplex pipe

Two ends of the pipe connected in a single process…

… emphasizing that the data in the pipe flows through the kernel



A "single-process pipe" in next to useless!

Figure 15.2 Two ways to view a half-duplex pipe

Half-duplex pipe

Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child, or vice versa

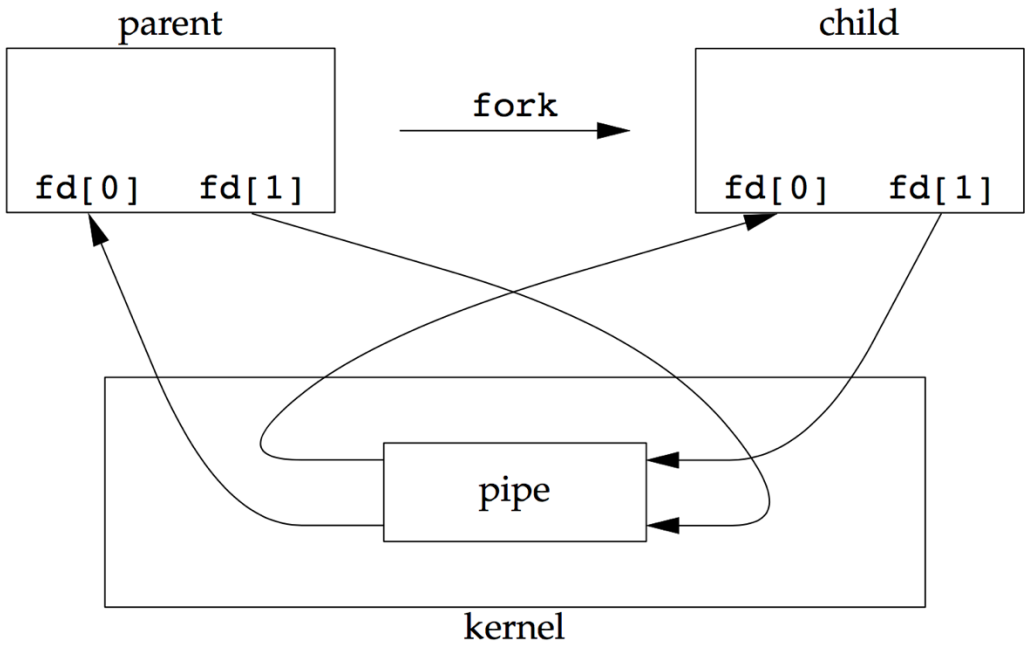


Figure 15.3 Half-duplex pipe after a `fork`

Half-duplex pipe: parent2child

- What happens after the fork depends on which direction of data flow we want
- For a pipe from the parent to the child:
  - the parent closes the read end of the pipe (fd[0])
  - the child closes the write end (fd[1])

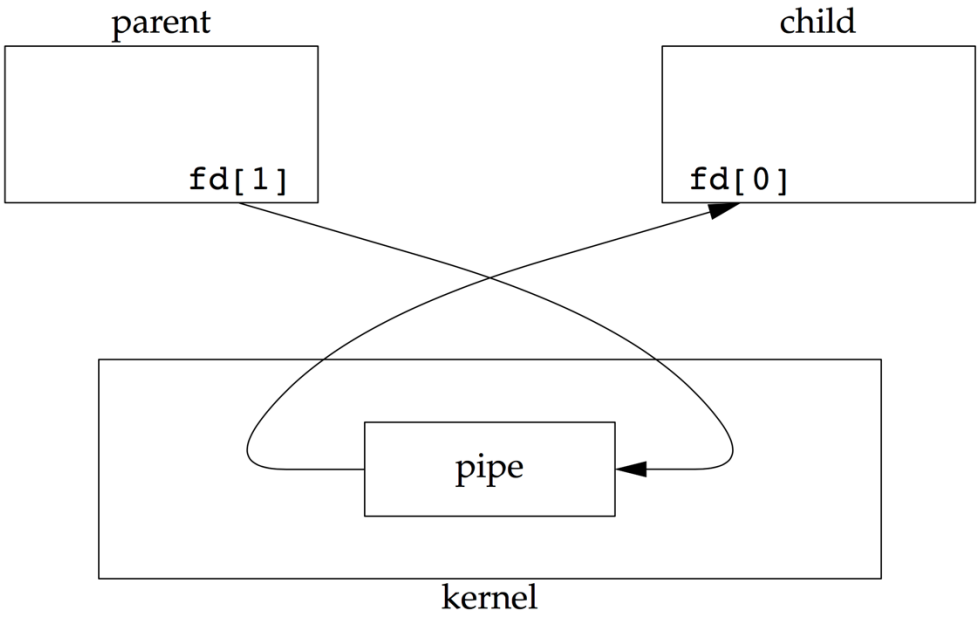


Figure 15.4 Pipe from parent to child

PIPEs: basic rules

When both ends of a pipe are open, two rules apply

1. If a process attempts to read from an empty pipe, then read() will block until data is available
2. If a process attempts to write to a full pipe (see below), then write() blocks until sufficient data has been read from the pipe to allow the write to complete
   - Nonblocking I/O is possible by using the fcntl() F_SETFL operation to enable the O_NONBLOCK open file status flag

I/O on pipes *VS* I/O on FIFOs

- The only difference between pipes and FIFOs is the manner in which they are created and opened
- Once these tasks have been accomplished, I/O on pipes and FIFOs has exactly the same semantics

PIPEs: basic rules

When one end of a pipe is closed, two rules apply

3. If we read from a pipe whose write end has been closed, read returns 0 after all the data has been read to indicate an end-of-file
   - Since it is possible to duplicate a pipe descriptor so that multiple processes have the pipe open for writing, technically, we should say that the end-of-file is not generated until there are no more writers for the pipe
   - In most cases, however, there is a single reader and a single writer for a pipe

4. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated
   4. If we either ignore the signal or catch it and return from the signal handler, write returns –1 with *errno* set to EPIPE

PIPEs: basic rules

- When we are writing to a pipe (or FIFO), the constant PIPE_BUF specifies the kernel's pipe buffer size

- PIPE_BUF specifies the maximum amount of data that can be written atomically. Thus, a write of PIPE_BUF bytes or less will not be interleaved with the writes from other processes to the same pipe (or FIFO).

