Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

Outline

- Pipe
- FIFO
- System V Message Queues

innovate achieve lead

BITS Pilani
Pilani Campus

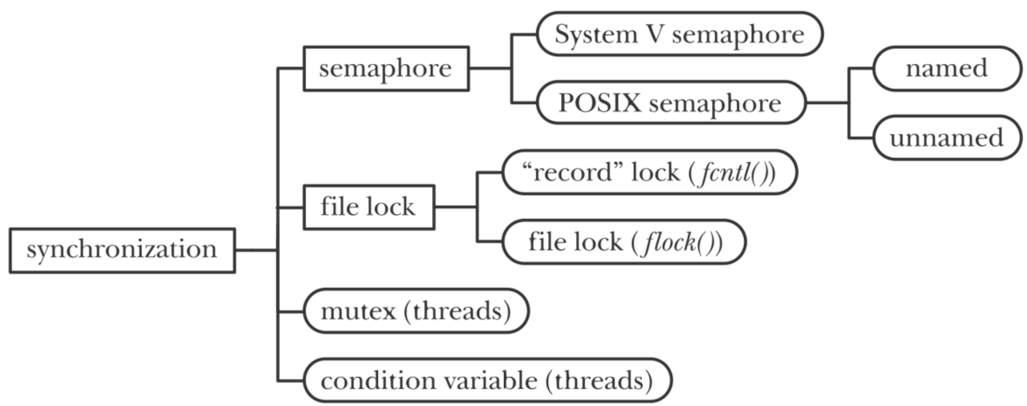
Overview of Inter Process Communication (IPC) (R1: Ch43)

What is IPC?

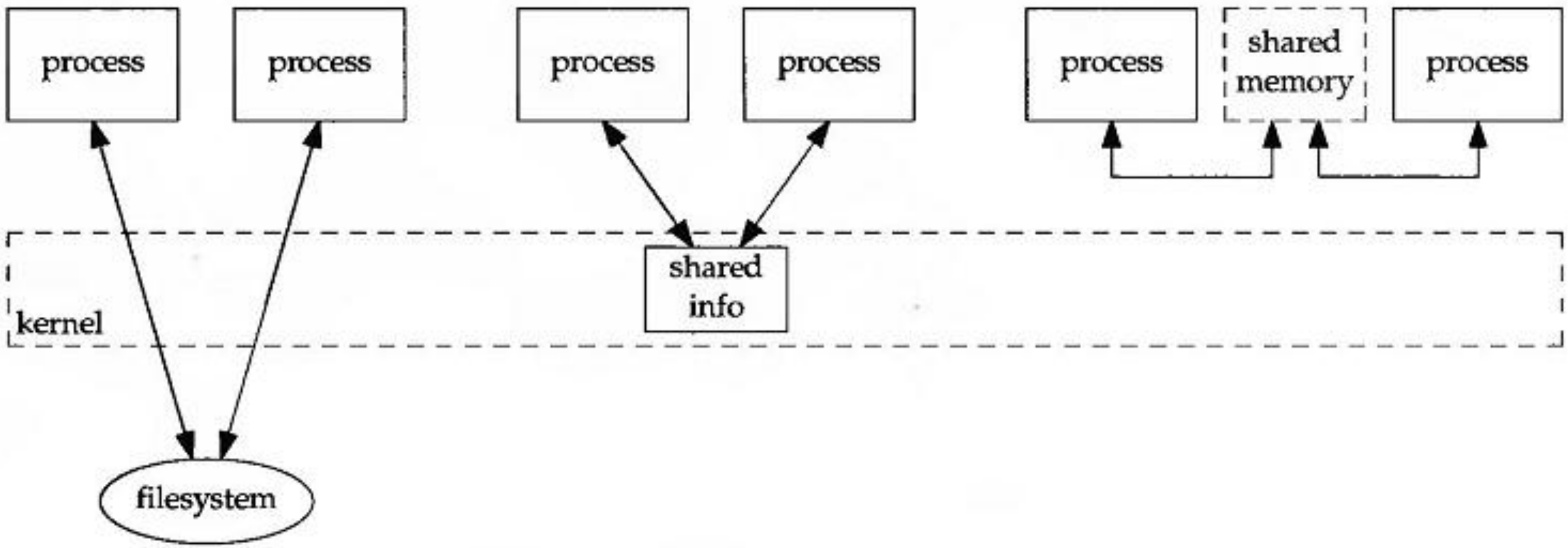
- Each process has a private address space. A process can't access the memory of another.
- Data Transfer
  - Facilities
    - Byte stream: Pipe, FIFO, stream socket
    - Message: System V message Queues, POSIX message queues, datagram sockets
  - Destructive reads.
  - Synchronization between reader and writer is automatic.
- Shared Memory
  - Facilities:
    - System V Shared memory, POSIX shared memory, memory mappings.
  - Faster, but needs synchronization.
  - Data placed on shared memory is available unless deleted.

