System Programming

Week 14: Interprocess Commnunication

Seongjin Lee

Updated: 2016-12-07
11_ipc

insight@hanyang.ac.kr
http://esos.hanyang.ac.kr
Esos Lab. Hanyang University

Table of contents

Introduction

This chapter covers other techniques for processes to communicate with one another: interprocess communication (IPC)

- pipes
- FIFOs
- Message Queues
- Shared Memory
- Semaphores

PIPES

Pipes

Pipes are the oldest form of UNIX system IPC

Pipes have two limitations

1. They have been half duplex (most commonly used), now provide full-duplex pipes (but never assume it is full-duplex)
2. Pipes can be used only between processes that have a common ancestor

Pipes

```
#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,
- `fd[1]` is open for writing.
- The output of `fd[1]` is the input for `fd[0]`.

Pipes

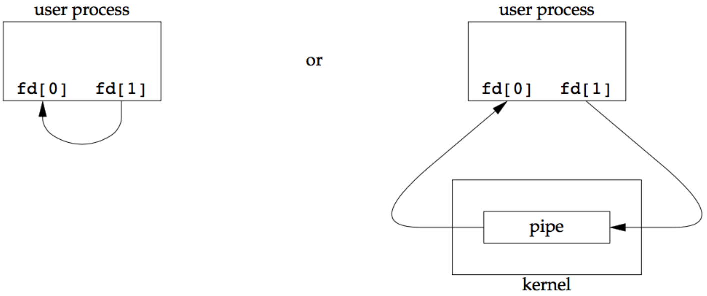


Figure: Two ways to veiw a half-duplex pipe

Pipes

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

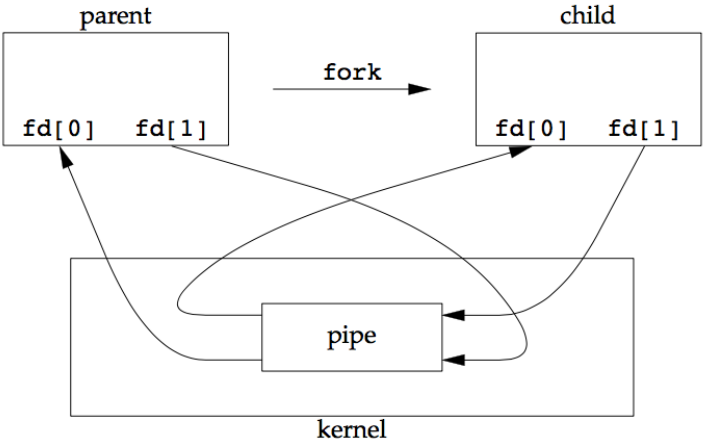


Figure: Half-duplex pipe after a fork

Pipes

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

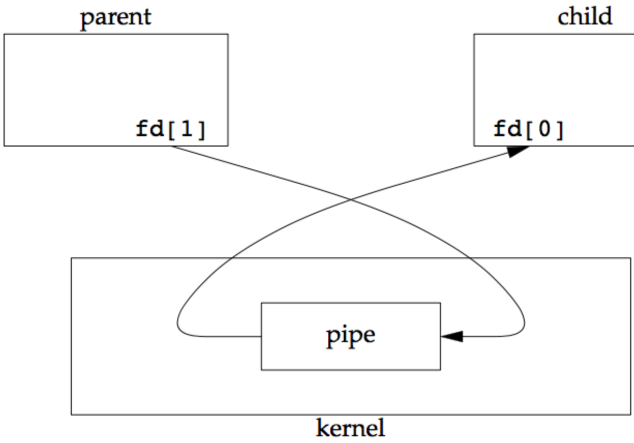


Figure: Pipe from parent to child

Pipes

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

1. If we read from a pipe whose write end has been closed, read returns 0 to indicate an end of file after all the data has been read.
2. If we write to a pipe whose read end has been closed, the signal SIGPIPE is generated.

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

Pipes Example: codes/pipe1.c I

In this example, we call read and write directly on the pipe descriptors.

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


#define MAXLINE 4096   /* max line length */

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

   if (pipe(fd) < 0){
      fprintf(stderr, "pipe error");
      exit(1);
   }
   if ((pid = fork()) < 0) {
```

Pipes Example: codes/pipe1.c II

```
        fprintf(stderr, "fork error");
        exit(1);
    } else if (pid > 0) {  /* parent */
        close(fd[0]);
        write(fd[1], "hello world\n", 12);
    } else {      /* child */
        close(fd[1]);
        n = read(fd[0], line, MAXLINE);
        write(STDOUT_FILENO, line, n);
    }
    exit(0);
}
```

Pipes Example: `codes/pipe2.c` I

- Before calling fork, we create a pipe.
- After the fork, the parent closes its read end, and the child closes its write end.
- The child then calls dup2 to have its standard input be the read end of the pipe.
- When the pager program is executed, its standard input will be the read end of the pipe.

