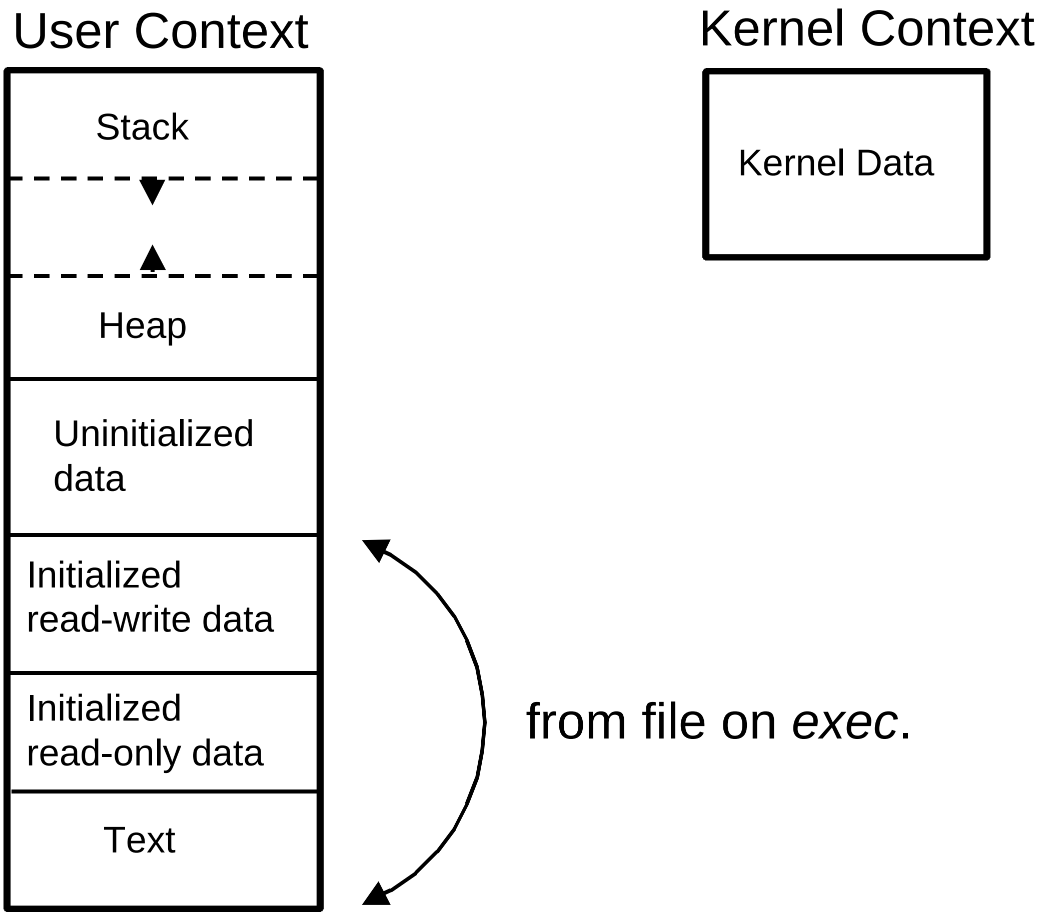
UNIX Inter-Process Communication

Pipes, Messages, Semaphores

Prof. Badri R. Narayanan

UNIX Process Model

- A *process* is an instance of a program in execution.
- Only way to create a new process is the *fork* system call.
- Each process has a *user context* and a *kernel context*.



User Context

Stack

Heap

Uninitialized data

Initialized read-write data

Initialized read-only data

Text

Kernel Context

Kernel Data

from file on *exec*.

Relevant Attributes of Kernel Context of a Process

Attribute	*System Call*
Process Id	getpid()
Parent PID	getppid()
Real User Id	getuid()
Real Group Id	getgid()
Effective UID	geteuid() *set-user-ID*
Effective GID	getegid() *set-group-ID*

UNIX Signals (Subset)

Signals notify processes of events.

- from a process
- from kernel

Name	*Description*	*Default Action*
SIGALRM	Alarm Clock	Terminate
SIGBUS	Bus Error	X with core
SIGCLD	Death of Child Process	Discarded
SIGCONT	Continue after SIGSTOP	Discarded
SIGHUP	Hangup	Terminate
SIGINT	Interrupt character	Terminate
SIGIO	I/O poss. on file desc.	Discarded
SIGKILL	Kill	Terminate
SIGPIPE	Write on pipe without reader	Terminate
SIGQUIT	Quit character	X with core
SIGSTOP	Stop	Stop (suspend)
SIGTSTP	Stop from keyboard	Stop
SIGURG	Urgent cond. on socket	Discarded
SIGUSR1	User defined signal	Terminate
SIGUSR2	User defined signal	Terminate

Generating Signals

Signals are sent in response to conditions.
Note: Signals usually occur
asynchronously.