What is IPC?

- Synchronization Facilities
  - Semaphores
    - Kernel maintained integer. Access to shared resource is controlled by incrementing or decrementing semaphore.
    - System V Semaphore
    - POSIX semaphore
  - File locks
    - *flock():* locks entire file.
    - *fcntl():* locks a region of a file.
  - Mutexes and condition variables
    - for pthreads



Sharing of Information



BITS Pilani
Pilani Campus

Pipe(R1: Ch44)

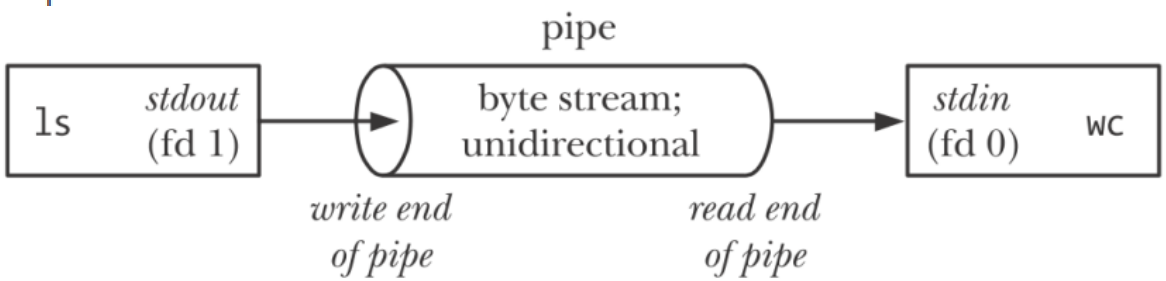
Pipe

- Pipes are the oldest form of UNIX System IPC and are provided by all UNIX systems
  - Yet most commonly used form of IPC
- Historically, they have been half duplex (i.e., data flows in only one direction).
- Because they don’t have names, 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

- We can think of pipe as piece of plumbing that allows data to flow from one process to another.

```
2  $ ls | wc -l
```

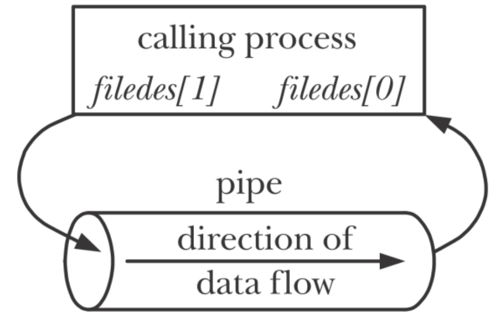


- A pipe is byte stream. No boundaries maintained between two writes of sender process.
- Pipe are unidirectional.
  - Data can travel in only one direction.
  - One end is used for reading and the other for writing.
- Pipe is simply a buffer maintained in kernel memory.
  - Its size PIPE_BUF is normally 4096 bytes.