- But if multiple processes are writing to a pipe (or FIFO), and if we write more than PIPE_BUF bytes, the data might be interleaved with the data from the other writers

- We can determine the value of PIPE_BUF by using pathconf or fpathconf (see Section 2.5.4 and Figure 2.12)

PIPEs example

pipe1.c

```c
#include "apue.h"

int
main(void)
{
    int     n;
    int     fd[2];
    pid_t   pid;
    char    line[MAXLINE];

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Line 1, Column 1 Tab Size: 4

What's the output?

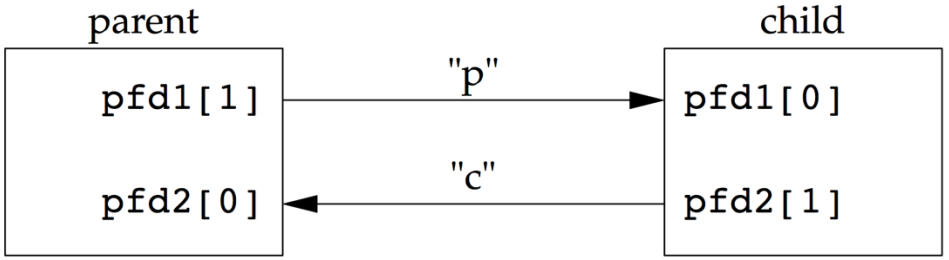
Who (parent or child) is doing what?

chap15-ipc1 — -bash — 61×8

```
desktop-jisqlks:chap15-ipc1 marcoautili$
desktop-jisqlks:chap15-ipc1 marcoautili$
desktop-jisqlks:chap15-ipc1 marcoautili$
desktop-jisqlks:chap15-ipc1 marcoautili$
desktop-jisqlks:chap15-ipc1 marcoautili$
desktop-jisqlks:chap15-ipc1 marcoautili$ ./pipe1
hello world
desktop-jisqlks:chap15-ipc1 marcoautili$
```

Back to “avoiding race conditions”

- Solving race conditions by using a two pipes for parent–child synchronization (see next slide)
- The parent writes the character “p” across the top pipe when TELL_CHILD is called
- The child writes the character “c” across the bottom pipe when TELL_PARENT is called
- The corresponding WAIT_xxx functions do a blocking read for the single character



- Note that each pipe has an extra reader, which doesn’t matter. That is, in addition to the child reading from pfd1[0], the parent has this end of the top pipe open for reading. This doesn’t affect us, since the parent doesn’t try to read from this pipe

Avoiding race conditions

tellwait2.c

```c
#include "apue.h"

static void charatatime(char *);

static int  pfd1[2], pfd2[2];

/* Routines to let a parent and child synchronize */

void TELL_WAIT (void);
void TELL_PARENT (pid_t pid);
void WAIT_PARENT (void);
void TELL_CHILD (pid_t pid);
void WAIT_CHILD (void);

int
main(void)
{
    pid_t   pid;
    int i,j;

    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {
        WAIT_PARENT();      /* parent goes first */

        for (i = 0; i < 1000; i++) {
            charatatime("\n output from child \n");
        }
    } else {
        for (j = 0; j < 1000; j++) {
            charatatime("\n output from parent \n");
        }
        TELL_CHILD(pid);
    }
    exit(0);
}
```

Line 1, Column 1 Spaces: 4 C

Avoiding race conditions by using a pipe-based implementation of TELL_XXX and WAIT_XXX

chap8-proc — -bash — 40×55

```
 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from desktop-jisqlks:chap8-proc
marcoautili$ child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child

 output from child
```

The parent went first… and here it terminated its execution…

Uncomment the two printf() in the code and see what happens ☺

?

popen and pclose functions

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
                                    Returns: file pointer if OK, NULL on error

int pclose(FILE *fp);
                        Returns: termination status of cmdstring, or –1 on error
```

- Since a common operation is to create a pipe to another process to either read ("r") its output or send ("w") it input, the standard I/O library has historically provided the popen and pclose functions (Section 15.3)
- These two functions handle all the dirty work that we have been doing ourselves:
    1. creating a pipe
    2. forking a child
    3. closing the unused ends of the pipe
    4. executing a shell to run the command
    5. waiting for the command to terminate

popen Function

- The popen function
  - does a fork
  - performs an exec to execute the *cmdstring*
  - returns a standard I/O file pointer

-c string

Remember that:

- If the -c option is present, then commands are read from a string
- If there are arguments, they are assigned to the positional parameters, starting with $0

- If *type* is "r", the file pointer is connected to the "standard output of *cmdstring*"
- The *cmdstring* is executed by the Bourne shell, as in /bin/sh -c *cmdstring*

- This means that the shell expands any of its special characters in *cmdstring*. This allows us to say, for example,

```
fp = popen("ls *.c", "r");

fp = popen("cmd 2>&1", "r");
```

Figure 15.9 Result of `fp = popen(`*cmdstring*`, "r")`

popen Function

- If *type* is "w", the file pointer is connected to the "standard input of *cmdstring*"

- One way to remember the final argument to popen is to remember that, like fopen, the returned file pointer is readable if *type* is "r" or writable if *type* is "w"

