INTERPROCESS COMMUNICATION

UNIT - V

Outline

- Pipes
- popen & pclose functions
- Coprocesses
- FIFOs
- Message Queues
- Semaphores
- Shared Memory

INTRODUCTION

IPC (InterProcess Communication)

- It is a mechanism whereby two or more processes communicate with each other to perform tasks.
- These processes may interact in a client/server manner or in a peer-to-peer fashion.
- IPC enables one application to control another application, and for several applications to share the same data without interfering with one another.
- IPC is required in all multiprocessing systems, but it is not generally supported by single-process operating systems.

INTRODUCTION

IPC (InterProcess Communication)

The various forms of IPC that are supported on a UNIX system are as follows :

1. Half duplex Pipes	2. Full duplex Pipes
3. FIFO’s	4. Named full duplex Pipes
5. Message queues	6. Shared memory
7. Semaphores	8. Sockets 9. STREAMS

The first seven forms of IPC are usually restricted to IPC between processes on the same host.

The final two i.e. Sockets and STREAMS are the only two that are generally supported for IPC between processes on different hosts.

PIPES

- Pipes are the oldest form of UNIX System IPC.
- Pipes have two limitations.

1. Historically, they have been half duplex (i.e.data flows in only one direction)
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.

PIPES

- A pipe is created by calling the pipe function.

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

Returns: 0 if OK, –1 on error

- Despite these limitations, half-duplex pipes are still the most commonly used form of IPC.
- Every time we type a sequence of commands in a pipeline for the shell to execute, the shell creates a separate process for each command and links the standard output of one to the standard input of the next using a pipe.

PIPES

Two file descriptors are returned through the fd argument:

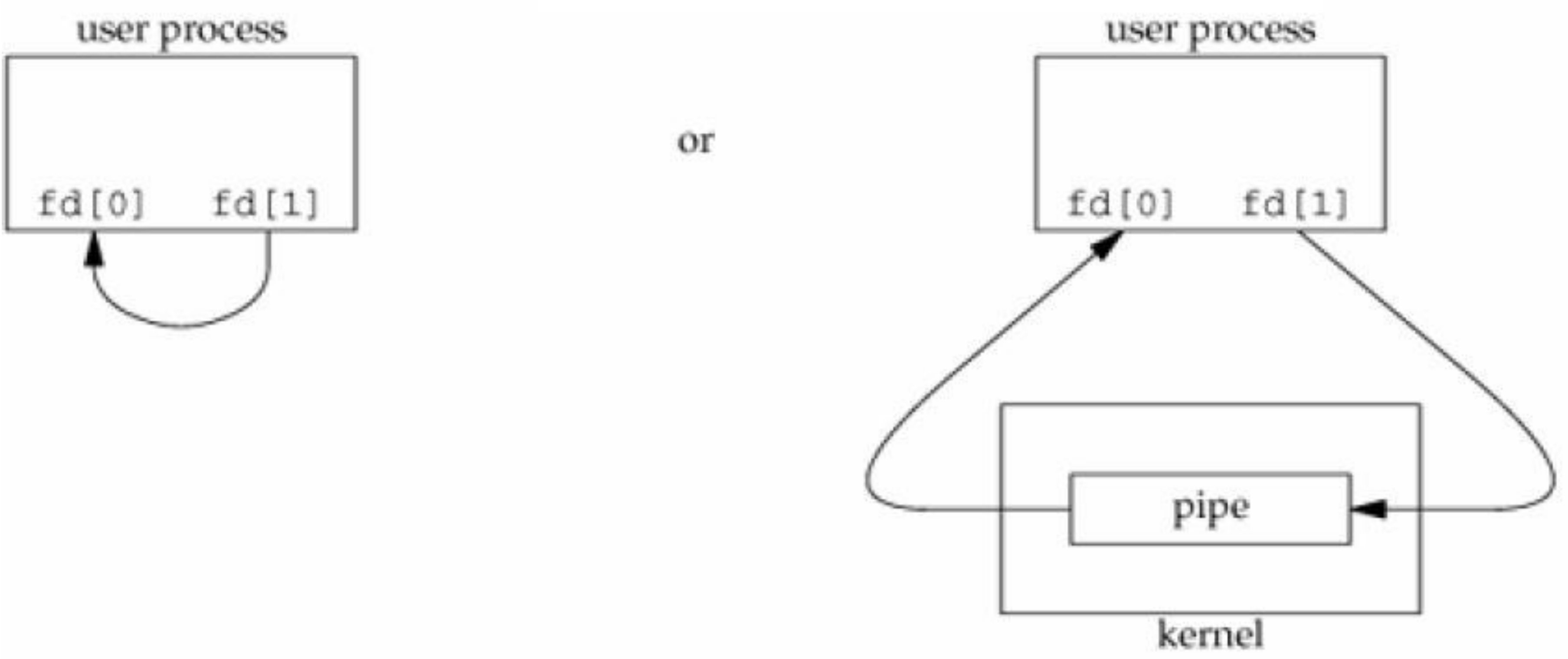
fd[0] is open for reading, and
fd[1] is open for writing.

The output of fd[1] is the input for fd[0].