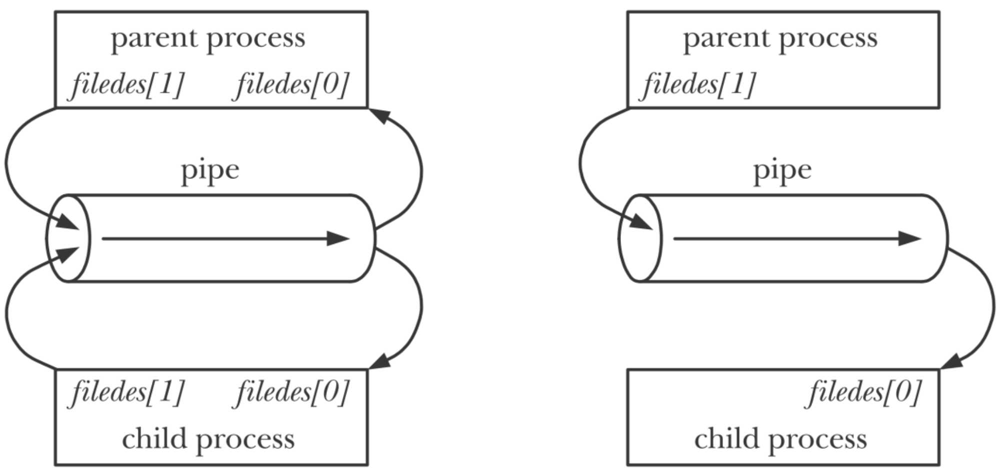
Creating and Using Pipes

- The pipe() system call creates a new pipe.

```c
#include <unistd.h>
int pipe(int  filedes [2]);
/*Returns 0 on success, or -1 on error*/
```



  - Successful call return two file descriptors.
    - Filedes[0] for read end and filedes[1] for write end.
- Normally pipe is used for communication between two processes. So *fork()* follows *pipe()* system call.



a) After *fork()*   **b) After closing unused descriptors**

Creating and Using Pipes

```c
int filedes[2];
if (pipe(filedes) == -1)          /* Create the pipe */
    errExit("pipe");
switch (fork()) {                /* Create a child process */
case -1:
    errExit("fork");
case 0:  /* Child */
    if (close(filedes[1]) == -1) /* Close unused write end */
        errExit("close");
    /* Child now reads from pipe */
    break;
default: /* Parent */
    if (close(filedes[0]) == -1)  /* Close unused read end */
        errExit("close");
    /* Parent now writes to pipe */
    break;
}
```

- Child and parent close the unused ends.
- Fir bi-directional transfer of data, use two pipes.
  - Using single pipe would lead to race conditions.

example

- Child writing to the parent using a pipe

```
main ()
{
  int i;
  int p[2];
  pid_t ret;
  pipe (p);          //creating pipe
  char buf[100];
  ret = fork ();
  if (ret == 0)
    {
     write (p[1], "hello", 6);//writing to parent through pipe
    }
  if (ret > 0)
    {
      read (p[0], buf, 6); //reading from child via pipe
      printf ("Child Said:%s\n", buf); //printing to stdout
    }
}
```

Bidirectional Transfer of Data

- If there is need for both parent and child to read and write data, then
  - Using single pipe leads to race conditions. Can be avoided using some synchronizations mechanism.
  - Simpler is to use to two pipes, one in each direction.
    - This may lead to deadlock situation. Both parent and child blocked in reading but there is no data in the pipes.
- Not only parent and child but any two processes having a common ancestor can use pipe provided that common ancestor has created the pipe.
  - p creates pipe.
  - c1 c2
    - c3 c4
  - c3 and c1/c4 can communicate using a pipe.