Figure 15.10 Result of `fp = popen(`*cmdstring*`, "w")`

- For example:

```
fp = popen("more", "w");
```

pclose Function

- The pclose function
  - closes the standard I/O stream
  - waits for the command to terminate
  - returns the termination status of the shell

    (recall that the system function, previously described, also returns the termination status)

Play with the example
chap15-ipc1/file2pager.c

FIFOs

FIFOs are sometimes called named pipes

- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe
- either parent and child or a common parent sets up the pipe and passes its file descriptors to its individual (descendant) processes

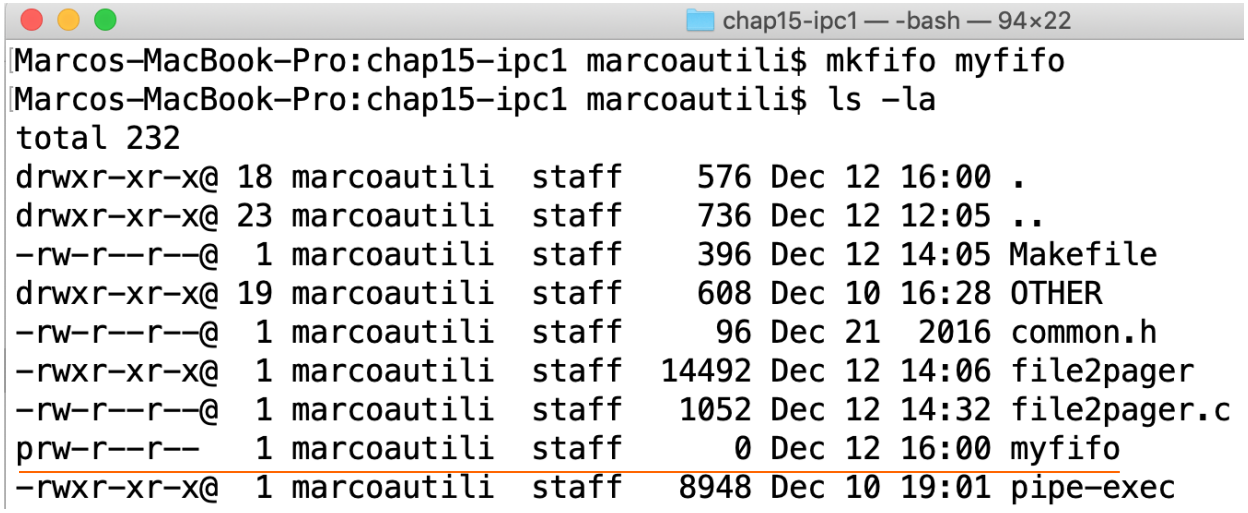
- With FIFOs also unrelated processes can exchange data
  - A FIFO is different because it has a name like a file
  - Since it exists in the filesystem, it is not necessary for the processes to be related to communicate using the FIFO

- Thus, since FIFO is a type of file, recall that
  - one of the encodings of the st_mode member of the stat structure (Section 4.2) indicates that a file is a FIFO
  - we can test for this with the S_ISFIFO macro

Creating a FIFO (from the shell)

Creating a FIFO is similar to creating a file

- A file of type "FIFO" identified with "p" (named pipe) is actually created in the file system



Creating a FIFO (programmatically)

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

int mkfifo(const char *path, mode_t mode);

int mkfifoat(int fd, const char *path, mode_t mode);
```

Both return: 0 if OK, –1 on error

The specification of the *mode* argument is the same as for the `open` function (Section 3.3). The rules for the user and group ownership of the new FIFO are the same as we described in Section 4.6.

The `mkfifoat` function is similar to the `mkfifo` function, except that it can be used to create a FIFO in a location relative to the directory represented by the *fd* file descriptor argument. Like the other `*at` functions, there are three cases:

1. If the *path* parameter specifies an absolute pathname, then the *fd* parameter is ignored and the `mkfifoat` function behaves like the `mkfifo` function.
2. If the *path* parameter specifies a relative pathname and the *fd* parameter is a valid file descriptor for an open directory, the pathname is evaluated relative to this directory.
3. If the *path* parameter specifies a relative pathname and the *fd* parameter has the special value `AT_FDCWD`, the pathname is evaluated starting in the current working directory, and `mkfifoat` behaves like `mkfifo`.

FIFO's rules and uses

FIFO's rules

- As with a pipe, if we write to a FIFO that no process has open for reading, the signal SIGPIPE is generated
- When the last writer for a FIFO closes the FIFO, an end of file is generated for the reader of the FIFO when all the data has been read
- It is common to have multiple writers for a given FIFO
  - use atomic writes to avoid interleaving of writes from multiple processes
  - as with pipes, the constant PIPE_BUF specifies the maximum amount of data that can be written atomically to a FIFO

FIFO's uses

1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files
2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers

Use 1.

Using FIFOs to Duplicate Output Streams

- FIFOs can be used to duplicate an output stream in a series of shell commands
  - Similarly to using pipes, this avoid using an intermediate disk file where to write the data
  - a FIFO has a name, so it can be easily used for nonlinear connections (processes simply need to have the name of the FIFO)

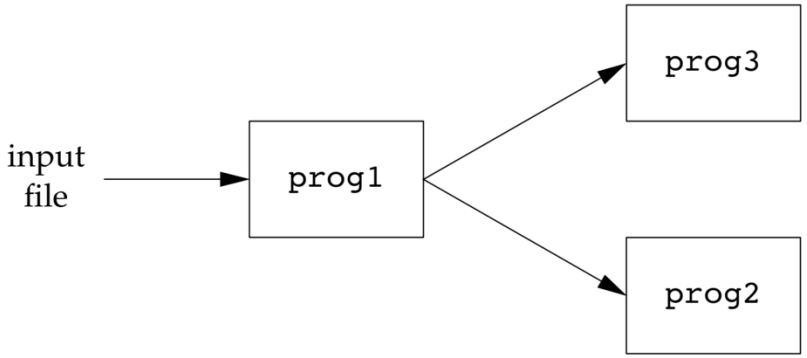


Figure 15.20 Procedure that processes a filtered input stream twice

Use 1.