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

PIPES

Two ways to picture a half-duplex pipe are shown in the given diagrams below:

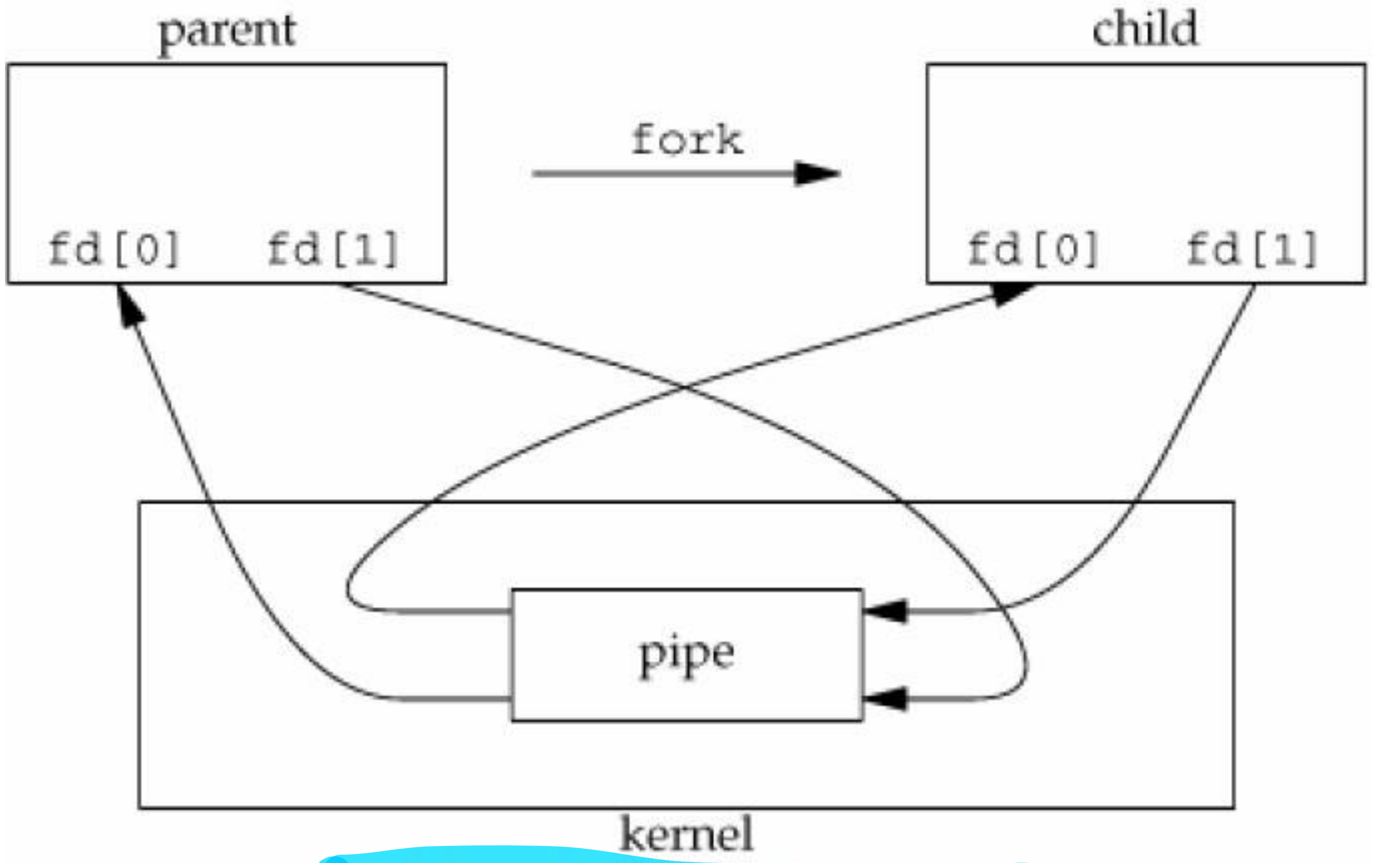


Two ways to view a half-duplex pipe

PIPES

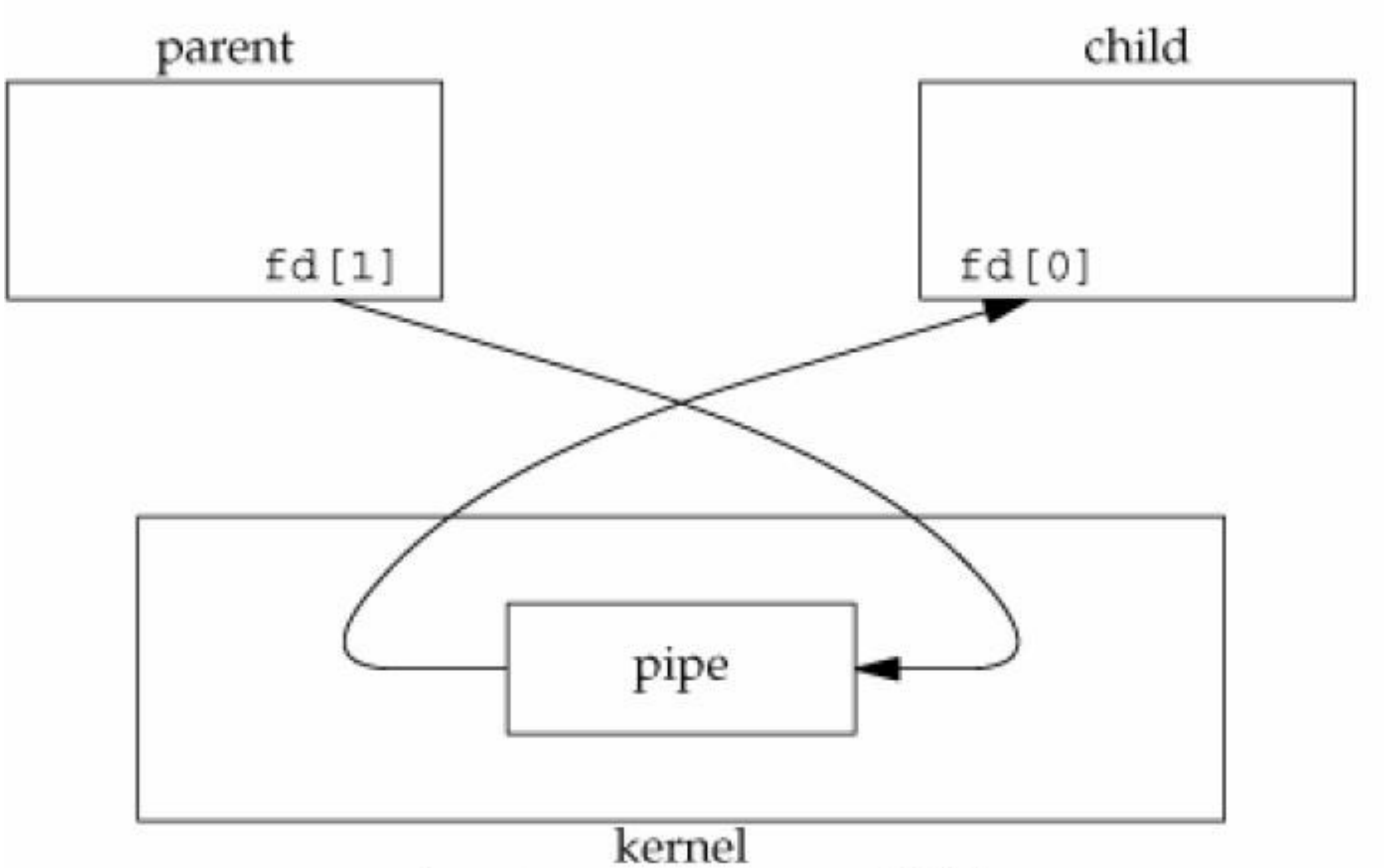
- The left half of the diagram shows the two ends of the pipe connected in a single process.
- The right half of the diagram emphasizes that the data in the pipe flows through the kernel.
- A pipe in a single process is next to useless.
- Normally, the process that calls pipe then calls fork, creating an IPC channel from the parent to the child or vice versa.