Closing Unused Pipe Descriptors

- Process reading from pipe closes write end of the pipe. Why?
  - While reading from pipe an EOF is encountered only if there are no more write ends open.
  - If not closed, the read may block indefinitely waiting.
- Process writing to pipe closes read end of the pipe. Why?
  - If a process tries to write to a pipe for which there is no read end open, then kernel generates SIGPIPE signal. This signal has default action, terminate process.
    - Although write() also returns an error EPIPE, but the generation of signal is meant for killing the sender process because the processes using pipe do not know that they are using pipe.
  - If the process doesn't close read end, process will still be able to write to the pipe, once full it will indefinitely blocked waiting for someone to read the pipe.
    - No other process has read end open.

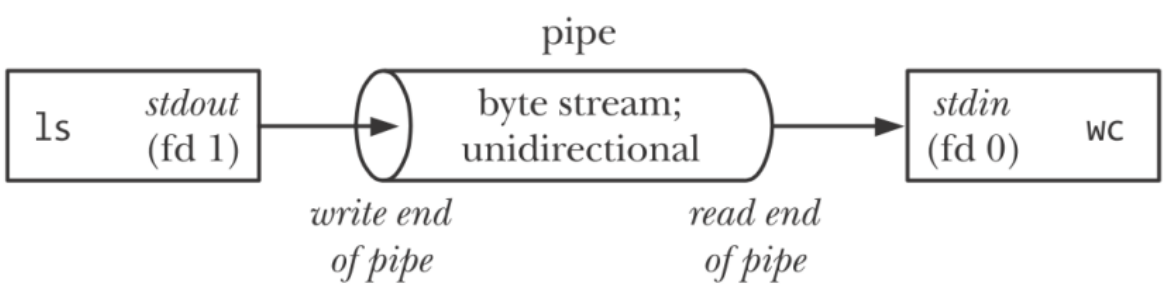
Pipe as a Method of Process Synchronization

```c
main ()
{
  int i;
  int p[2];
  pid_t ret;
  pipe (p);         //creating pipe
  char buf[1];
  ret = fork ();
  if (ret == 0)
    {/*do some work*/
      write (p[1], "1", 1);/*do work and tell parent*/
    }
  if (ret > 0)
    {
      read (p[0], buf, 1); /*wait for the child */
      /*continue ahead*/
    }
}
```

- Synchronizing with pipes has advantage over signals:
  - Multiple signals can’t be queued. Multiple processes can block on pipe.
- Signals have advantage that broad can be done in a process group.

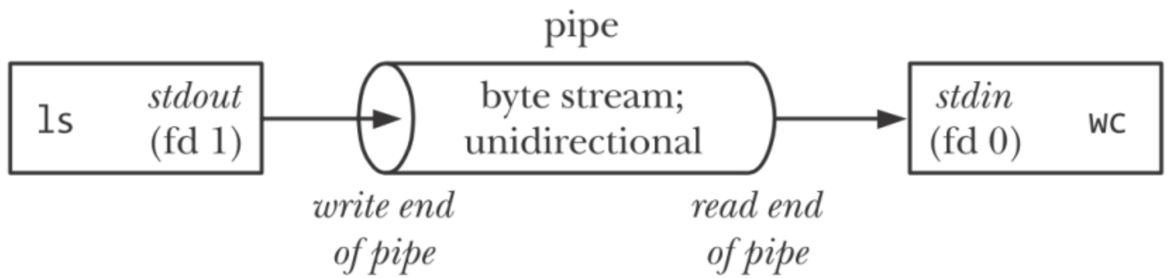
Using Pipes to Connect Filters

- *ls | wc*
  - Create a pipe in the parent
  - Fork a child
  - Duplicate the standard output descriptor to write end of pipe
  - Exec '*ls*' program
  - In the parent wait for the child.
  - Duplicate the standard input descriptor to read end of pipe
  - Exec '*wc*' program



ls | wc

```
main ()
{ int i;
  int p[2];
  pid_t ret;
  pipe (p);
  ret = fork ();
  if (ret == 0)
    {
      close (1);
      dup (p[1]);
      close (p[0]);
      execlp ("ls", "ls", (char *) 0);
    }
  if (ret > 0)
    {
      close (0);
      dup (p[0]);
      close (p[1]);
      wait (NULL);
      execlp ("wc", "wc", (char *) 0);
    }}
```