```
mkfifo fifo13
prog3 < fifo13 & prog1 < infile | tee fifo13 | prog2
```

Then, you may want to remove the created fifo

```
rm fifo13 (OR unlink fifo13)
```

1. Remember that "&" means "run in the background" (the shell executes the command asynchronously in a subshell);
2. After `prog3 < fifo13` prog3 blocks waiting for writer(s), `tee` in our case

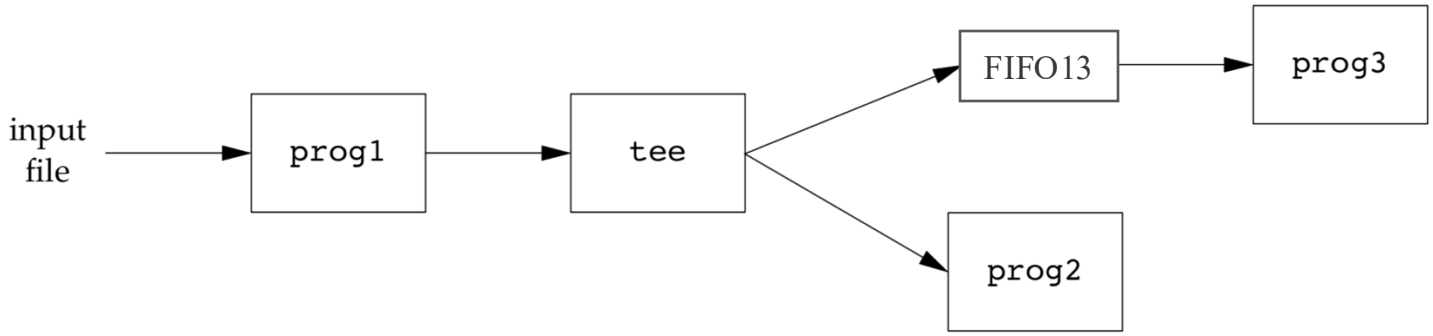


Figure 15.21 Using a FIFO and `tee` to send a stream to two different processes

A possible instance is chap15-ipc1/fifo-tee-script-1.sh

Use 1.

Exercise

- Realize the following configuration as first,