PIPES



Half-duplex pipe after a `fork`

PIPES



Pipe from parent to child

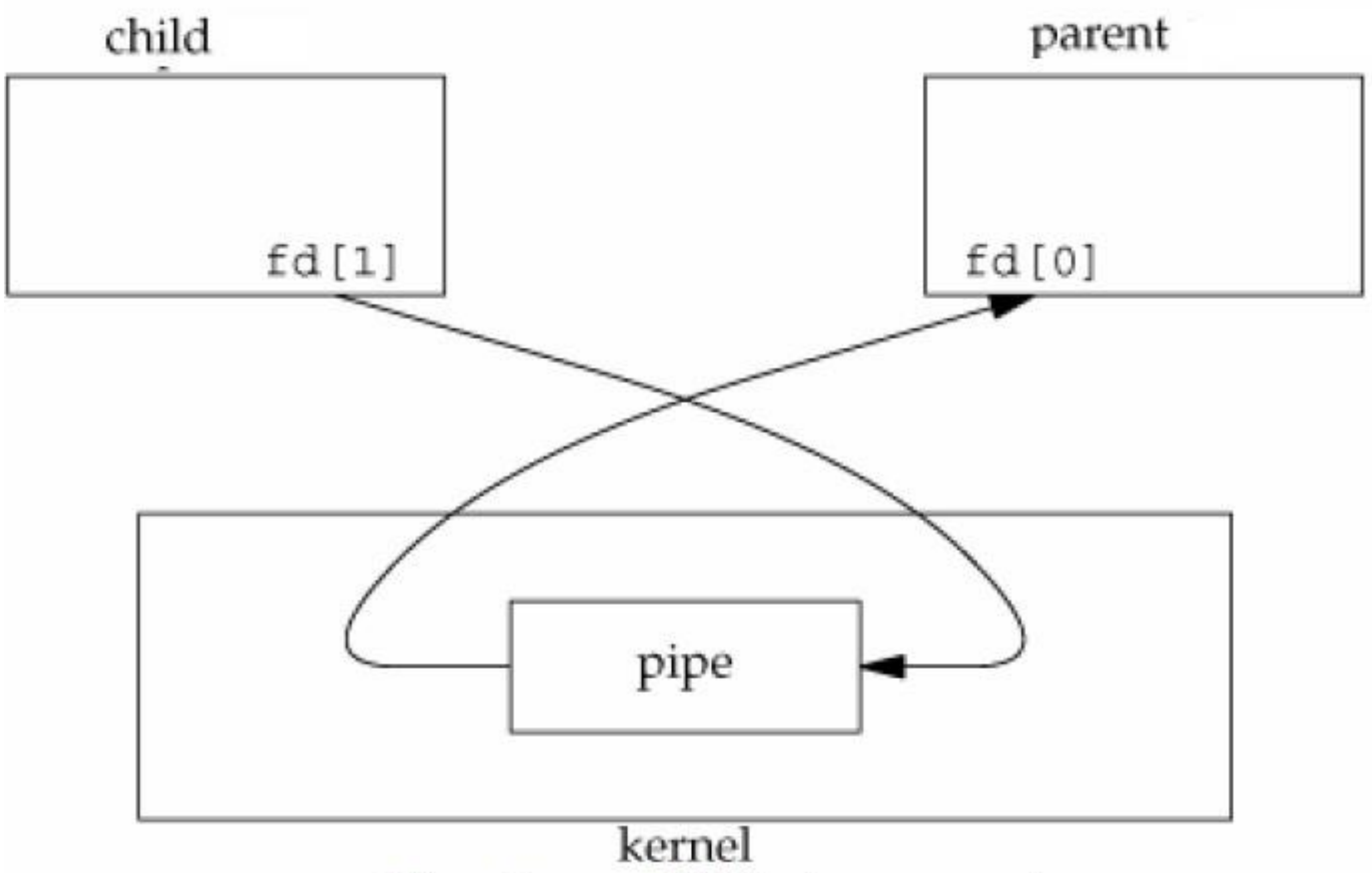
For a pipe from the parent to the child, the parent closes the read end of the pipe (fd[0]), and the child closes the write end (fd[1]).

PIPES

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].

PIPES



Pipe from child to parent

For a pipe from the child to the parent, the parent closes fd[1], and the child closes fd[0].

PIPES

When one end of a pipe is closed, the following 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. If we either ignore the signal or catch it and return from the signal handler, write returns –1 with errno set to EPIPE.