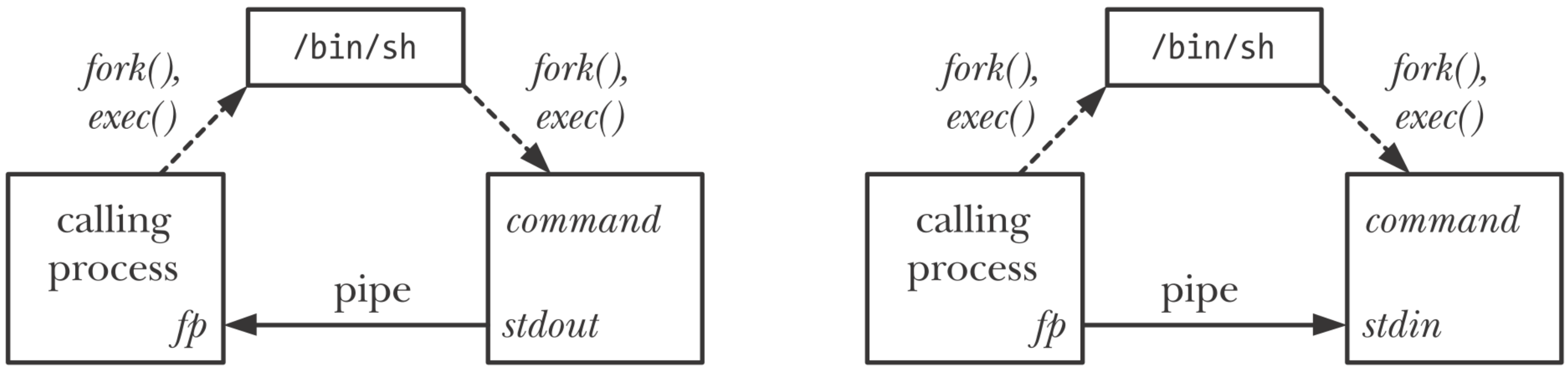
Talking to a Shell Command via a Pipe: *popen()*

- A common use for pipes is to execute a shell command and either read its output or send it some input.
  - The *popen*() and *pclose*() functions are provided to simplify this task.
  - The popen() function
    - creates a pipe
    - forks a child process that execs a shell
    - shell creates a child process to execute the string given in command.

```
#include <stdio.h>
FILE *popen(const char * command , const char * mode );
/*Returns file stream, or  NULL  on error*/
int pclose(FILE * stream );
/*Returns termination status of child process, or -1 on error*/
```

- The mode argument is a string that determines whether the calling process will read from the pipe (mode is r) or write to it (mode is w).

Talking to a Shell Command via a Pipe: *popen()*



a) ***mode*** **is** ***r*** **b)** ***mode*** **is** ***w***

```
fp = popen(popenCmd, "r");
if (fp == NULL) {
    printf("popen() failed\n");
    continue;
}
/* Read resulting list of pathnames until EOF */
fileCnt = 0;
while (fgets(pathname, PATH_MAX, fp) != NULL) {
    printf("%s", pathname);
    fileCnt++;
}
```

BITS Pilani
Pilani Campus

FIFO (R1: Ch44)

FIFO (named pipes)

- A FIFO is similar to a pipe. Difference is that a FIFO has a name within the file system and is opened in the same way as a regular file.
- This allows FIFO to be used for communication between unrelated processes.
  - E.g. client-server
- Just as with pipe, FIFO also has read and write end.
  - Read end is opened using *open()* with O_RDONLY mode.
  - Write end is opened using *open()* with O_WRONLY mode.
- FIFO stands for *first in, first out*.
- It is unidirectional (half-duplex) just like pipe.

