UE20CS254

Operating Systems

Suresh Jamadagni

Department of Computer Science and Engineering

Operating Systems

System calls for Inter Process Communication

Suresh Jamadagni

Department of Computer Science and Engineering

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.

pipe()

- A pipe is created by calling the **pipe()** function.

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

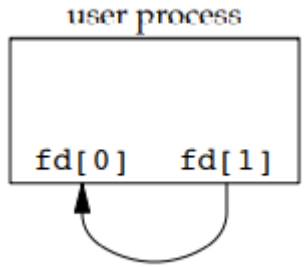
- Return value: 0 if OK, −1 on error
- Two file descriptors are returned through the fd argument:
  1. fd[0] is open for reading
  2. fd[1] is open for writing.
- The output of fd[1] is the input for fd[0]

OPERATING SYSTEMS

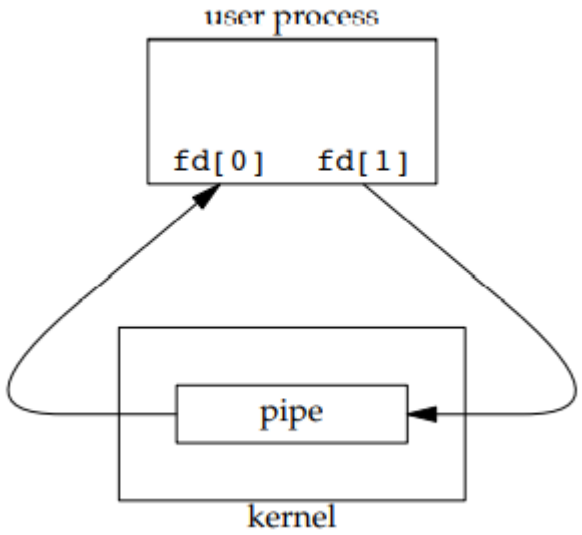
pipe()

- Two ways to picture a half-duplex pipe

1. The two ends of the pipe connected in a single process



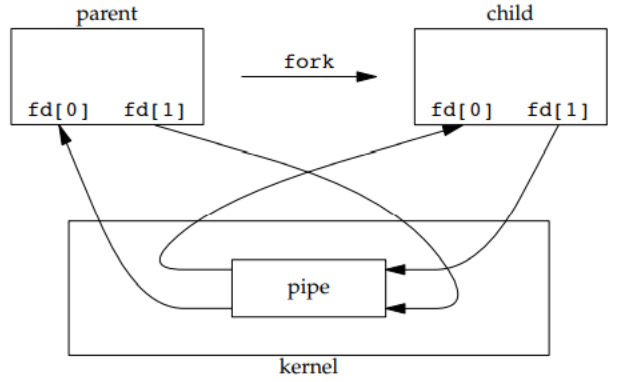
2. The data in the pipe flows through the kernel



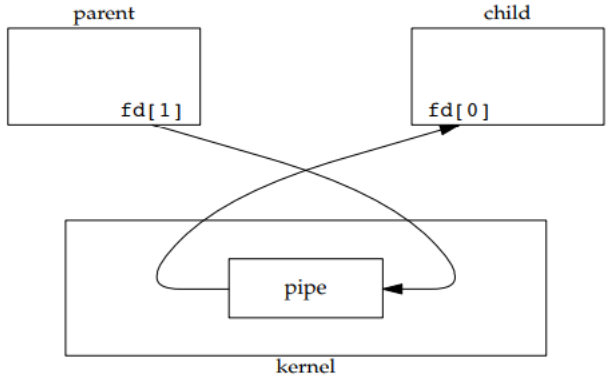
OPERATING SYSTEMS

pipe()

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



- For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]) and the child closes the write end (fd[1]).
- For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0]



Pipe from parent to child

OPERATING SYSTEMS

popen() and pclose()

- A common operation is to create a pipe to another process to either read its output or send it input, the standard I/O library has historically provided the **popen()** and **pclose()** functions
- These two functions handle all the work: creating a pipe, forking a child, closing the unused ends of the pipe, executing a shell to run the command, and waiting for the command to terminate.

```
#include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);

int pclose(FILE *fp);
```

- The function **popen()** does a fork and exec to execute the cmdstring and returns a standard I/O file pointer.
- If type is "r", the file pointer is connected to the standard output of cmdstring.
- If type is "w", the file pointer is connected to the standard input of cmdstring
- The function popen() returns file pointer if OK, NULL on error
- The function **pclose()** closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell or -1 on error

OPERATING SYSTEMS

FIFOs

- FIFOs are sometimes called **named pipes**.
- Unnamed pipes can be used only between related processes when a common ancestor has created the pipe.
- With FIFOs, unrelated processes can exchange data
- Creating a FIFO is similar to creating a file

```
#include <sys/stat.h>
int mkfifo(const char *path, mode_t mode);
int mkfifoat(int fd, const char *path, mode_t mode);
```

- Functions mkfifo() and mkfifoat() return 0 if OK, −1 on error
- Once a FIFO has been created using mkfifo() or mkfifoat() , it can be opened using open(). Normal file I/O functions (e.g., close, read, write, unlink) all work with FIFOs.
- There are two uses for FIFOs.
    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

OPERATING SYSTEMS

Message Queues

- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
- A new queue is created or an existing queue opened by **msgget**.

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

int msgget(key_t key, int flag);
```

- Returns message queue ID if OK, −1 on error
- New messages are added to the end of a queue by **msgsnd**.

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```

- Every message has a positive long integer type field, a non-negative length, and the actual data bytes all of which are specified to **msgsnd** when the message is added to a queue.
- Messages are fetched from a queue by **msgrcv**.

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

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
```

- Messages need not be fetched in a first-in, first-out order. Instead, can be fetched based on their type field

OPERATING SYSTEMS

Semaphores

- 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.
- 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.

OPERATING SYSTEMS

Semaphores

- 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
- **semctl** function is the catchall for various semaphore operations.

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

- The function **semop** atomically performs an array of operations on a semaphore set

```
#include <sys/sem.h>
int semop(int semid, struct sembuf semoparray[], size_t nops);
```

- The ore operations
- Returns 0 if OK, −1 on error

OPERATING SYSTEMS

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 challenge in using shared memory is synchronizing access to a given region among multiple processes.
- Semaphores and mutexes are used to synchronize shared memory access
- Function **shmget** is used to obtain a shared memory identifier

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

int shmget(key_t key, size_t size, int flag);
```

- Returns shared memory ID if OK, −1 on error

OPERATING SYSTEMS

Shared memory

- Function **shmctl** is the catchall for various shared memory operations

```
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Returns 0 if OK, −1 on error
- The cmd argument specifies one of the commands to be performed on the segment specified by shmid
- Once a shared memory segment has been created, a process is attached to this memory segment by calling **shmat**

```
#include <sys/shm.h>
void *shmat(int shmid, const void *addr, int flag);
```

- Returns: pointer to shared memory segment if OK, −1 on error

OPERATING SYSTEMS

Shared memory

- Function call **shmdt** will detach a shared memory segment from a process

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

int shmdt(const void *addr);
```

- Returns 0 if OK, −1 on error
- A shared memory segment can be removed by calling **shmctl** with a command of IPC_RMID.

THANK YOU

Suresh Jamadagni
Department of Computer Science and Engineering

sureshjamadagni@pes.edu