PIPES

Program to show the code to create a pipe between a parent and its child and to send data down the pipe.

```
int main(void)
{
int n, fd[2];
pid_t pid;
char line[1000];
if (pipe(fd) < 0)
perror("pipe error");
if ((pid = fork()) < 0)
   perror("fork error");
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,1000);
write(1, line, n);
}
exit(0);
}
```

PIPES

Program to show the code for I/O redirection using dup2().

```
#include<fcntl.h>
main()
{
int fd,fd2;
char c[256],con[]="This is simple file
for demonstration";
fd=open("sample1",O_RDWR|O_CREAT,0777);
printf("original file desc is %d\n",fd);
fd2=dup2(fd,7);
printf("new file desc is %d\n", fd2);
if(fd==-1)
 perror("Can't creat file");
else if(read(fd2,&c,10)==0)
        write(fd2,con,sizeof(con));
else
   printf("%s\n",c);
close(fd);
}
```

PIPES

Program to show the code for I/O redirection using dup().

```
#include<fcntl.h>
main()
{
int fd,fd2;
char c[256],con[]="This is simple file
for demonstration";
fd=open("sample1",O_RDWR|O_CR
EAT,0777);
printf("original file desc is %d\n",fd);
fd2=dup(fd);
printf("new file desc is %d\n", fd2);
if(fd==-1)
 perror("Can't creat file");
else if(read(fd2,&c,10)==0)
        write(fd2,con,sizeof(con));
else
   printf("%s\n",c);
close(fd);
}
```

PIPES

Program to show the code child writes to the parent through pipe.

```
#include<unistd.h>
#include<sys/types.h>
main()
{
pid_t pid;
int fd[2],s;
char c[5];
pipe(fd);
pid=fork();
if (pid==0)
{
close(fd[0]);
write(fd[1],"hello",5);
}
else
{
wait(&s);
close(fd[1]);
read(fd[0],c,5);
c[5]='\0';
printf("%s\n",c);
}
}
```

PIPES

Program to show the code for broken pipe.

```
#include<unistd.h>
#include<sys/types.h>
main()
{
pid_t pid;
int fd[2];
char c[5];
pipe(fd);
pid=fork();
if (pid>0)
{
close(fd[0]);
write(fd[1],"hello",5);
}
else
{
close(fd[0]); /*read end closed*/
sleep(1);
read(fd[0],c,5);
printf("%s\n",c);
exit(0);
}
}
```

PIPES

Program to show the code for UNIX command redirection for “ls|wc -l”.

```
#include<fcntl.h>
main()
{
int p[2],pid;
pipe(p);
pid=fork();
if(pid==0)
{
close(p[0]);
printf("p[1]=%d\n",p[1]);
dup2(p[1],1);
execl("/bin/ls","ls",(char *)0);
perror("from ls:");
}
else
{
close(p[1]);
printf("p[0]=%d\n",p[0]);
dup2(p[0],0);
execl("/usr/bin/wc","wc","-l",(char *)0);
perror("from wc");
}
}
```

PIPES

Program to **implement unix command “who|sort|wc –l”**

```
main()
{
int p[2], q[2], pid, pid1;
pipe(p);
pid = fork();
if(0 == pid)
{
  close(1);
 close(p[0]);
 dup(p[1]);
 execlp("who", "who",0);
 perror("error at cat");
 }
```

```
else
{
  pipe(q);
pid1 = fork();
if(0 == pid1)
{
   close(1);
   close(0);
   close(p[1]);
   close(q[0]);
   dup(p[0]);
   dup(q[1]);
   execlp("sort", "sort", (char*)0);
   perror("error at grep");
}
```

Popen() and pclose() Functions

- Since 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 dirty work that we've been doing ourselves: 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.

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

Popen() and pclose() Functions

- 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

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

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

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

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.

If the shell cannot be executed, the termination status returned by pclose is as if the shell had executed exit(127).

The cmdstring is executed by the Bourne shell, as in

$ sh -c cmdstring

This means that the shell expands any of its special characters in cmdstring. Example:

fp = popen("ls *.c", "r"); or fp = popen("cmd 2>&1", "r");

PIPES

Program to sort the strings using popen().

```
#define MAXSTRS 4
int main(void)
{
int i;
FILE *pipe_fp;
char *strings[MAXSTRS] =
{ "echo", "bravo", "alpha", "charlie"};
/* Create 1 way pipe line with call to popen() */
pipe_fp = popen("sort", "w");
if (pipe_fp== NULL)
{
perror("popen");
exit(1);
}
/* Processing loop */
for(i=0; i<MAXSTRS; i++)
{
fputs(strings[i], pipe_fp);
fputc('\n', pipe_fp);
}
/* Close the pipe */
pclose(pipe_fp);
return(0);
}
```

PIPES

Program to implement UNIX command redirection for “ls|sort” using popen().

```
int main(void)
{
FILE *pipein_fp, *pipeout_fp;
char readbuf[80];
/* Create one way pipe line with call to popen() */
pipein_fp = popen("ls", "r");
if (pipein_fp== NULL)
{
perror("popen");
exit(1);
}
/* Create one way pipe line with call to popen() */
pipeout_fp = popen("sort", "w“);
if (pipeout_fp == NULL)
{
perror("popen");
exit(1);
 }
/* Processing loop */
while(fgets(readbuf, 80, pipein_fp))
    fputs(readbuf, pipeout_fp);
 /* Close the pipes */
pclose(pipein_fp);
pclose(pipeout_fp);
return(0); }
```

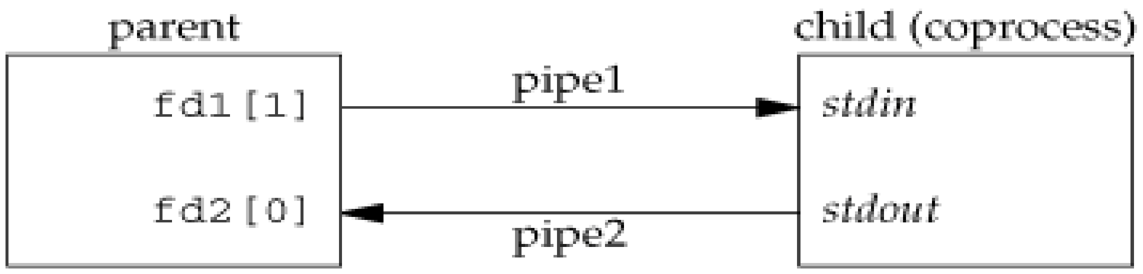
Coprocesses

- 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's output.
- The Korn shell provides coprocesses.
- The Bourne shell, the Bourne-again shell, and the C shell don't provide a way to connect processes together as coprocesses.
- A coprocess normally runs in the background from a shell, and its standard input and standard output are connected to another program using a pipe.