Pipes Example: codes/pipe2.c II

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

#define MAXLINE 4096   /* max line length */
#define DEF_PAGER "/usr/bin/more"  /* default pager program */

int
main(int argc, char *argv[])
{
   int  n;
   int  fd[2];
   pid_t pid;
   char *pager, *argv0;
   char line[MAXLINE];
   FILE *fp;

   if (argc != 2){
      fprintf(stderr, "usage: a.out <pathname>");
      exit(1);
   }
```

Pipes Example: codes/pipe2.c III

```

    if ((fp = fopen(argv[1], "r")) == NULL){
       fprintf(stderr, "can’t open %s", argv[1]);
       exit(1);
    }
    if (pipe(fd) < 0){
       fprintf(stderr, "pipe error");
       exit(1);
    }

    if ((pid = fork()) < 0) {
       fprintf(stderr, "fork error");
       exit(1);
    } else if (pid > 0) {         /* parent */
       close(fd[0]);  /* close read end */

       /* parent copies argv[1] to pipe */
       while (fgets(line, MAXLINE, fp) != NULL) {
          n = strlen(line);
          if (write(fd[1], line, n) != n)
             fprintf(stderr, "write error to pipe");
       }
       if (ferror(fp))
```

Pipes Example: codes/pipe2.c IV

```
            fprintf(stderr, "fgets error");

        close(fd[1]); /* close write end of pipe for reader */

        if (waitpid(pid, NULL, 0) < 0){
            fprintf(stderr, "waitpid error");
            exit(1);
        }
        exit(0);
    } else {            /* child */
        close(fd[1]); /* close write end */
        if (fd[0] != STDIN_FILENO) {
            if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO){
                fprintf(stderr, "dup2 error to stdin");
                exit(1);
            }
            close(fd[0]); /* don't need this after dup2 */
        }

        /* get arguments for execl() */
        if ((pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;
        if ((argv0 = strrchr(pager, '/')) != NULL)
```

Pipes Example: `codes/pipe2.c` V

```
            argv0++;  /* step past rightmost slash */
        else
            argv0 = pager; /* no slash in pager */

        if (execl(pager, argv0, (char *)0) < 0){
            fprintf(stderr, "execl error for %s", pager);
            exit(1);
        }
    }
    exit(0);
}
```

Defensive programming measure

Whenever we call dup2 and close to duplicate one descriptor onto another, we'll always compare the descriptors first.

PIPES : popen AND pclose FUNCTIONS

popen and pclose Functions

Standard I/O library to create a pipe to another process to either read or write

```
#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
```

The function popen does a `fork` and `exec` to execute the *cmdstring* and returns a standard I/O file pointer.

`popen` and `pclose` Functions

If *type* is "`r`", the file pointer is connected to the standard output of *cmdstring*

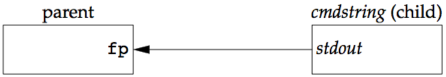
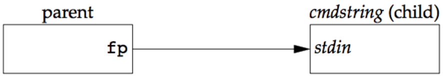


Figure: Result of `fp = popen(cmdstring, ''r'')`

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



popen and pclose Functions

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

popen and pclose Example I

redo of the program using popen: codes/popen2.c

The shell command ${PAGER:-more} says to use the value of the shell variable PAGER if it is defined and non-null; otherwise, use the string more.

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

#define MAXLINE 4096   /* max line length */
#define PAGER "${PAGER:-more}" /* environment variable, or default */

int
main(int argc, char *argv[])
{
   char line[MAXLINE];
   FILE *fpin, *fpout;

   if (argc != 2){
```

popen and pclose Example II

```
        fprintf(stderr, "usage: a.out <pathname>");
        exit(1);
    }
    if ((fpin = fopen(argv[1], "r")) == NULL){
        fprintf(stderr, "can’t open %s", argv[1]);
        exit(1);
    }

    if ((fpout = popen(PAGER, "w")) == NULL){
        fprintf(stderr, "popen error");
        exit(1);
    }

    /* copy argv[1] to pager */
    while (fgets(line, MAXLINE, fpin) != NULL) {
        if (fputs(line, fpout) == EOF){
            fprintf(stderr, "fputs error to pipe");
            exit(1);
        }
    }
    if (ferror(fpin)){
        fprintf(stderr, "fgets error");
        exit(1);
```

popen and pclose Example III

```
    }
    if (pclose(fpout) == -1){
       fprintf(stderr, "pclose error");
       exit(1);
    }

    exit(0);
}
```

popen and pclose Example

codes/myuclc.c : a filter program that changes upper cases to lower cases.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>

int
main(void)
{
   int  c;

   while ((c = getchar()) != EOF) {
      if (isupper(c))
         c = tolower(c);
      else if (islower(c))
         c = toupper(c);
      if (putchar(c) == EOF){
         fprintf(stderr, "output error");
         exit(1);
      }
      if (c == '\n')
         fflush(stdout);
   }
   exit(0);
```

popen and pclose Example I

codes/popen1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

#define MAXLINE 4096   /* max line length */

int
main(void)
{
   char line[MAXLINE];
   FILE *fpin;

   if ((fpin = popen("./myuclc", "r")) == NULL){
      fprintf(stderr, "popen error");
      exit(1);
   }

    /* type 111 to exit the program */
```

popen and pclose Example II

```
    while (strcmp(line, "111\n") != 0){
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            fprintf(stderr, "fputs error to pipe");
    }
    if (pclose(fpin) == -1){
        fprintf(stderr, "pclose error");
        exit(1);
    }
    putchar('\n');
    exit(0);
}
```

Coprocesses

Coprocesses

A UNIX system filter is a program that reads from standard input and writes to standard output.