Condition	*Example*
kill system call	int kill(int *pid*, int *sig*) Send *sig* to *pid.* *Many variations.*
kill command	Issues *kill* system call
Terminal characters	Control-C -- SIGINT Control-Z -- SIGSTP
Hardware Conditions	Floating Point Error -- SIGFPE
Software Conditions	Out-of-band data on socket -- SIGURG

Handling Signals

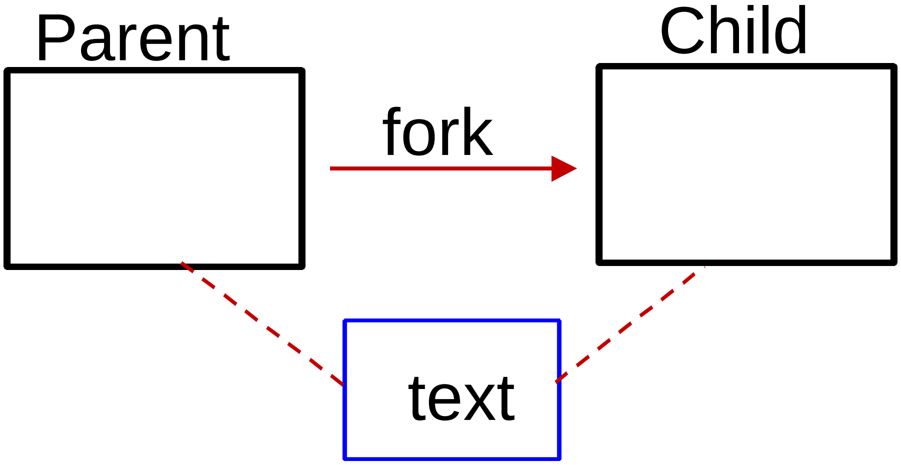
- Process can *catch* signals by providing signal handler functions.
- Process can choose to ignore all signals (except SIGKILL which guarantees termination).
- Process can allow the default action to occur.

Using *signal* system call

```
#include <signal.h>
int (*signal (int sig, void (*func) (int))) (int);

/* SIG_IGN and SIG_DFL */
signal(SIGUSR1, SIG_IGN);
/* User-defined handler */
extern void myintr();
if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, myintr);
```

Process Control

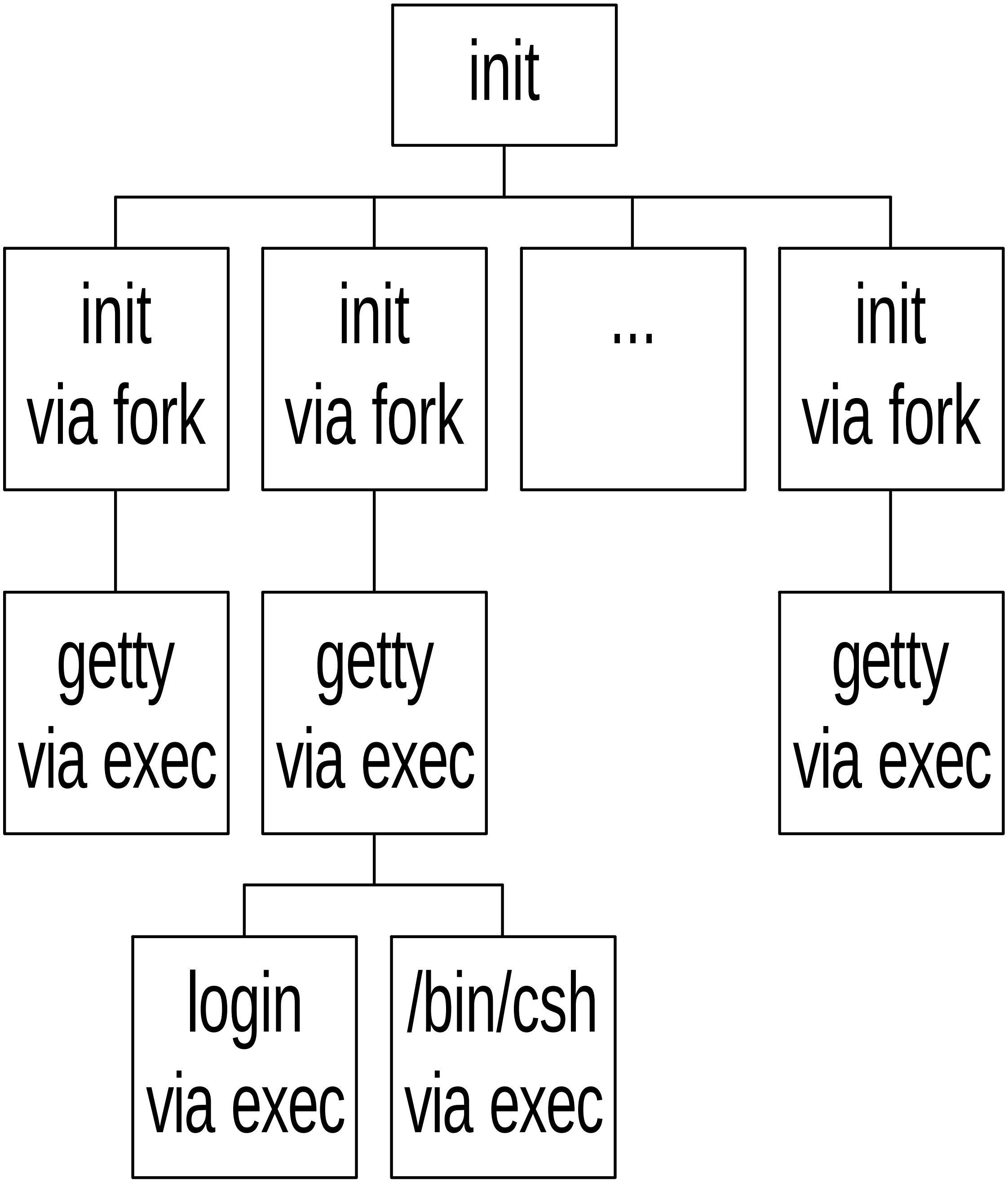
fork, exit, exec, wait



- *fork* is called by parent process.
- It creates an identical child process sharing text, open file handles.
- fork returns child pid to parent.
- fork returns 0 to child process.

```
main()
{       int childpid;
        if ( (childpid = fork()) == -1) {
                perror( "fork failed!");
                exit(1);
        } else
        if (childpid == 0) { /* child process */
        }
        else { /* parent process */
        }
}
```