Coprocesses

- Whereas popen gives us a one-way pipe to the standard input or from the standard output of another process.
- With a coprocess, we have two one-way pipes to the other process: one to its standard input and one from its standard output.

- The process creates two pipes: one is the standard input of the coprocess, and the other is the standard output of the coprocess.
- The diagram shows this arrangement.



FIFOs

- FIFOs are sometimes called named pipes.
- 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.
- Indeed, the pathname for a FIFO exists in the file system.

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

int mkfifo(const char *pathname, mode_t mode);
```

Returns: 0 if OK, –1 on error

FIFOs

- Once we have used mkfifo to create a FIFO, we open it using open().
- When we open a FIFO, the non blocking flag (O_NONBLOCK) affects what happens:
  1. In the normal case (O_NONBLOCK not specified), an open for read-only blocks until some other process opens the FIFO for writing. Similarly, an open for write-only blocks until some other process opens the FIFO for reading
  2. If O_NONBLOCK is specified, an open for read-only returns immediately. But an open for write-only returns 1 with errno set to ENXIO if no process has the FIFO open for reading.

FIFOs

- There are two uses for FIFOs.

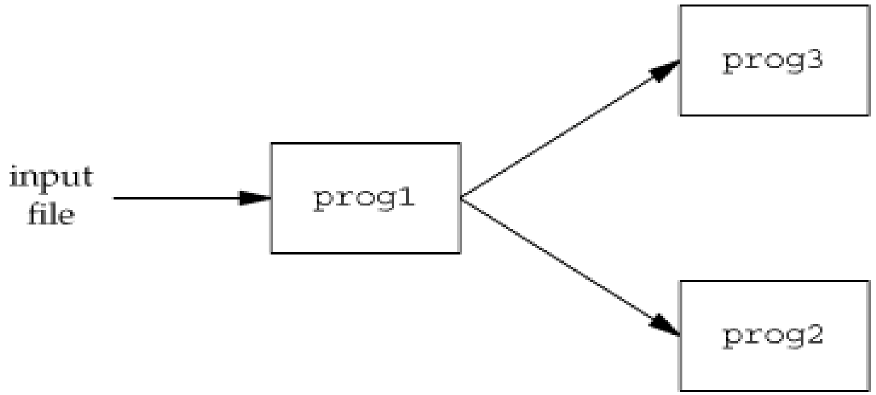
1. FIFOs are used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files.

2. FIFOs are used as rendezvous points in client–server applications to pass data between the clients and the servers.

FIFOs

Example Using FIFOs to Duplicate Output Streams:

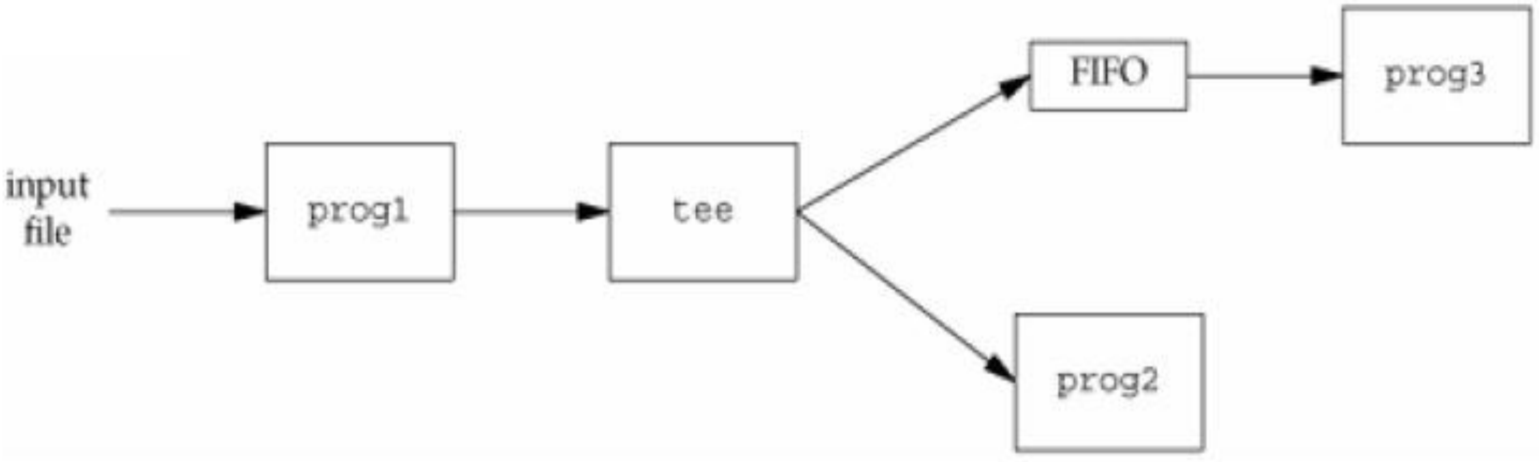
- FIFOs can be used to duplicate an output stream in a series of shell commands.
- This prevents writing the data to an intermediate disk file.
- Consider a procedure that needs to process a filtered input stream twice.



Procedure that processes a filtered input stream twice.

FIFOs

Example Using FIFOs to Duplicate Output Streams:



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

With a FIFO and the UNIX program tee(1), we can accomplish this procedure without using a temporary file.

FIFOs

Example Using FIFOs to Duplicate Output Streams:

- The tee program copies its standard input to both its standard output and to the file named on its command line.