Filters are normally connected linearly in shell pipelines.

A filter becomes a *coprocess* when the same program generates the filter's input and reads the filter output

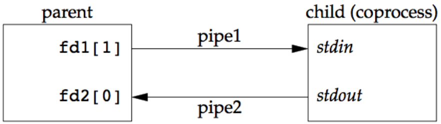


Figure: Driving a coprocess by writing its standard input and reading its standard output

Coprocess Example I

A simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output.

`codes/add2.c; make add2`

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

#define MAXLINE 4096   /* max line length */

int
main(void)
{
   int  n, int1, int2;
   char line[MAXLINE];

   while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
      line[n] = 0;  /* null terminate */
      if (sscanf(line, "%d%d", &int1, &int2) == 2) {
         sprintf(line, "%d\n", int1 + int2);
         n = strlen(line);
         if (write(STDOUT_FILENO, line, n) != n){
```

Coprocess Example II

```
                fprintf(stderr, "write error");
                exit(1);
            }
        } else {
            if (write(STDOUT_FILENO, "invalid args\n", 13) != 13){
                fprintf(stderr, "write error");
                exit(1);
            }
        }
    }
    exit(0);
}
```

Here, we create two pipes, with the parent and the child closing the ends they don’t need.

Coprocess Example III

codes/pipe4.c; make pipe4

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

#define MAXLINE 4096   /* max line length */

static void sig_pipe(int);  /* our signal handler */

int
main(void)
{
   int  n, fd1[2], fd2[2];
   pid_t pid;
   char line[MAXLINE];

   if (signal(SIGPIPE, sig_pipe) == SIG_ERR){
      fprintf(stderr, "signal error");
      exit(1);
   }

   if (pipe(fd1) < 0 || pipe(fd2) < 0){
      fprintf(stderr, "pipe error");
      exit(1);
```

Coprocess Example IV

```
    }

    if ((pid = fork()) < 0) {
       fprintf(stderr, "fork error");
       exit(1);
    } else if (pid > 0) {        /* parent */
       close(fd1[0]);
       close(fd2[1]);

       while (fgets(line, MAXLINE, stdin) != NULL) {
          if(strcmp(line, "quit\n") == 0 )
             exit(0);
          n = strlen(line);
          if (write(fd1[1], line, n) != n){
             fprintf(stderr, "write error to pipe");
             exit(1);
          }
          if ((n = read(fd2[0], line, MAXLINE)) < 0){
             fprintf(stderr, "read error from pipe");
             exit(1);
          }
          if (n == 0) {
             fprintf(stderr, "child closed pipe");
             break;
          }
          line[n] = 0; /* null terminate */
```

Coprocess Example V

```
          if (fputs(line, stdout) == EOF){
             fprintf(stderr, "fputs error");
             exit(1);
          }
       }

       if (ferror(stdin)){
          fprintf(stderr, "fgets error on stdin");
          exit(1);
       }
       exit(0);
    } else {          /* child */
       close(fd1[1]);
       close(fd2[0]);
       if (fd1[0] != STDIN_FILENO) {
          if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO){
             fprintf(stderr, "dup2 error to stdin");
             exit(1);
          }
          close(fd1[0]);
       }

       if (fd2[1] != STDOUT_FILENO) {
          if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO){
             fprintf(stderr, "dup2 error to stdout");
             exit(1);
```

Coprocess Example VI

```
      }
      close(fd2[1]);
    }
    if (execl("./add2", "add2", (char *)0) < 0){
      fprintf(stderr, "execl error");
      exit(1);
    }
  }
  exit(0);
}

static void
sig_pipe(int signo)
{
  printf("SIGPIPE caught\n");
  exit(1);
}
```

Coprocess Example VII

If we `kill` the add2 coprocess while the program is waiting for our input and then enter two numbers, the signal handler is invoked when the program writes to the pipe that has no reader.

From another terminal type the following

```
$ ps | grep add2
40467 ttys000 0:00.00 add2
$ kill 40467
```

then input two numbers to pipe4

FIFOs

FIFOs

FIFOs are sometimes called named pipes.

- 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.