- then, realize more and different “connections”

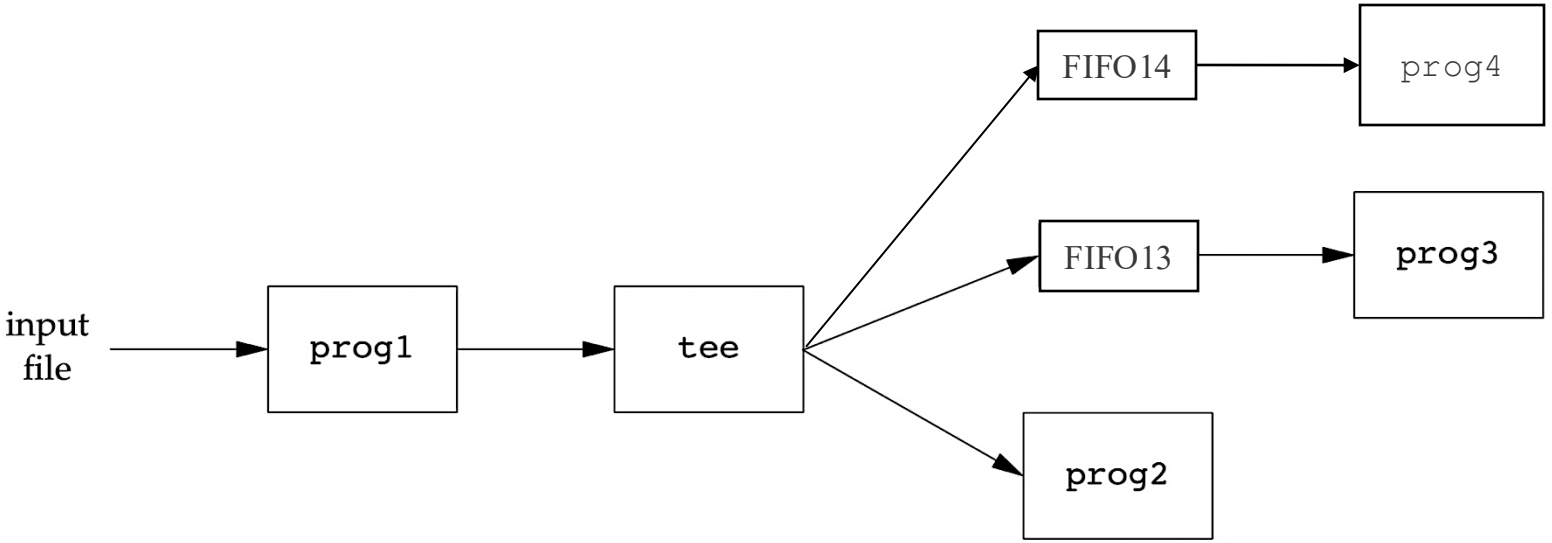


Figure 15.21 Using a FIFO and `tee` to send a stream to two different processes

Use 2.

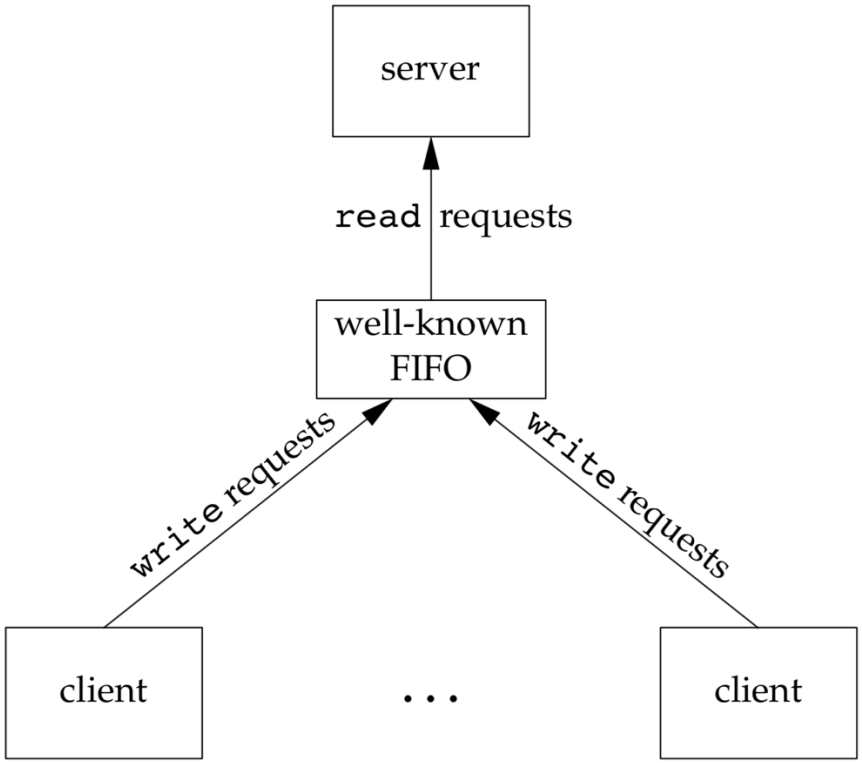


Figure 15.22 Clients sending requests to a server using a FIFO

Use 2.

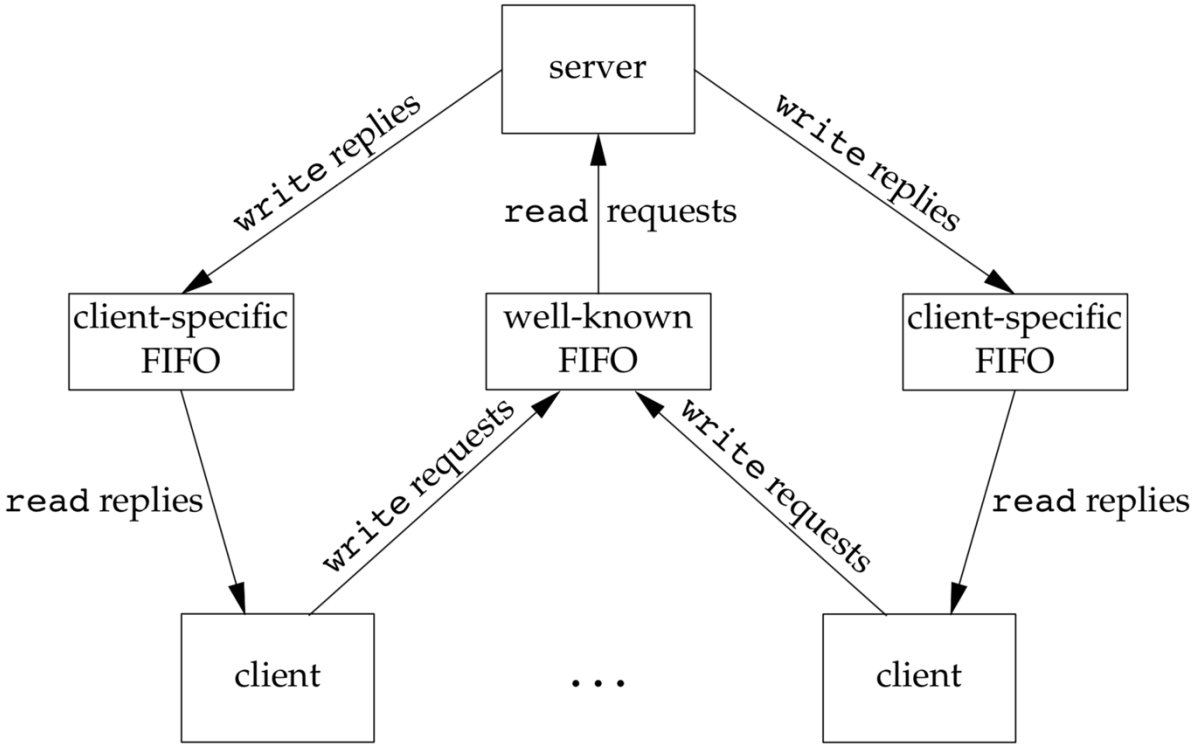


Figure 15.23 Client–server communication using FIFOs

Play with the Client-Server example in the folder chap15-ipc1/CS-FIFO-1/

Use 2.

After playing with the example for a while, the following may happen:

CS-FIFO-1 — bash — 126×38