Process Relationships



init

init
via fork

init
via fork

...

init
via fork

getty
via exec

getty
via exec

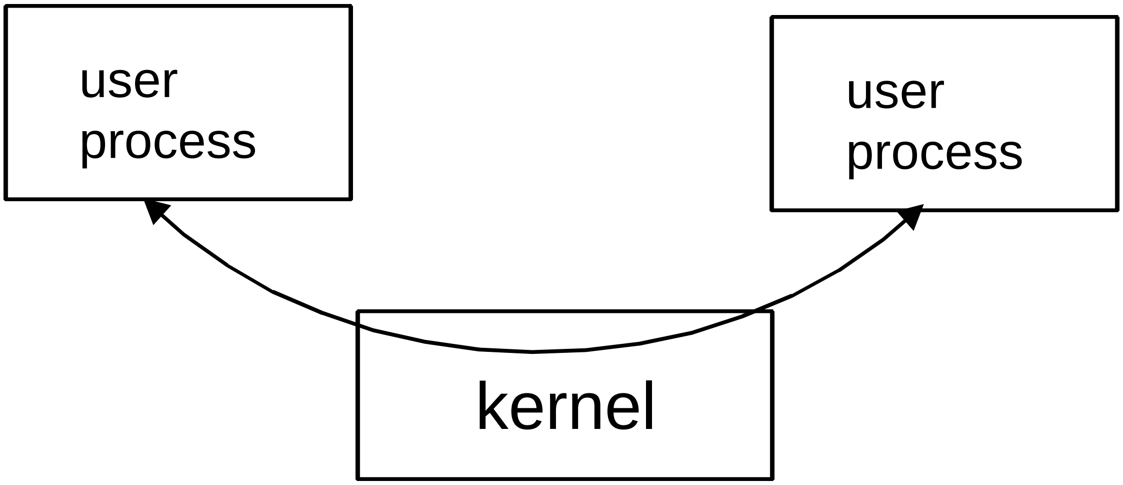
getty
via exec

login
via exec

/bin/csh
via exec

IPC Basics

IPC on single system

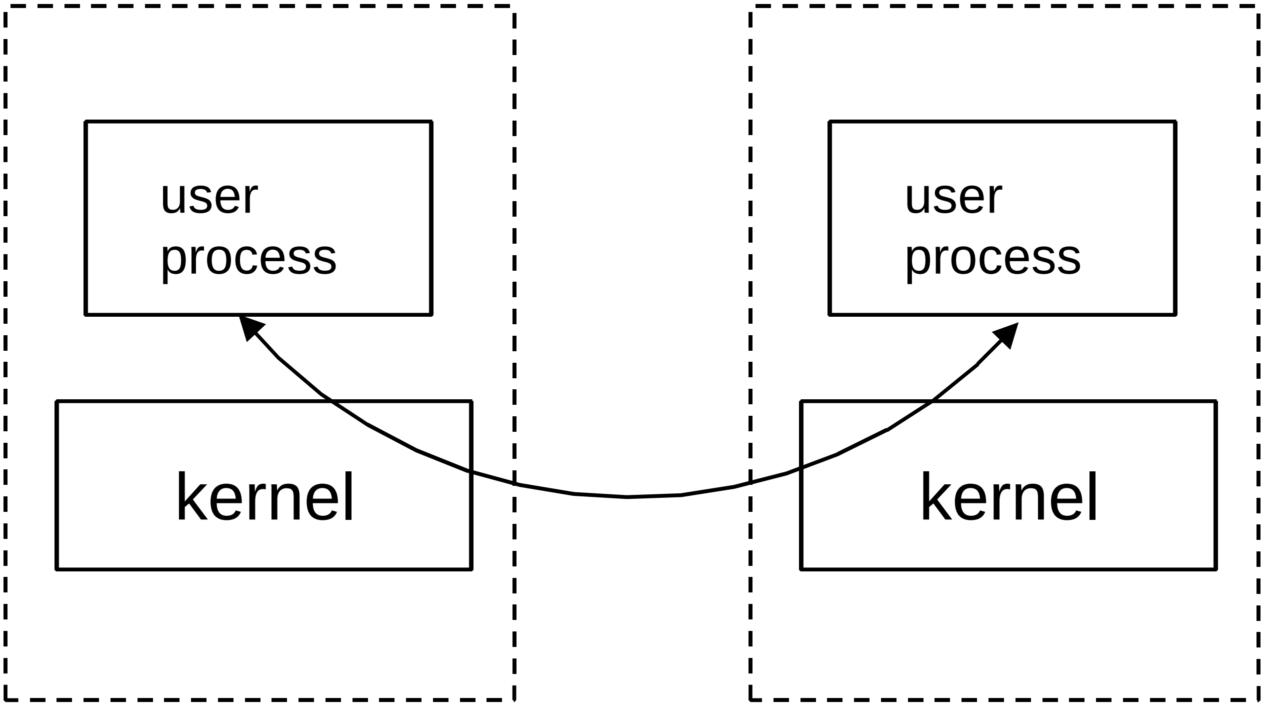


user process

user process

kernel

IPC across networked systems



user process

kernel

user process

kernel

Need for Mutual Exclusion Revisited

Unique sequence number for print job

- Read shared sequence number file
- Use number from file
- Write incremented value to file

```
#define SEQFILE "jobnumber"
#define MAXBUFF 64
main() {
        int fdes, i, n, pid, seqno;
        char buffer[MAXBUFF + 1];

        pid = getpid();
        if ((fdes = open(SEQFILE, 2)) < 0)
                err_sys("Could not open %s", SEQFILE);
        for (i=0; i<5; i++) {
                my_lock(fdes);
                lseek(fdes, 0L, 0);
                if ((n = read(fdes, buff, MAXBUFF)) <= 0)
                 err_sys("Reading error");
                buff[n] = '\0';
                if ((n = sscanf(buff, "&d\n", &seqno)) != 1)
                 err_sys("sscanf error");
                printf("pid = %d, seq# = %d\n", pid, seqno);
                seqno++;
                sprintf(buff, "%03d\n", seqno);