FIFOs

Creating a FIFO is similar to creating a file. Indeed, the pathname for a FIFO exists in the file system

```
 #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
```

FIFOs

- *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
- `mkfifoat` creates a FIFO in a location relative to the directory represented by the `fd` file descriptor argument.
  - *path* == absolute pathname: `mkfifoat` function behaves like the `mkfifo` function.
  - *path* == relative pathname && `fd` == valid directory: the pathname is evaluated relative to this directory.
  - *path* == relative pathname && `fd` == `AT_FDCWD`: starts incurrent working directory, and mkfifoat behaves like mkfifo.

Once we have used `mkfifo` or `mkfifoat` to create a FIFO, we open it using `open`. Indeed, the normal file I/O functions (e.g., close, `read`, `write`, `unlink`) all work with FIFOs.

FIFOs

When we open a FIFO, the nonblocking flag (`O_NONBLOCK`) affects what happens.

- without `O_NONBLOCK`:
    - open for read-only blocks until some other process opens the FIFO for writing
    - open for write-only blocks until some other process opens the FIFO for reading.
- with `O_NONBLOCK`:
    - open for read-only returns immediately.
    - open for write-only returns -1 with `errno` set to `ENXIO` if no process has the FIFO open for reading.

FIFO Use Cases

There are two uses for FIFO

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`.

Example - Using FIFOs to Duplicate Output Streams

FIFOs can be used to duplicate an output stream in a series of shell commands.

A FIFO has a name, so it can be used for nonlinear connections.

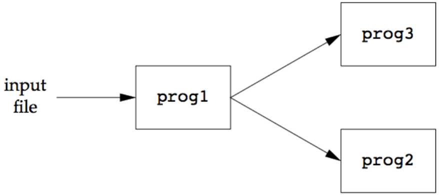


Figure: Procedure that processes a filtered input stream twice

Example - Using FIFOs to Duplicate Output Streams

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

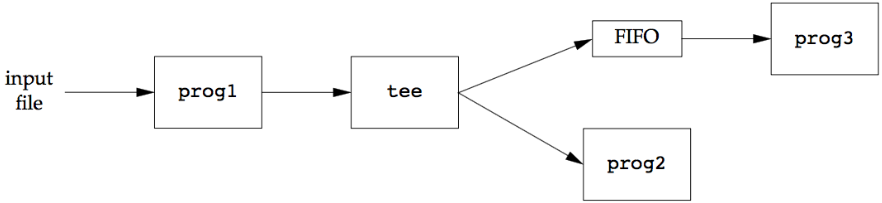


Figure: Using a FIFO and tee to send a stream to two different processes

Example - Client Server Communication Using FIFO

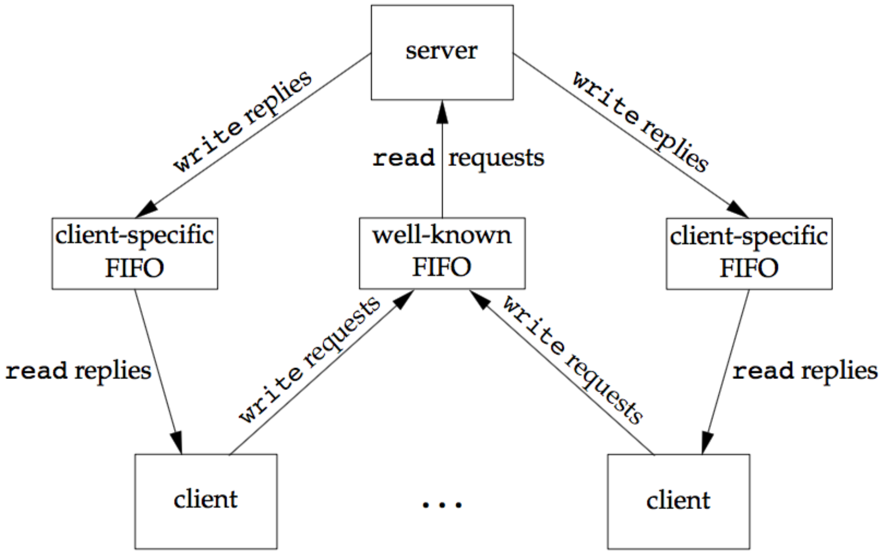


Figure: Client-server communication using FIFOs

Example - Client Server Communication Using FIFO I

codes/create_fifo.c

make create_fifo to create a FIFO for server and client

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

int main()
{
   int server, client;

   /* create a FIFO for server */
   server = mkfifo("FIFO-server", 0666);
   if(server < 0) {
      fprintf(stderr, "Failed to create a FIFO for server");
      exit(-1);
   }

   /* create a FIFO for client */
   client = mkfifo("FIFO-client", 0666);
```

Example - Client Server Communication Using FIFO II

```
    if(client < 0) {
        fprintf(stderr, "Failed to create a FIFO for client");
        exit(-1);
    }

    printf("FIFO for server and child created successfuly");

    return 0;
}
```

Example - Client Server Communication Using FIFO III

codes/server.c

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

int main()
{
   int FIFO_server,FIFO_client;
   int choice;
   char *buf;

   FIFO_server = open("FIFO-server", O_RDWR);
   if(FIFO_server < 0) {
      fprintf(stderr, "Error opening server side FIFO file");
      exit(-1);
   }

   read(FIFO_server, &choice, sizeof(int));

   sleep(1);

   FIFO_client = open("FIFO-client", O_RDWR);
   if(FIFO_client < 0) {
```