```
bash-3.2$
bash-3.2$
bash-3.2$ ls -la
total 368
drwx------  23 marcoautili  staff    736 Dec 21 17:30 .
drwx------  24 marcoautili  staff    768 Nov 24 13:03 ..
-rwx------@  1 marcoautili  staff    323 Jan 14  2021 Makefile
-rwx------@  1 marcoautili  staff    856 Dec 21 16:20 README.txt
prw-r--r--   1 marcoautili  staff      0 Dec 21 16:36 add_client_response_fifo57575
prw-r--r--   1 marcoautili  staff      0 Dec 21 16:39 add_client_response_fifo_57620
prw-r--r--   1 marcoautili  staff      0 Dec 21 16:57 add_client_response_fifo_57821
prw-r--r--   1 marcoautili  staff      0 Dec 21 16:59 add_client_response_fifo_57908
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:00 add_client_response_fifo_57947
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:01 add_client_response_fifo_57961
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:07 add_client_response_fifo_57987
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:18 add_client_response_fifo_58257
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:22 add_client_response_fifo_58282
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:30 add_client_response_fifo_58452
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:30 add_client_response_fifo_58462
prw-r--r--   1 marcoautili  staff      0 Dec 21 17:22 adder_server_fifo
-rwxr-xr-x   1 marcoautili  staff  50224 Dec 21 17:30 client
-rwx------@  1 marcoautili  staff   2606 Dec 21 17:30 client.c
-rwx------@  1 marcoautili  staff    377 Dec 13  2022 common.h
-rwxr-xr-x   1 marcoautili  staff  50088 Dec 21 17:19 server
-rwx------@  1 marcoautili  staff   6455 Dec 21 17:19 server.c
-rwxr-xr-x   1 marcoautili  staff  49600 Dec 21 17:27 strtod-test
-rwx------@  1 marcoautili  staff    627 Dec 21 17:27 strtod-test.c
bash-3.2$
bash-3.2$
bash-3.2$ rm add_client_response_fifo*
bash-3.2$
```

XSI IPC

- There are three types of IPC that we call XSI IPC

  message queues, semaphores, and shared memory

- Each IPC structure in the kernel is referred to by a non-negative integer identifier
- Unlike file descriptors, IPC identifiers are not small integers
  - When a given IPC structure is created and then removed, the identifier associated with that structure continually increases until it reaches the maximum positive value for an integer, and then wraps around to 0
- The identifier is an internal name for an IPC object. Thus, cooperating processes need an external naming scheme to be able to rendezvous using the same IPC object
  - For this purpose, an IPC object is associated with a key that acts as an external name
- Whenever an IPC structure is being created by calling msgget(), semget(), or shmget(), a key must be specified
  - The data type of this key is the primitive system data type key_t, which is often defined as a long integer in the header <sys/types.h>
- The key is then converted into an identifier by the kernel

XSI IPC

- There are various ways for a client and a server to rendezvous at the same IPC structure (see Sec 15.6.1). One of them is as follows:
  1. The client and the server can agree on a pathname (an existing file) and project ID (the project ID is a character value between 0 and 255) and call the function *ftok* to convert these two values into a key
  2. The client and the server can agree on the generated key by sharing it in a file whose name might be defined in a common header
     - For 1. see the line of code **`sem_key = ftok("./sem_server.c", 42);`** of our example **`chap15-ipc1/sem-server.c`** and
     - For 2. see the **`sem-common.h`** header file

```
#include <sys/ipc.h>

key_t ftok(const char *path, int id);
```

Returns: key if OK, `(key_t)`−1 on error

- The only service provided by ftok is a way of generating a key from a pathname and (project) ID
- The *path* argument must refer to an existing file
- Only the lower 8 bits of *id* (0 - 255) are used when generating the key

XSI IPC permission Structure

XSI IPC associates an **ipc_perm** structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
struct ipc_perm {
  uid_t  uid;  /* owner's effective user ID */
  gid_t  gid;  /* owner's effective group ID */
  uid_t  cuid; /* creator's effective user ID */
  gid_t  cgid; /* creator's effective group ID */
  mode_t mode; /* access modes */
  :
};
```

Each implementation includes additional members. See **<sys/ipc.h>** on your system for the complete definition.

All the fields are initialized when the IPC structure is created.

See for instance semget() in the following slides

Message queue

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier (*queue ID*)

- *msgget* permits to create a new queue or open an existing queue

- New messages are added to the end of a queue by *msgsnd*
  - Every message has a positive long integer type field, a non-negative length, and the actual data bytes (corresponding to the length), all of which are specified to *msgsnd* when the message is added to a queue

- Messages are fetched from a queue by *msgrcv*
  - We do not have to fetch the messages in a first-in, first-out order
  - Instead, we can fetch messages based on their type field

Shared Memory

- Shared memory allows two or more processes to share a given region of memory
- This is the fastest form of IPC, because the data does not need to be copied between the client and the server
- The only trick in using shared memory is synchronizing access to a given region among multiple processes
  - If the server is placing data into a shared memory region, the client should not try to access the data until the server is done
- Often, semaphores are used to synchronize shared memory access

Semaphores

- A semaphore is not a form of IPC similar to the others that we have described (pipes, FIFOs, shared memory and message queues)

- A semaphore is a counter used to provide access to a shared data object for multiple processes

- To obtain a shared resource, a process needs to do the following:
  1) Test the semaphore that controls the resource
  2) If the value of the semaphore is positive, the process can use the resource. In this case, the process decrements the semaphore value by 1, indicating that it has used one unit of the resource
  3) Otherwise, if the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up, it returns to step 1

Semaphores

- When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened

- To implement semaphores correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation. For this reason, semaphores are normally implemented inside the kernel

- A common form of semaphore is called a binary semaphore
  - It controls a single resource, and its value is initialized to 1
  - In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing

Traffic lights VS Linux/UNIX semaphore

Differently from the green/orange/red traffic lights we all have in mind

- The UNIX/Linux system calls for semaphores work on an array of semaphores and not on a single semaphore
- It is possible to specify how much to increase/decrease the value of a semaphore
- A semaphore is not simply a single non-negative value
- Semaphores have a “wait-for-zero” semantics if 0 is specified (following slides)

Semaphores: system calls at-a-glance

semget()

- To get a set of (an array of) semaphores and obtain the the corresponding semaphore ID

semop()

- To atomically performs an array of operations on a semaphore set

semctl()

- To perform various control operations, e.g., removing a vector of semaphores, setting permissions, returning the semaphore value, etc.