                n = strlen(buff);
                lseek(fdes, 0L, 0);
                if (write(fdes, buff, n) != n)
                    err_sys("Writing error");
                my_unlock(fdes);
        }
}
```

File Locking for Mutual Exclusion

Without Locks

```
my_lock(fdes)
int fdes;
{ return; }

my_unlock(fdes)
int fdes;
{ return; }
```

Run program twice

a.out & a.out &

Possible output

```
pid = 12, seq# = 1
pid = 12, seq# = 2

pid = 13, seq# = 2
pid = 13, seq# = 3
pid = 13, seq# = 4

pid = 12, seq# = 3

pid = 13, seq# = 4
```

File locking allows for safe and exclusive access of files.

With Locks

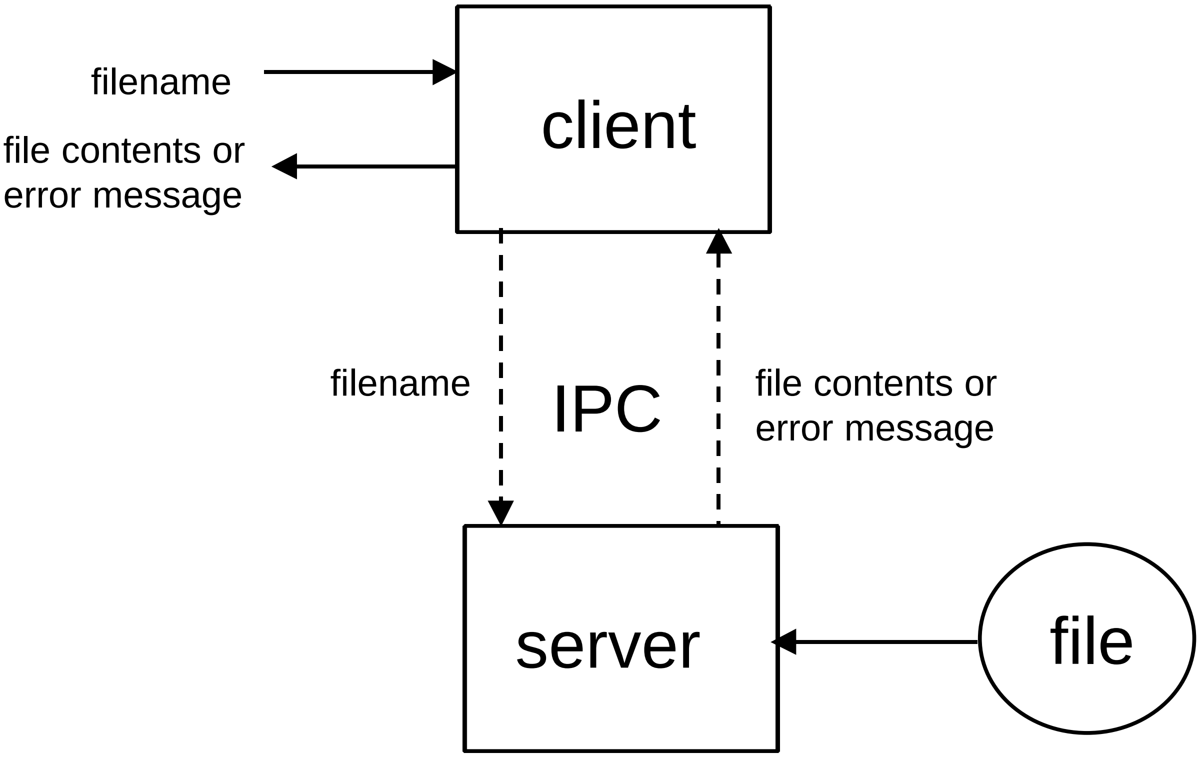
```
#include <sys/file.h>
my_lock(fdes)
int fdes;
{  if (flock(fdes, LOCK_EX) == -1)
         err_sys("Cannot LOCK_EX");
 }

my_unlock(fdes)
int fdes;
{  if (flock(fdes, LOCK_UN) == -1)
err_sys("Cannot LOCK_UN");
 }
```

Possible output

```
pid = 15, seq# = 1
pid = 15, seq# = 2

pid = 16, seq# = 3
pid = 16, seq# = 4
pid = 16, seq# = 5

pid = 15, seq# = 6
pid = 15, seq# = 7

pid = 16, seq# = 8
pid = 16, seq# = 9
```

Client-Server Example

- Client writes filename to IPC channel.
- Server reads filename and
  - either returns contents via IPC
  - or returns opening error via IPC
- Client reads from IPC.