Example - Client Server Communication Using FIFO IV

```
        fprintf(stderr, "Error opening client side FIFO file");
        exit(-1);
    }

    switch(choice) {
        case 1:
            buf="Linux";
            break;
        case 2:
            buf="Debian";
            break;
        case 3:
            buf="4.0";
    }

    write(FIFO_client,buf,10*sizeof(char));
    printf("\n### Data sent to client ###\n");

    close(FIFO_server);
    close(FIFO_client);
    return 0;
}
```

Example - Client Server Communication Using FIFO V

codes/client.c

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

int main()
{
   char *buf;
   int choice = 1;
   int fifo_server, fifo_client;

   printf("Chose one of following options to send to the server\n");
   printf("\t\t 1 for the name of the OS \n \
           2 for the name of distribution \n \
           3 for the version of Kernel \n");
   printf("Your choice: ");
   scanf("%d",&choice);

   fifo_server=open("FIFO-server",O_RDWR);
   if(fifo_server < 0) {
      fprintf(stderr, "Error in opening file");
      exit(-1);
    }

```

Example - Client Server Communication Using FIFO VI

```
    write(fifo_server, &choice, sizeof(int));

    fifo_client=open("FIFO-client", O_RDWR);

    if(fifo_client < 0) {
        fprintf(stderr, "Error in opening file");
        exit(-1);
     }

    buf=malloc(10*sizeof(char));
    read (fifo_client, buf, 10*sizeof(char));
    printf("\n ### Server replied with %s ###\n", buf);

    close(fifo_server);
    close(fifo_client);
    return 0;
}
```

XSI IPC

XSI IPC : General Background

XSI IPC

There are three types of IPC that we call XSI IPC

1. message queues
2. semaphores
3. shared memory

Identifiers and Keys

Each IPC *structure* in the kernel is referred to by a non-negative integer *identifier*

- to send a message to or fetch a message from a message queue, for example, all we need to know is the identifier for the queue
- when a given IPC structure is created and then removed
  - identifier associated with that structure conitunally increases
- identifier is an internal name for an IPC object

Identifiers and Keys

Whenever an IPC structure is being created, a key must be specified

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
// Returns: key if OK, (key_t)1 on error
```

- *path* must refer to an existing file
- only the lower 8 bits of *id* are used when generating the key
- key created by `ftok` is foremd by taking parts of `st_dev` and `st_ino` fields in the `stat` structure

Identifiers and Keys

The three `get` functions (`msgget`, `semget`, and `shmget`) all have two similar arguments: a *key* and an integer *flag*

If we want to create a new unique IPC structure, we must specify a *flag* with both the `IPC_CREAT` and `IPC_EXCL` bits set

Permission Structure

XSI IPC associates an `ipc_perm` structure with each IPC structure

All the fields are initialized when the IPC structure is created

`{msg | sem | shm}ctl` can modify the `uid`, `gid`, and `mode` fields

```
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 */
    .
    .
    .
};
```

Permission	Bit
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

IPC vs File

File		IPC
File name	$\longleftrightarrow$	Key
file	$\longleftrightarrow$	Object
fd	$\longleftrightarrow$	Identifier

Difference between object identifiers and `fd`s

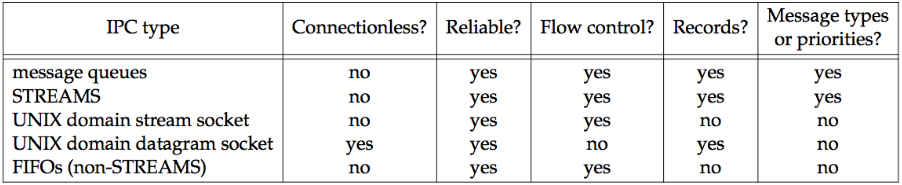
- IPC identifiers are kernel-persistent; `fd`s are process persistent
- IPC identifiers are globally visible; `fd`s are restricted to related processes

Advantages and Disadvantages

Fundamental problem with XSI IPC:

1. IPC structures are systemwide and do not have a reference count
   - After creating a message queue, place some messages on the queue, and the terminate, the message queue and its contents ar not deleted
   - They remain until specifically read or deleted by `msgrcv` or `msgctl`, by executing the `ipcrm(1)` command, or by system being rebooted
2. IPC structures are not kwown by names in the file system
   - new system calls required (`msgget`, `semop`, `shmat`, and so on)
   - can't see the IPC objects with `ls` command or remove them with `rm`; instead two new commands—`ipcs(1)` and `ipcrm(1)`—were added

Advantages and Disadvantages



IPC type	Connectionless?	Reliable?	Flow control?	Records?	Message types or priorities?
message queues	no	yes	yes	yes	yes
STREAMS	no	yes	yes	yes	yes
UNIX domain stream socket	no	yes	yes	no	no
UNIX domain datagram socket	yes	yes	no	yes	no
FIFOs (non-STREAMS)	no	yes	yes	no	no

Figure: Comparison of features of various forms of IPC

By "connectionless," we mean the ability to send a message without having to call some form of an open function first

"Flow control" means that the sender is put to sleep if there is a shortage of system resources (buffers) or if the receiver can't accept any more messages.