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

- We create the FIFO and then start prog3 in the background, reading from the FIFO.
- We then start prog1 and use tee to send its input to both the FIFO and prog2.

FIFOs

Example Client-Server Communication Using a FIFO

- FIFO’s can be used to send data between a client and a server.
- If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates.
- Since there are multiple writers for the FIFO, the requests sent by the clients to the server need to be less than PIPE_BUF bytes in size.
- This prevents any interleaving of the client writes.
- The problem in using FIFOs for this type of client server communication is how to send replies back from the server to each client.

FIFOs

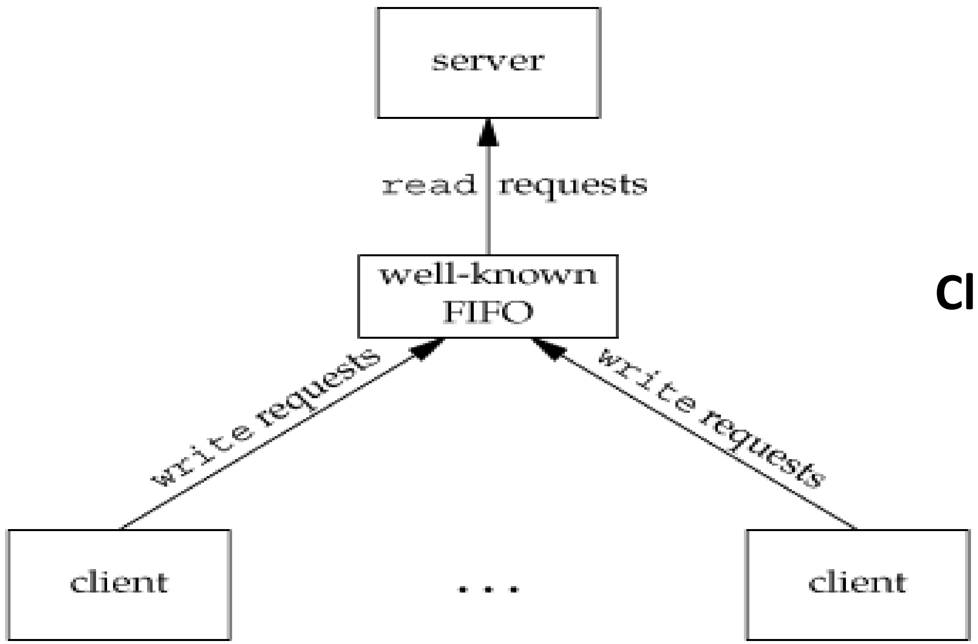
Example Client-Server Communication Using a FIFO

- A single FIFO can’t be used, as the clients would never know when to read their response versus responses for other clients.
- One solution is for each client to send its process ID with the request.
- The server then creates a unique FIFO for each client, using a pathname based on the client’s process ID.
- For example, the server can create a FIFO with the name /home/ser.XXXXX, where XXXXX is replaced with the client’s process ID.
- This arrangement works, although it is impossible for the server to tell whether a client crashes.

FIFOs

Example Client-Server Communication Using a FIFO

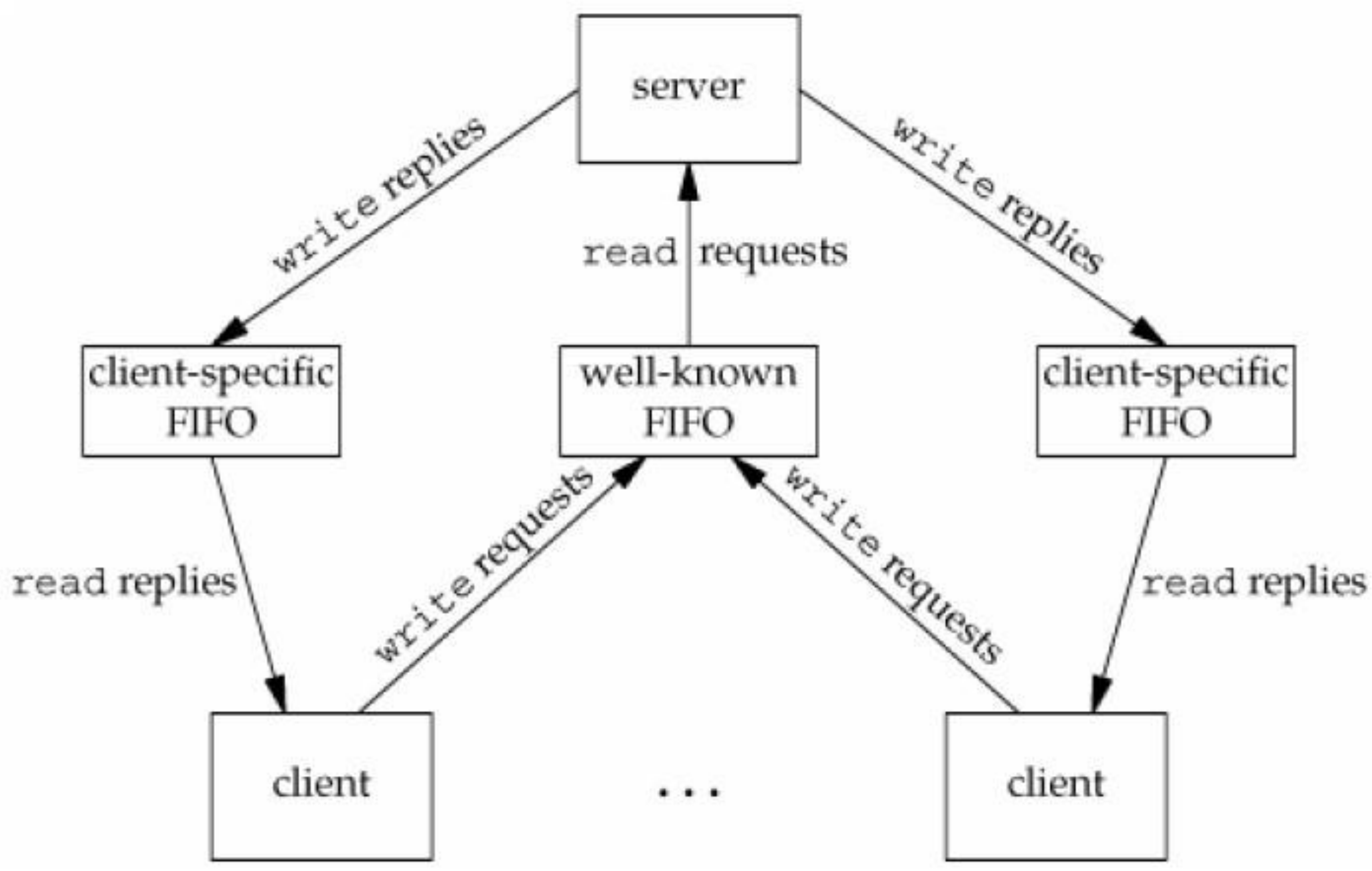
- This causes the client-specific FIFOs to be left in the file system.
- The server also must catch SIGPIPE, since it's possible for a client to send a request and terminate before reading the response, leaving the client-specific FIFO with one writer (the server) and no reader.



Clients sending requests to a server using a FIFO

FIFOs

Example Client-Server Communication Using a FIFO



Client–server communication using FIFOs

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().
- 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 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 Queues

This structure defines the current status of the queue.

The members shown are the ones defined by the Single UNIX Specification.

Each queue has the following msqid_ds structure associated with it:

```
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

The first function normally called is 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

Message Queues

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

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

Message Queues

- Each message is composed of a positive long integer type field, a non-negative length (nbytes), and the actual data bytes (corresponding to the length).
- Messages are always placed at the end of the queue.
- The ptr argument points to a long integer that contains the positive integer message type, and it is immediately followed by the message data.
- There is no message data if nbytes is 0. If the largest message we send is 512 bytes, we can define the following structure:

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

- The ptr argument is then 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.

Message Queues

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

Message Queues

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

int main()
{
 int msqid;

msqid = msgget((key_t)5, IPC_CREAT | IPC_EXCL | 0777);

if(-1 == msqid) {
perror("msgget:");
exit(1);
}

printf("msgid = %d\n",msqid);
}
```

Message Queues

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct msgbuf{
long mtype;
char mtext[40];
};

int main()
{
 int msqid, len, ret;
struct msgbuf msgsend={0,"\0"};

msqid = msgget((key_t)5, IPC_CREAT | 0666);
if(-1 == msqid){
perror("msgget:");
exit(1);
}
```

```c
printf("Enter message type:   \n");
scanf("%d",&msgsend.mtype);
printf("Enter message text\n");
//make use of fgets() if u want to send msg with spaces
scanf("%s", msgsend.mtext);
len = strlen(msgsend.mtext);
ret = msgsnd(msqid, &msgsend, len, 0);

if(-1 == ret)
{
perror("msgsnd:");
exit(1);
}
else
printf("message sent\n");
}
```

Message Queues

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct msgbuf{
long mtype;
char mtext[40];
};
main()
{
 int msqid, len, ret, type;
struct msgbuf msgread={0, "\0"};
fflush(stdin);
msqid = msgget((key_t)5,0);
if(-1 == msqid)
{
perror("msgget:");
exit(1);
}
printf("Enter the message no:\n");
scanf("%d", &type);
len = sizeof(msgread.mtext);
ret = msgrcv(msqid, &msgread, len, type, IPC_NOWAIT );

printf("ret = %d\n", ret);
if(-1 == ret){
perror("msgrcv:");
exit(1);
}
else
printf("message type = %d message text = %s\n",
msgread.mtype, msgread.mtext);
}
```

Semaphores

Semaphore is a counter used to provide access to a shared resource for multiple processes.

To obtain a shared resource, a process need to do the following:

- Test the semaphore that controls the resource.

- 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.

- 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 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.

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

- It controls a single resource, and its value is initialized to 1.

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

The kernel maintains a semid_ds structure for each semaphore set:

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

Each semaphore is represented by an anonymous structure contains at least the following members:

semget

```
#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.

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 current time.

sem_nsems is set to nsems.

semctl

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

int semctl(int semid, int semnum, int  cmd,
           ... /* union semun arg */);
```

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

The fourth argument is optional, depending on the command requested, and if present, is of type *semun,* a union of various command specific arguments:

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

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.

IPC_STAT	Fetch the semid_ds structure, storing it in the structure pointed to by arg.buf
IPC_SET	Set the sem_perm.uid, sem_perm.gid, sem_perm.mode
IPC_RMID	Remove the semaphore set from the system. This removal is immediate.
GETVAL	Return the value of semval for the member semnum
SETVAL	Set the value of semval for the member semnum
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 semaphore values in the set. Values stored in array pointed to by arg.array
SETALL	Set all semaphore values in the set to the values pointed to by arg. array

semop

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```

Returns: 0 if OK, –1 on error