Unix IPC - Pipes

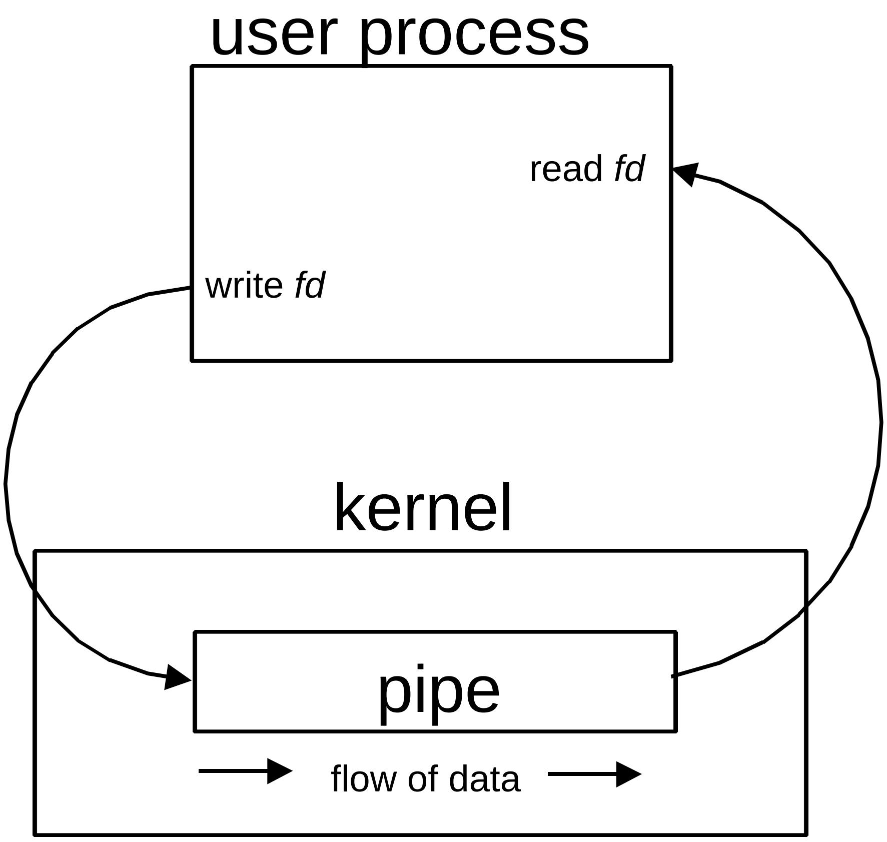
pipe System Call

- One-way flow of data created by

  int pipe(int **filedes*);
- Two file descriptors are returned

  filedes[0] open for reading

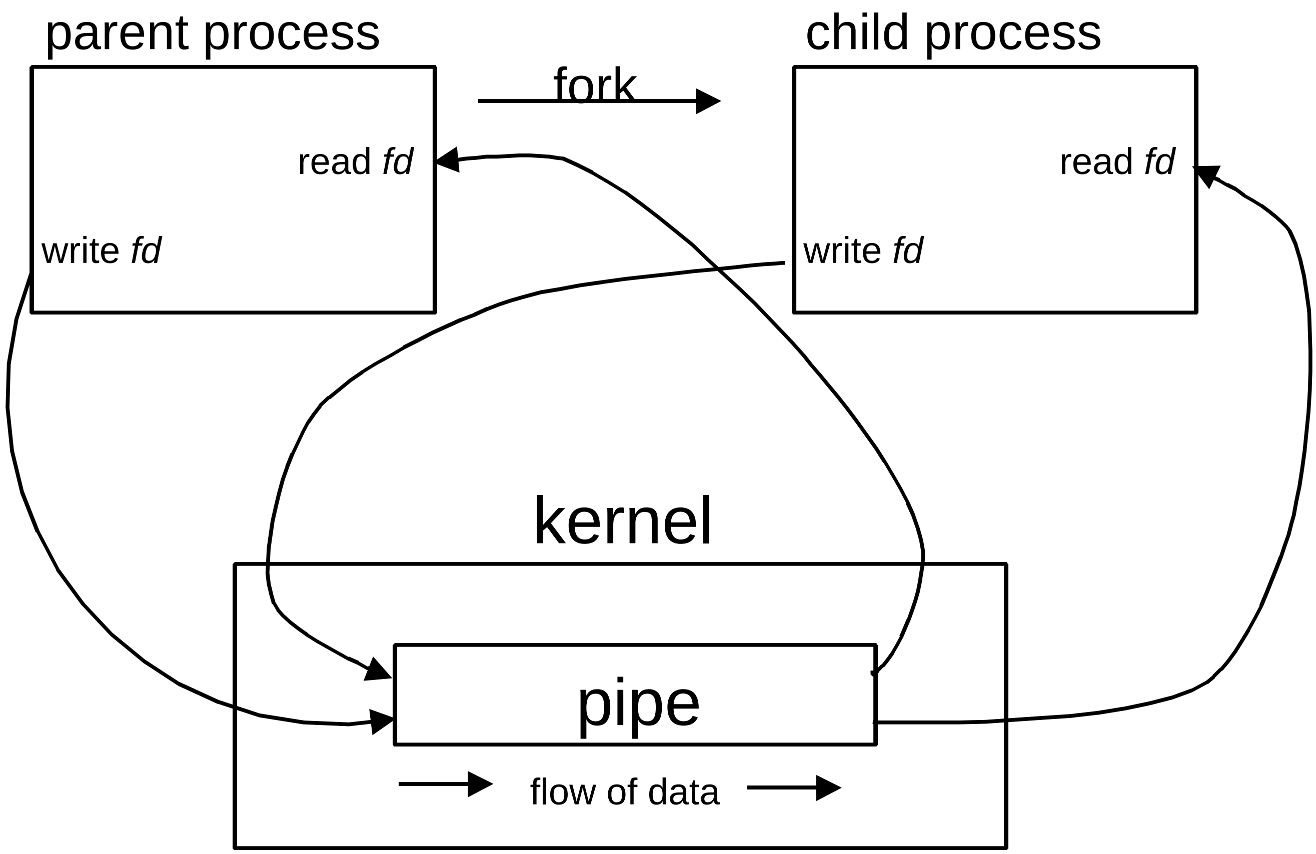
  filedes[1] open for writing



Using a Pipe for IPC

Parent process forks after creating a pipe

- Open file descriptors copied in child
- Parent, child share both ends of pipe



Note: As an example, the parent process could be the producer and the child process could be the consumer.