XSI IPC : XSI Message Queues

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`.
- New messages are added to the end of a queue by `msgsnd`.
- Messages are fetched from a queue by `msgrcv`.
- We don't have to fetch the messages in a first-in, first-out order. Instead, we can fetch messages based on their type field.

Message Qeues Datastructure

```
struct msqid_ds {
  struct ipc_perm msg_perm; /* see Section 15.6.2 */
  msgqnum_t msg_qnum; /* # of messages on queue */
  msglen_t msg_qbytes; /* max # of bytes on queue */
  pid_t msg_lspid; /* pid of last msgsnd() */
  pid_t msg_lrpid; /* pid of last msgrcv() */
  time_t msg_stime; /* last-msgsnd() time */
  time_t msg_rtime; /* last-msgrcv() time */
  time_t msg_ctime; /* last-change time */
. };
```

Message Queues: `msgget`

`msgget` to either open an existing queue or create a new queue.

```
#include <sys/msg.h>
int msgget(key_t key, int flag);
// Returns: message queue ID if OK, 1 on error
```

- The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of `flag`.
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
- `msg_ctime` is set to the current time.
- `msg_qbytes` is set to the system limit.

On success, `msgget` returns the non-negative queue ID. This value is then used with the other three message queue functions.

Message Queues: msgctl

The msgctl function performs various operations on a queue

```
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msqid_ds *buf );
// Returns: 0 if OK, 1 on error
```

cmd specifies the command to be performed on the queue specified by *msqid*

- IPC_STAT: Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
- IPC_SET: Copy the msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes fields from the structure pointed to by buf to the msqid_ds structure associated with this queue.
- IPC_RMID: Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of EIDRM on its next attempted operation on the queue.

Message Queues: msgsnd

Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
// Returns: 0 if OK, 1 on error
```

Messages are always placed at the end of the queue.

ptr argument is a pointer to a mymesg structure.

- The message type can be used by the receiver to fetch messages in an order other than first in, first out.

```
struct mymesg {
    long mtype; /* positive message type */
    char mtext[512]; /* message data, of length nbytes */
};
```

Message Queues: msgsnd

Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
// Returns: 0 if OK, 1 on error
```

flag value of IPC_NOWAIT can be specified (similar to nonblocking I/O flag for file I/O)

- If IPC_NOWAIT is set and message queue is full: msgsnd to return immediately with an error of EAGAIN
- If IPC_NOWAIT is not set, we are blocked until there is room for the message, the queue is removed from the system, or a signal is caught and the signal handler returns. EINTR is returned

Since reference count is not maintained, removal of a queue simply generates error on the next queue operation

Message Queues: msgsnd

Data is placed onto a message queue by calling msgsnd.

```
#include <sys/msg.h>
int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
// Returns: 0 if OK, 1 on error
```

When msgsnd returns successfully, the msqid_ds structure associated with the message queue is updated

- to indicate the process ID that made the call (msg_lspid),
- the time that the call was made (msg_stime),
- and that one more message is on the queue (msg_qnum).

Message Queues: msgrcv

Messages are retrieved from a queue by msgrcv.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
// Returns: size of data portion of message if OK, 1 on error
```

As with msgsnd, the *ptr* argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data.

- if returned message is larger than *nbytes*, MSG_NOERROR is set: the message is truncated
- if returned message is larger than *nbytes*, MSG_NOERROR is not set: error E2BIG is returned

Message Queues: `msgrcv`

Messages are retrieved from a queue by `msgrcv`.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
// Returns: size of data portion of message if OK, 1 on error
```

type lets us specify which message we want

- `type == 0` : The first message on the queue is returned.
- `type > 0` : The first message on the queue whose message type equals *type* is returned.
- `type < 0` : The first message on the queue whose message type is the lowest value less than or equal to the absolute value of *type* is returned.

A nonzero type is used to read the messages in an order other than first in, first out.