```
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 */
};
```

The *semoparray* argument is a pointer to an array of semaphore operations, represented by sembuf structures:

The *nops* argument specifies the number of operations in the array.

The operation on each member of the set is specified by the corresponding sem_op value.

The value can be negative, 0, or positive.

Case 1 (sem_op is positive)

This case corresponds to the returning of the resources by the process.

The value of sem_op is added to the semaphore's value.

Case 2 (sem_op is negative)

If the semaphore's value is less than the absolute value of sem_op, the following conditions apply.

- If IPC_NOWAIT is specified, semop returns with an error of EAGAIN.
- If IPC_NOWAIT is not specified, the **semncnt** value for this semaphore is incremented and the calling process is suspended until one of the following occurs:
  - The semaphore value becomes greater than or equal to the absolute value of sem_op.
  - The semaphore is removed from the system.
  - A signal is caught by the process, and the signal handler returns.

Case 3 (sem_op is 0)

Means that the calling process wants to wait until the semaphore value becomes 0.

- If the semaphore value is currently 0, the function returns immediately.
- Otherwise,
  - If IPC_NOWAIT is specified, return is made with an error of EAGAIN.
  - If IPC_NOWAIT is not specified, the **semzcnt** value for this semaphore is incremented and the calling process is suspended until one of the following occurs:
    - The semaphore value becomes 0.
    - The semaphore is removed from the system.
    - A signal is caught by the process, and the signal handler returns.

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

The type shmatt_t is defined to be an unsigned integer at least as large as unsigned short.

shmget

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

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

Returns: shared memory ID if OK, –1 on error

ipc_perm structure is initialized

shm_lpid, *shm_nattach*, *shm_atime*, *shm_dtime* are all set to 0

shm_ctime is set to the current time

shm_segsz is set to the size requested

shmctl

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Returns: 0 if OK, –1 on error

IPC_STAT	Fetch the shmid_ds structure for this segment
IPC_SET	Set the shm_perm.uid, shm_perm.gid, and shm_perm.mode
IPC_RMID	Remove the shared memory segment set from the system
SHM_LOCK	Lock the shared memory segment in memory
SHM_UNLOCK	Unlock the shared memory segment

The *cmd* argument specifies one of the following five commands on the segment specified by *shmid.*

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

Once the shared memory segment has been created, a process attaches it to its address space by calling shmat.

If *addr* is 0, the segment is attached at the first available address selected by the kernel.
If *addr* is non zero and SHM_RND is not specified, the segment is attached at the address given by *addr*.
If *addr* is non zero and SHM_RND is specified, the segment is attached at the address given by (*addr* – (*addr* modulus SHMLBA))

shmdt

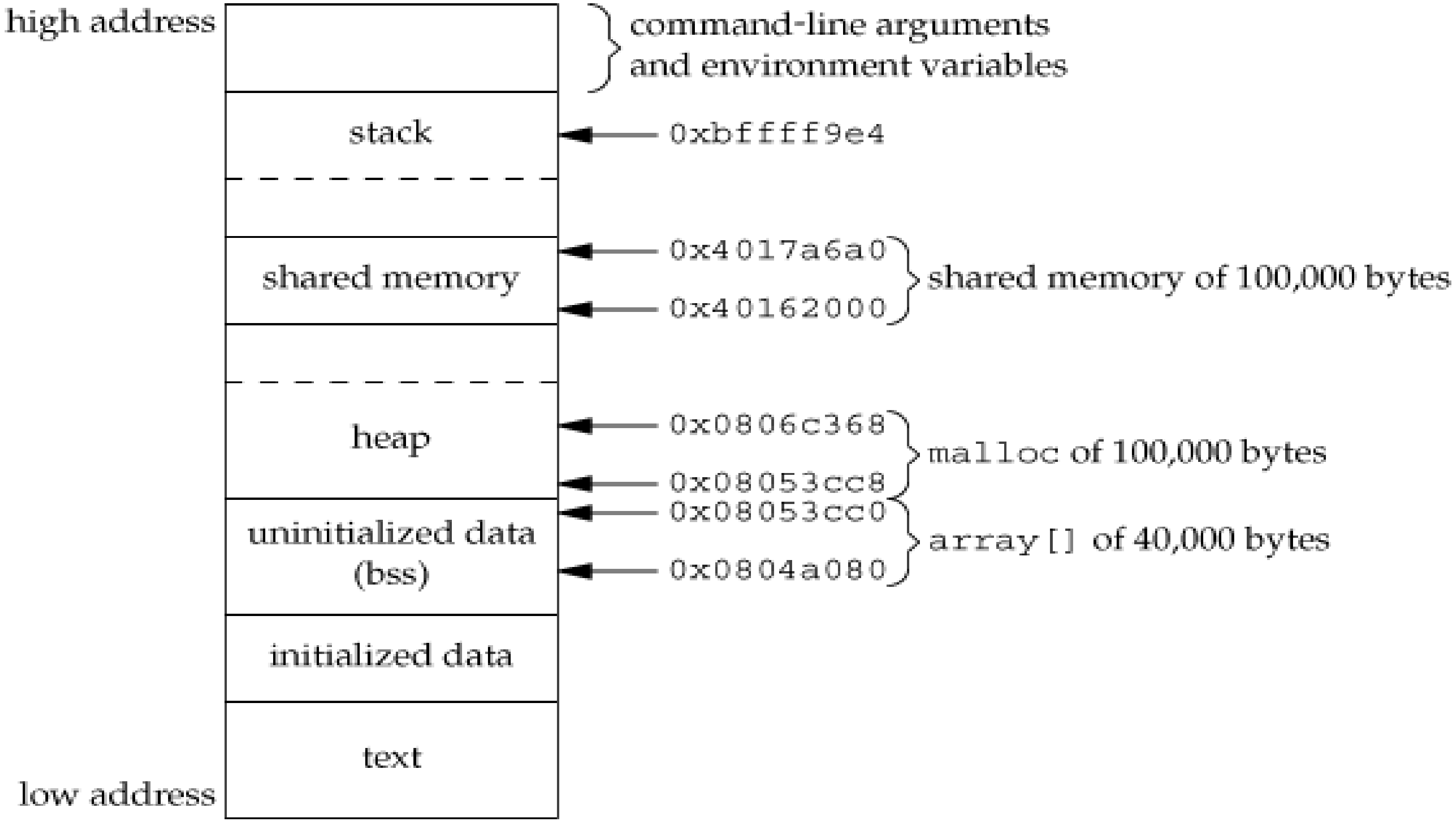
```
#include <sys/shm.h>

int shmdt(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.



Memory layout on an Intel Based Linux System

END