The producer should write to the pipe.

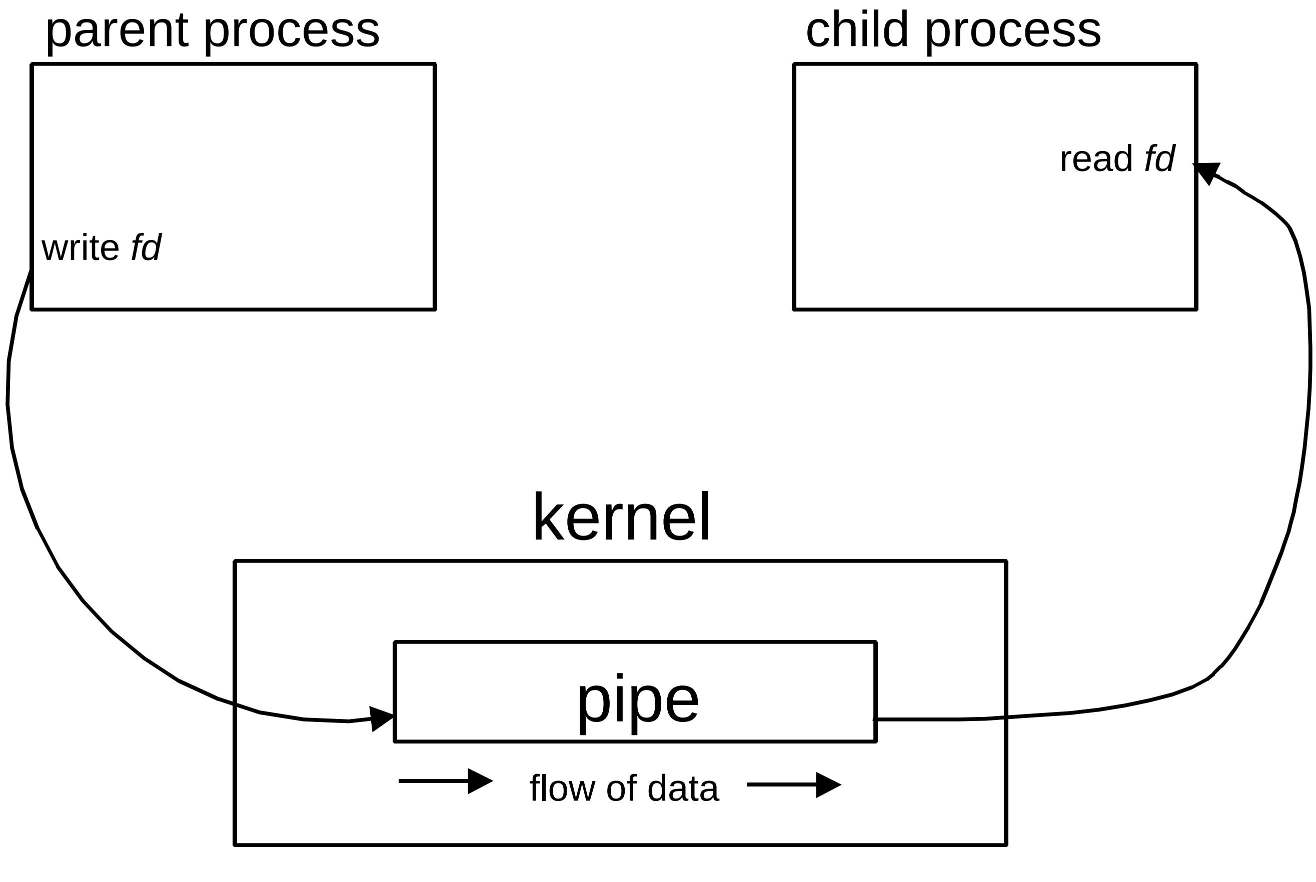
The consumer should read from the pipe.

<u>Note:</u> Pipes are only usable between processes that share an ancestor.