Message Queues: msgrcv

Messages are retrieved from a queue by msgrcv.

```
#include <sys/msg.h>
ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
// Returns: size of data portion of message if OK, 1 on error
```

We can specify a flag value of IPC_NOWAIT to make the operation nonblocking,

When msgrcv succeeds, the kernel updates the msqid_ds structure associated with the message queue to indicate

- the caller's process ID (msg_lrpid),
- the time of the call (msg_rtime),

Message Queue Example I

codes/msg_send.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>

#define MSGSZ 128

/* the message structure. */
typedef struct mymesg {
        long mtype;
        char mtext[MSGSZ];
        } message_buf;

int main()
{
    int msqid;
    int msgflg = IPC_CREAT | 0666;
    key_t key;
    message_buf sbuf;
```

Message Queue Example II

```
    size_t buf_length;

    if((key = ftok("Makefile", 'R')) == -1){
        perror("ftok");
        exit(1);
    }

    fprintf(stderr, "\nmsgget: Calling msgget(%#x, %#o)\n", key, msgflg);

    if ((msqid = msgget(key, msgflg )) < 0) {
        perror("msgget");
        exit(1);
    }
    fprintf(stderr,"msgget: msgget succeeded: msqid = %d\n", msqid);

    /* We'll send message type 1 */
    sbuf.mtype = 1;
    printf("Enter a message to add to the message queue: ");
    scanf("%[^\n]", sbuf.mtext);
    getchar();

    buf_length = strlen(sbuf.mtext) + 1 ;

    /* Send a message.*/
    if (msgsnd(msqid, &sbuf, buf_length, IPC_NOWAIT) < 0) {
       printf ("%d, %ld, %s, %ld\n", msqid, sbuf.mtype, sbuf.mtext, buf_length);
        perror("msgsnd");
```

Message Queue Example III

```
        exit(1);
    } else
      printf("Message: \"%s\" Sent\n", sbuf.mtext);

    exit(0);
}
```

Message Queue Example IV

codes/msg_receive.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>

#define MSGSZ 128

/* Declare the message structure */

typedef struct msgbuf {
    long mtype;
    char mtext[MSGSZ];
} message_buf;


int main()
{
    int msqid;
    key_t key;
    message_buf rbuf;

    if((key = ftok("Makefile", 'R')) == -1){
        perror("ftok");
```

Message Queue Example V

```
            exit(1);
        }

        if ((msqid = msgget(key, 0666)) < 0) {
            perror("msgget");
            exit(1);
        }

        /* Receive an answer of message type 1. */
        if (msgrcv(msqid, &rbuf, MSGSZ, 1, 0) < 0) {
            perror("msgrcv");
            exit(1);
        }

        /* Print the answer. */
        printf("%s\n", rbuf.mtext);
        exit(0);
    }
```

XSI IPC : XSI Semaphore

Semaphores

Semaphore is a counter 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.

A common form of semaphore is called a *binary semaphore*

- controls a signle resource, and it is initialized to 1

XSI Semahores

XSI semaphores are more complicated than this. Three features contribute to this unnecessary complication.

1. A semaphore is defined as a set of one or more semaphore values. When we create a semaphore, we specify the number of values in the set.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`). We cannot atomically create a new semaphore set and initialize all the values in the set.
3. Since all forms of XSI IPC remain in existence even when no process is using them, we have to worry about a program that terminates without releasing the semaphores it has been allocated.

XSI Sempahores Data Structures

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 */
    .
    .
    .
};
```

XSI Sempahores Data Structures

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 */
    .
    .
    .
};
```

XSI Semaphores: semget

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 ipc_perm structure is initialized as described in Section 15.6.2. The mode member of this structure is set to the corresponding permission bits of flag.
- sem_otime is set to 0.
- sem_ctime is set to the current time.
- sem_nsems is set to *nsems*.
    - The number of semaphores in the set is nsems.
    - If a new set is being created (typically by a server), we must specify nsems.
    - If we are referencing an existing set (a client), we can specify nsems as 0.

XSI Semaphores: semctl

The semctl function is the catchall for various semaphore operations.

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

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 */ };
```

XSI Semaphores: `semctl`

The *cmd* argument specifies one of the following ten commands to be performed on the set specified by semid.

- `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.
- `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.
- `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*.

XSI IPC : XSI Shared Memory

XSI 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 kernel maintains a structure with at least the following members for each shared memory segment:

```
struct shmid_ds {
  struct ipc_perm shm_perm; /* see Section 15.6.2 */
  size_t shm_segsz; /* size of segment in bytes */
  pid_t shm_lpid; /* pid of last shmop() */
  pid_t shm_cpid; /* pid of creator */
  shmatt_t shm_nattach; /* number of current attaches */
  time_t shm_atime; /* last-attach time */
  time_t shm_dtime; /* last-detach time */
  time_t shm_ctime; /* last-change time */
  .
};
```

XSI Shared Memory: shmget

The first function called is usually shmget, 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
```

- The ipc_perm structure is initialized as described in Section 15.6.2. The mode member of this structure is set to the corresponding permission bits of flag.
- shm_lpid, shm_nattch, shm_atime, and shm_dtime are all set to 0.
- shm_ctime is set to the current time.
- shm_segsz is set to the *size* requested
- The *size* parameter is the size of the shared memory segment in *bytes*

XSI Shared Memory: `shmctl`

The `shmctl` function is the catchall for various 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 following five commands to be performed, on the segment specified by *shmid*.