Semaphores

- We have to define a semaphore as a set of (an array of) one or more semaphore
- When we create a semaphore, we specify the number of semaphores in the set
- The kernel maintains a semid_ds structure for each semaphore set

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  :
};
```

See previous slide

When a new set is created

- sem_otime is set to 0
- sem_ctime is set to the current time
- sem_nsems is set to nsems → next slide

Semaphore representation

- Each semaphore is represented by an anonymous structure containing at least the following members

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /* # processes awaiting semval>curval */
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  :
};
```

Obtaining a Semaphore

When we want to use XSI semaphores, we first need to obtain a semaphore ID by calling the *semget* function

```
#include <sys/sem.h>
int semget(key_t key, int nsems, int flag);
                                    Returns: semaphore ID if OK, –1 on error
```

The number of semaphores in the set is *nsems*

- If a new set is being created (typically by the server), we must specify *nsems*
- Flag is the ORing ("|") of the constants IPC_CREAT and IPC_EXCL (exclusive creation) and Permission bits (r to know its value, w to update its value, x no meaning)

In general, semget() can be used to create a pool of semaphores. However, by passing *nsems* as 1, only one semaphore will be created in the pool identified by the returned id

- The semaphore will be then referenced with "index" 0

```
sem_id = semget(sem_key, 1, IPC_CREAT | IPC_EXCL | 0600);
```

From the example chap15-ipc1/sem-server.c

Operating with a semaphore

- The semctl() function is the catch-all for various semaphore operations (see Sec. 15.8 for further details)

```
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd, ... /* union semun arg */ );
                                                    Returns: (see following)
```

- The cmd argument specifies one of ten commands
  IPC_STAT, IPC_SET, IPC_RMID, GETVAL, SETVAL, GETPID, GETNCNT, GETZCNT, GETALL, SETALL
  to be performed on the set specified by semid (see next slide)
- The five commands that refer to one particular semaphore value use semnum to specify one member of the set. The value of semnum is between 0 and nsems – 1, inclusive
- The fourth argument is optional, depending on the command requested, and if present, is of type semun, a union of various command-specific arguments

```
union semun {
  int               val;    /* for SETVAL */
  struct semid_ds  *buf;    /* for IPC_STAT and IPC_SET */
  unsigned short   *array;  /* for GETALL and SETALL */
};
```

Units of available resources

Semaphore commands

`IPC_STAT`	Fetch the `semid_ds` structure for this set, storing it in the structure pointed to by *arg.buf*.
`IPC_SET`	Set the `sem_perm.uid`, `sem_perm.gid`, and `sem_perm.mode` fields from the structure pointed to by *arg.buf* in the `semid_ds` structure associated with this set. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.
`IPC_RMID`	Remove the semaphore set from the system. This removal is immediate. Any other process still using the semaphore will get an error of `EIDRM` on its next attempted operation on the semaphore. This command can be executed only by a process whose effective user ID equals `sem_perm.cuid` or `sem_perm.uid` or by a process with superuser privileges.
`GETVAL`	Return the value of `semval` for the member *semnum*.
`SETVAL`	Set the value of `semval` for the member *semnum*. The value is specified by *arg.val*.
`GETPID`	Return the value of `sempid` for the member *semnum*.
`GETNCNT`	Return the value of `semncnt` for the member *semnum*.
`GETZCNT`	Return the value of `semzcnt` for the member *semnum*.
`GETALL`	Fetch all the semaphore values in the set. These values are stored in the array pointed to by *arg.array*.
`SETALL`	Set all the semaphore values in the set to the values pointed to by *arg.array*.