Pipe Between Two Processes

After the *fork* that creates the shared pipe

- Writer closes the read end of pipe
- Reader closes the write end of pipe



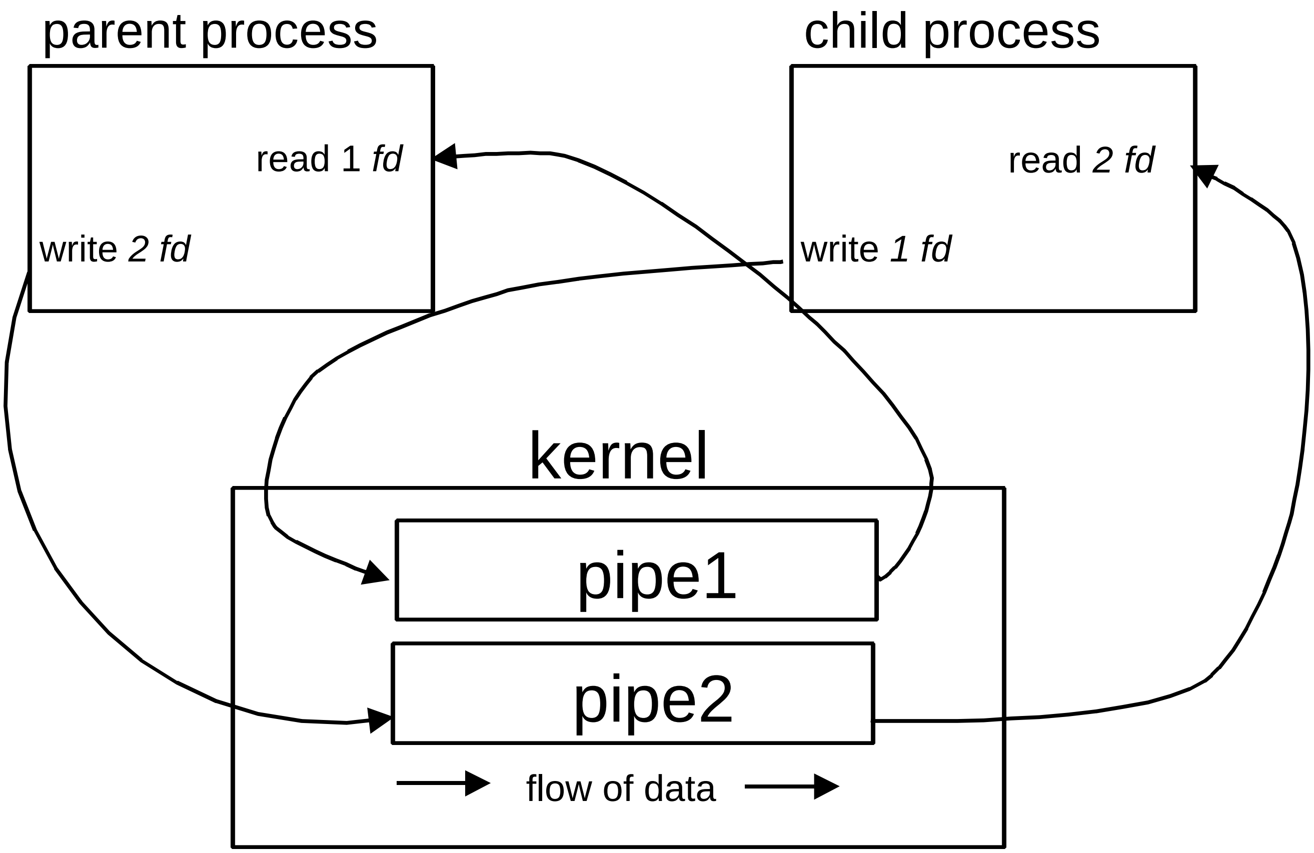
For 2-way communication

- Open 2 pipes before forking.
- Use 1 pipe as above.
- Reverse roles for 2nd pipe.

Rules for Reading and Writing Pipes

- A *read* returns the lesser of the requested amount of data or contents of pipe.
- A *read* waits until there is data in the pipe if pipe is open for writing; else, it returns 0 for EOF.
- A *write* is atomic if data is less than the capacity of the pipe; else it is equivalent to many atomic *write* operations.
- A *write* waits until space is available if the pipe is full and open for reading; else the SIGPIPE signal is generated and 0 is returned. (If process does not catch this signal it is terminated.)
- The data is an uninterpreted stream of bytes.

Client-Server Example



```
main()
{       int child, pipe1[2], pipe2[2];
        if (pipe(pipe1) < 0 || pipe(pipe2) < 0)   err_sys("can't create pipes");
        if (( child = fork()) < 0)      err_sys("can't fork");
        else if (child > 0) {  /* PARENT CODE */
                close(pipe1[1]);      close(pipe2[0]);
                client(pipe1[0], pipe2[1]);
                while (wait((int *) 0) != child) ;    /* WAIT FOR THIS CHILD */
                close(pipe1[0]);      close(pipe2[1]);
                exit(0);
                }
        else {                 /* CHILD CODE */
                close(pipe1[0]);      close(pipe2[1]);
                server(pipe2[0], pipe1[1]);
                close(pipe2[0]);      close(pipe1[1]);
                exit(0);
                }
}
```

Client Implementation

The client writes filename to the server.

- ★Read filename from *stdin*.
- ★Write to IPC channel.

Read data from server.

- ★Read IPC channel.
- ★Write to *stdout*.

```
#include <stdio.h>
#define MAXBUFF 1024
client(readfd, writefd)
int readfd; int writefd;
{
        char buff[MAXBUFF];
        int n;
        if (fgets(buff, MAXBUFF, stdin) == NULL)
                err_sys("client: filename read error");
        n = strlen(buff);
        if (buff[n-1] == '\n') n--;
        if (write(writefd, buff, n) != n)
                err_sys("client: filename write error");
        while ((n = read(readfd, buff, MAXBUFF)) > 0)
                if (write(STDOUT, buff, n) != n)
                          err_sys("client: data write error");
        if (n < 0) err_sys("client: data read error");
}
```