- `IPC_STAT`: Fetch the `shmid_ds` structure for this segment, storing it in the structure pointed to by *buf*.
- `IPC_SET`: Set the following three fields from the structure pointed to by *buf* in the `shmid_ds` structure associated with this shared memory segment: `shm_perm.uid`, `shm_perm.gid`, and `shm_perm.mode`.
- `IPC_RMID`: Remove the shared memory segment set from the system. Since an attachment count is maintained for shared memory segments (the `shm_nattch` field in the `shmid_ds` structure), the segment is not removed until the last process using the segment terminates or detaches it. Regardless of whether the segment is still in use, the segment's identifier is immediately removed so that shmat can no longer attach the segment.

XSI Shared Memory: attaching

Once a shared memory segment has been created, a process attaches it to its address space 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
```

The address in the calling process at which the segment is attached depends on the *addr* argument and whether the SHM_RND bit is specified in *flag*.

- If *addr* is 0, the segment is attached at the first available address selected by the kernel. **This is the recommended technique.**
- If *addr* is nonzero and SHM_RND is not specified, the segment is attached at the address given by *addr*.
- If *addr* is nonzero and SHM_RND is specified, the segment is attached at the address given by (*addr*-(*addr* modulus SHMLBA)).

XSI Shared Memory: detaching

When we're done with a shared memory segment, we call shmdt to detach it.

```
#include <sys/shm.h>
int shmdt(const void *addr);
// Returns: 0 if OK, 1 on error
```

The *addr* argument is the value that was returned by a previous call to `shmat`. If successful, `shmdt` will decrement the `shm_nattch` counter in the associated `shmid_ds` structure.

Shared Memory Example: codes/tshm.c I

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600 /* user read/write */

char array[ARRAY_SIZE]; /* uninitialized data = bss */

int
main(void)
{
   int  shmid;
   char *ptr, *shmptr;

   printf("array[] from %p to %p\n", (void *)&array[0],
     (void *)&array[ARRAY_SIZE]);
   printf("stack around %p\n", (void *)&shmid);

   if ((ptr = malloc(MALLOC_SIZE)) == NULL){
      perror("malloc error");
      exit(1);
   }
   printf("malloced from %p to %p\n", (void *)ptr,
```

Shared Memory Example: `codes/tshm.c` II

```
      (void *)ptr+MALLOC_SIZE);

   if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0){
      perror("shmget error");
      exit(1);
   }
   if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1){
      perror("shmat error");
      exit(1);
   }
   printf("shared memory attached from %p to %p\n", (void *)shmptr,
      (void *)shmptr+SHM_SIZE);

   if (shmctl(shmid, IPC_RMID, 0) < 0){
      perror("shmctl error");
      exit(1);
   }

   exit(0);
}
```

Shared Memory Example: `codes/tshm.c` III

```
$ ./tshm
array[] from 0x1085d3050 to 0x1085dcc90
stack around 0x7fff5762cf88
malloced from 0x7f8212802000 to 0x7f821281a6a0
shared memory attached from 0x108611000 to 0x1086296a0
```

Shared Memory Example: `codes/tshm.c`

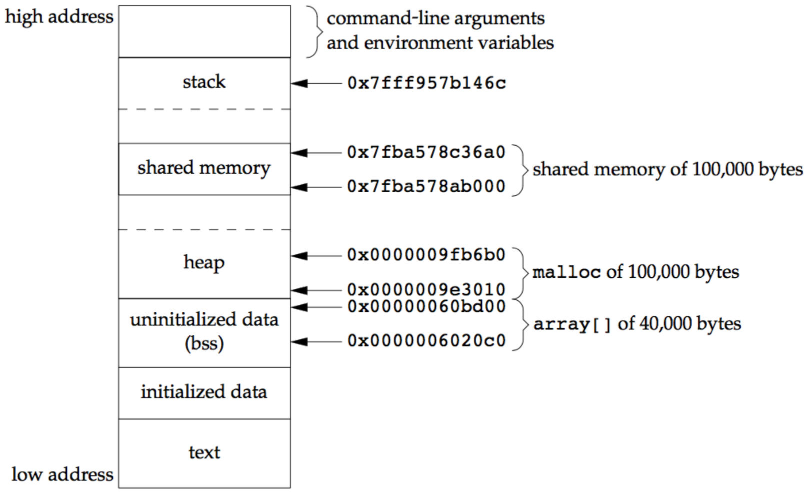


Figure: Memory layout on an Intel-based Linux System

Shared Memory Example: Server - Client

codes/shm_send.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHSIZE 100

int main(int argc, char *argv[])
{
   int shmid;
   key_t key;
   char *shm, *s;
   char *message = "The server sends a line of text0";

   // generate key
   if ( (key = ftok("Makefile", 'R')) == -1 )
   {
      perror("ftok");
      exit(1);
   }

   // connect to the segment
   if ( (shmid = shmget( key, SHSIZE, IPC_CREAT | 0666 )) < 0 )

      perror( "shmget" );
      exit(1);
```

Last Words

Last Words

- prepare for the exam