Operating with a semaphore

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
                                                    Returns: 0 if OK, –1 on error
```

- The function semop() atomically performs an array of operations on a semaphore set
- The *semoparray* argument is a pointer to an array of semaphore operations, represented by *sembuf* structures
- The *nops* argument specifies the number of operations (elements) in the array.

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

See following slides

semop() semantics

The operation on each member of the set is specified by the corresponding `sem_op` value. This value can be negative, 0, or positive. (In the following discussion, we refer to the "undo" flag for a semaphore. This flag corresponds to the `SEM_UNDO` bit in the corresponding `sem_flg` member.)

1. The easiest case is when `sem_op` is positive. This case corresponds to the returning of resources by the process. The value of `sem_op` is added to the semaphore's value. If the undo flag is specified, `sem_op` is also subtracted from the semaphore's adjustment value for this process.

semop() semantics

2. If `sem_op` is negative, we want to obtain resources that the semaphore controls.

   If the semaphore's value is greater than or equal to the absolute value of `sem_op` (the resources are available), the absolute value of `sem_op` is subtracted from the semaphore's value. This guarantees the resulting semaphore value is greater than or equal to 0. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

   If the semaphore's value is less than the absolute value of `sem_op` (the resources are not available), the following conditions apply.

   a. If `IPC_NOWAIT` is specified, `semop` returns with an error of `EAGAIN`.

   b. If `IPC_NOWAIT` is not specified, the `semncnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

      i. The semaphore's value becomes greater than or equal to the absolute value of `sem_op` (i.e., some other process has released some resources). The value of `semncnt` for this semaphore is decremented (since the calling process is done waiting), and the absolute value of `sem_op` is subtracted from the semaphore's value. If the undo flag is specified, the absolute value of `sem_op` is also added to the semaphore's adjustment value for this process.

      ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.

      iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semncnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.

See last slide

The operation could not proceed immediately (and IPC_NOWAIT was specified in sem_flg)

It means "Resource temporarily unavailable"

It is equivalent to EWOULDBLOCK

semop() semantics

3. If `sem_op` is 0, this means that the calling process wants to wait until the semaphore's value becomes 0.

   If the semaphore's value is currently 0, the function returns immediately.

   If the semaphore's value is nonzero, the following conditions apply.

   a. If `IPC_NOWAIT` is specified, return is made with an error of `EAGAIN`.

   b. If `IPC_NOWAIT` is not specified, the `semzcnt` value for this semaphore is incremented (since the caller is about to go to sleep), and the calling process is suspended until one of the following occurs.

      i. The semaphore's value becomes 0. The value of `semzcnt` for this semaphore is decremented (since the calling process is done waiting).

      ii. The semaphore is removed from the system. In this case, the function returns an error of `EIDRM`.

      iii. A signal is caught by the process, and the signal handler returns. In this case, the value of `semzcnt` for this semaphore is decremented (since the calling process is no longer waiting), and the function returns an error of `EINTR`.

"wait-for-zero" semantics

The `semop` function operates atomically; it does either all the operations in the array or none of them.

Semaphore Adjustment on exit

As we mentioned earlier, it is a problem if a process terminates while it has resources allocated through a semaphore

- Whenever we specify the SEM_UNDO flag for a semaphore operation and we allocate resources (a sem_op value less than 0), the kernel remembers how many resources we allocated from that particular semaphore (the absolute value of sem_op)
- When the process terminates, either voluntarily or involuntarily, the kernel checks whether the process has any outstanding semaphore adjustments and, if so, applies the adjustment to the corresponding semaphore
- If we set the value of a semaphore using semctl, with either the SETVAL or SETALL commands, the adjustment value for that semaphore in all processes is set to 0

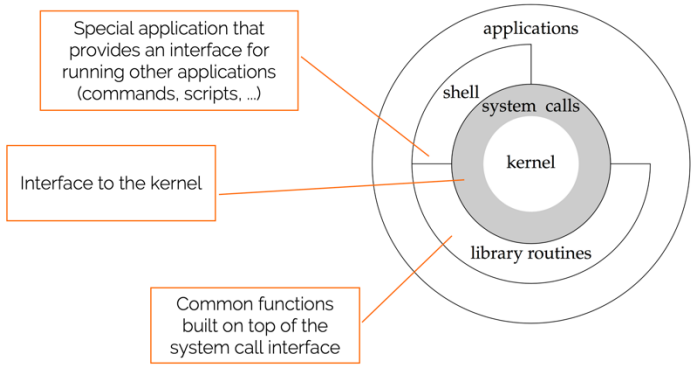
Play with the example

chap15-ipc1/sem-server.c and chap15-ipc1/sem-client.c

That's all folks ☺

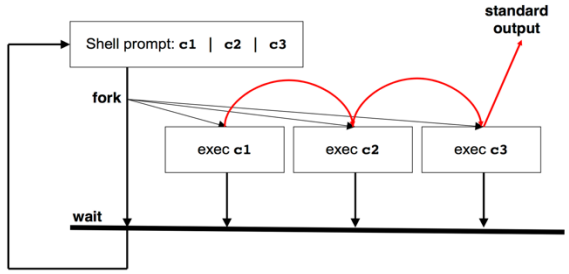
UNIX Architecture

- An operating system can be defined as the software that controls the hardware resources of the computer and provides an environment for running programs

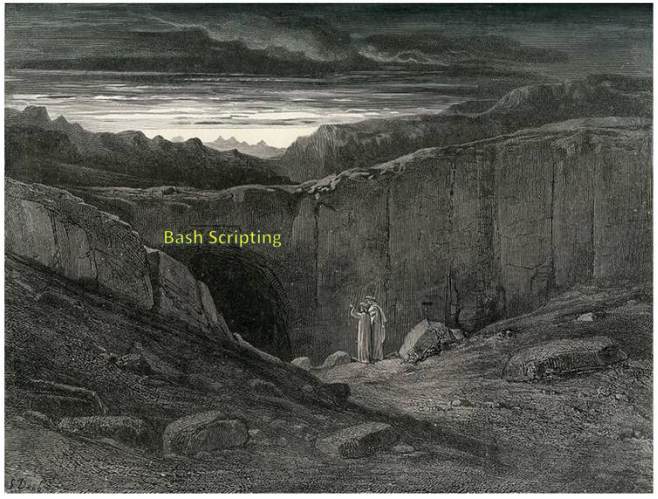


Pipeline execution order (from Part 2)

- The order the processes are run does not matter and is not guaranteed
- The shell first creates the pipe, the conduit for the data that will flow between the processes, and then creates the processes with the ends of the pipe connected to them
- For example, the first process that is run may block waiting for input from the second process, or block waiting for the second process to start reading data from the pipe. These waits can be arbitrarily long and do not matter
- Whichever order the processes are run, the data eventually gets transferred and everything works



"Lasciate ogni speranza, voi ch'intrate" 😯



Use 2.

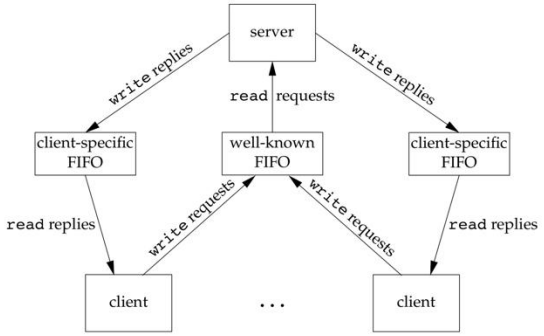


Figure 15.23 Client–server communication using FIFOs

See Client-Server example in the folder chap15-ipc1/CS-FIFO-1/