Server Implementation

- Read filename from IPC channel.
- If *open* fails write error on IPC.
- Else write data on IPC channel.

```
#include <stdio.h>
#define MAXBUFF 1024
server(readfd, writefd)
int readfd; int writefd;
{
        char buff[MAXBUFF], errmsg[256], *sys_err_str();
        int n, fd;
        extern int errno;
        if ((n = read(readfd, buff, MAXBUFF)) <= 0)
                err_sys("server: filename read error");
        buff[n] = '\0';
        if ( (fd = open(buff, 0)  ) < 0) {
                sprintf(errmsg, ": can't open, %s\n", sys_errr_str());
                strcat(buff, errmsg);
                n = strlen(buff);
                if (write(writefd, buff, n) != n)
                 err_sys("server: errmsg write error");
        } else {
                while ((n = read(fd, buff, MAXBUFF)) > 0)
                if (write(writefd, buff, n) != n)
                 err_sys("server: data write error");
                if (n < 0) err_sys("server: data read error");
                }
}
```

Message Queue IPC (System V)

- System calls for message queue IPC.
  - *msgget* System call to create or open an already created message queue.
  - *msgctl* System call for control operations on message queue.
  - *msgsnd* System call for sending a message to message queue.
  - *msgrcv* System call to receive a message from message queue.
    - IPC key Used to identify the message queue. Created by calling `key_t ftok(char *pathname, char proj);` with pathname and char agreed upon by client and server.
  - *<sys/msg.h>* Include file for message queues.

Client-Server Example using Message Queue IPC

Message Data Structure (mesg.h)

```
typedef struct {
        int mesg_len;
        long mesg_type;
        char mesg_data[MAXMESGDATA];
} Mesg;
```

Client Process

```
Mesg mesg;
client(ipcreadfd, ipcwritefd)
int ipcreadfd; int ipcwritefd;
{       int n;
        if (fgets(mesg.mesg_data, MAXMSGDATA, stdin) == NULL)
                err_sys("Client: filename read error");
        n = strlen(mesg.mesg_data);
        if (mesg.mesg_data[n-1] == '\n')  n--;
        mesg.mesg_len = n;
        mesg.mesg_type = 1L;
        mesg_send(ipcwritefd, &mesg);

        while ( (n = mesg_recv(ipcreadfd, &mesg)) > 0)
                if (write(1, mesg.mesg_data, n) != n)
                 err_sys("Client: Data write error");
        if (n < 0) err_sys("Client: Data read error");
}
```

Client-Server Example (continued)

Server Process

```
Mesg mesg;
server(ipcreadfd, ipcwritefd)
int ipcreadfd; int ipcwritefd;
{       int n, fileid;
        char errmesg[256], *sys_err_str();

        mesg.mesg_type = 1L;
        if ( (n = mesg_recv(ipcreadfd, &mesg)) <= 0)
                err_sys("Server: Filename read error");
        mesg.mesg_data[n] = '\0';
        if ( (fileid = open(mesg.mesg_data, 0)) < 0) {
                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(mesg.mesg_data, errmesg);
                mesg.mesg_len = strlen(mesg.mesg_data);
                mesg_send(ipcwritefd, &mesg);
        }
        else {
                while ( (n = read(fileid, mesg.mesg_data, MAXMESGDATA)) > 0)
                {mesg.mesg_len = n;
                 mesg_send(ipcwritefd, &mesg);
                }
                close(fileid);
                if (n < 0) err_sys("Server: Data read error");
        }
        /* Empty message to denote end */
        mesg.mesg_len = 0;
        mesg_send(ipcwritefd, &mesg);
}
```

Client-Server main Functions

```
#include "msgq.h"

main()
{
        int readfd, writefd;
        if ((readfd = msgget(MKEY1, PERMS | IPC_CREAT) ) < 0)
                err_sys("Server: Can't get message queue 1");
        if ((writefd = msgget(MKEY2, PERMS | IPC_CREAT) ) < 0)
                err_sys("Server: Can't get message queue 2");

        server(readfd, writefd);

        exit(0);
}
```

msgq.h

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

#include <sys/errno.h>
extern int errno;

#define MKEY1 1234L
#define MKEY2 2345L

#define PERMS 0666
```

```
#include "msgq.h"

main()
{
        int readfd, writefd;
        if ((writefd = msgget(MKEY1, PERMS | IPC_CREAT) ) < 0)
                err_sys("Client: Can't get message queue 1");
        if ((readfd = msgget(MKEY2, PERMS | IPC_CREAT) ) < 0)
                err_sys("Client: Can't get message queue 2");
        client(readfd, writefd);
        if (msgctl(readid, IPC_RMID, (struct msqid_ds *) 0 )) < 0)
                err_sys("Client: Can't RMID message queue 2");
        if (msgctl(writeid, IPC_RMID, (struct msqid_ds *) 0 )) < 0)
        err_sys("Client: Can't RMID message queue 1");
        exit(0);
}
```

Data Abstraction mesg_send & mesg_recv

```
#include "mesg..h"

mesg_send(msgQid, mesgptr)
int msgQid;
Mesg *mesgptr;
{
        /* Use msgsnd system call to send message */

        if (msgsnd(msgQid, (char *) &(mesgptr->mesg_type),
                 mesgptr->mesg_len, 0) != 0)
                err_sys("msgsnd error");
}
```

```
#include "mesg.h"

int mesg_recv(msgQid, mesgptr)
int msgQid;
Mesg *mesgptr;
{
        int n;

        /* Read the first message of specified message type */

        n = msgrcv(msgQid, (char *) &(mesgptr->mesg_type),
                 MAXMESGDATA, mesgptr->mesg_type, 0);
        if ( (mesgptr->mesg_len = n) < 0)
                err_dump("msgrcv error");
        return (n);          /* n = 0 signifies EOF */
}
```