Create FIFO

- FIFO is created using *mkfifo*() function.

```c
#include <sys/stat.h>
int mkfifo(const char * pathname , mode_t  mode );
/*Returns 0 on success, or -1 on error*/
```

  - pathname refers to a file path. File is created with FIFO file type.
  - mode refers to permissions.
  - mkfifo returns error ‘EEXIST’ if the FIFO already exists at the given path

- One can use shell command also.

```
$ mkfifo [ -m mode ] pathname
$ mkfifo myfifo
$ wc -l < myfifo &
$ ls -l > myfifo
```

Opening FIFO

- Once a FIFO is created, it should be opened either for reading or writing
  - wfd=open("fifo1",O_WRONLY); or
  - FILE *fp = fopen(“fifo1”, “w”);
- FIFO can’t be opened both for reading and writing at the same time
- Unlike pipe, FIFO is not deleted as soon as all the processes referring to it exit. It has to be explicitly deleted from system.
  - unlink(“fifo1”)

Opening FIFO

- Generally, the only sensible use of a FIFO is to have a reading process and a writing process on each end.
    - Therefore, by default, opening a FIFO for reading (the open() O_RDONLY flag) blocks until another process opens the FIFO for writing (the open() O_WRONLY flag).
- Conversely, opening the FIFO for writing blocks until another process opens the FIFO for reading.
- In other words, opening a FIFO synchronizes the reading and writing processes.

FIFOs between parent and child


```
if ( (childpid = Fork()) == 0) {        /* child */
        readfd = Open(FIFO1, O_RDONLY, 0);
        writefd = Open(FIFO2, O_WRONLY, 0);

        server(readfd, writefd);
        exit(0);
}
        /* parent */
writefd = Open(FIFO1, O_WRONLY, 0);
readfd = Open(FIFO2, O_RDONLY, 0);
```

- If the order of the last two lines is changed there will be deadlock.

Nonblocking I/O

- Using O_NONBLOCK flag when opening a FIFO serves two purposes
  - It allows single process to open both ends of a FIFO.
    - Open read end by specifying)_NONBLOCK
    - Then open write end normally
  - It prevents deadlocks between process opening two FIFOs.

```
fd = open("fifopath", O_RDONLY | O_NONBLOCK);
if (fd == -1)
    errExit("open");
```

Type of *open()*		Result of *open()*	
open for	**additional flags**	**other end of FIFO open**	**other end of FIFO closed**
reading	none (blocking)	succeeds immediately	blocks
	O_NONBLOCK	succeeds immediately	succeeds immediately
writing	none (blocking)	succeeds immediately	blocks
	O_NONBLOCK	succeeds immediately	fails (ENXIO)

  - Attempt to open FIFO for writing when no reader is open will result in error.

Semantics of read() and write()

Table 44-2: Semantics of reading n bytes from a pipe or FIFO containing p bytes

O_NONBLOCK enabled?	Data bytes available in pipe or FIFO (p)			
	$p = 0$, write end open	$p = 0$, write end closed	$p < n$	$p >= n$
No	block	return 0 (EOF)	read p bytes	read n bytes
Yes	fail (EAGAIN)	return 0 (EOF)	read p bytes	read n bytes

Table 44-3: Semantics of writing n bytes to a pipe or FIFO

O_NONBLOCK enabled?	Read end open		Read end closed
	n <= *PIPE_BUF*	n > *PIPE_BUF*	
No	Atomically write n bytes; may block until sufficient data is read for *write()* to be performed	Write n bytes; may block until sufficient data read for *write()* to complete; data may be interleaved with writes by other processes	SIGPIPE + EPIPE
Yes	If sufficient space is available to immediately write n bytes, then *write()* succeeds atomically; otherwise, it fails (EAGAIN)	If there is sufficient space to immediately write some bytes, then write between 1 and n bytes (which may be interleaved with data written by other processes); otherwise, *write()* fails (EAGAIN)	

innovate achieve lead

BITS Pilani
Pilani Campus

Message Queues

Message Queues

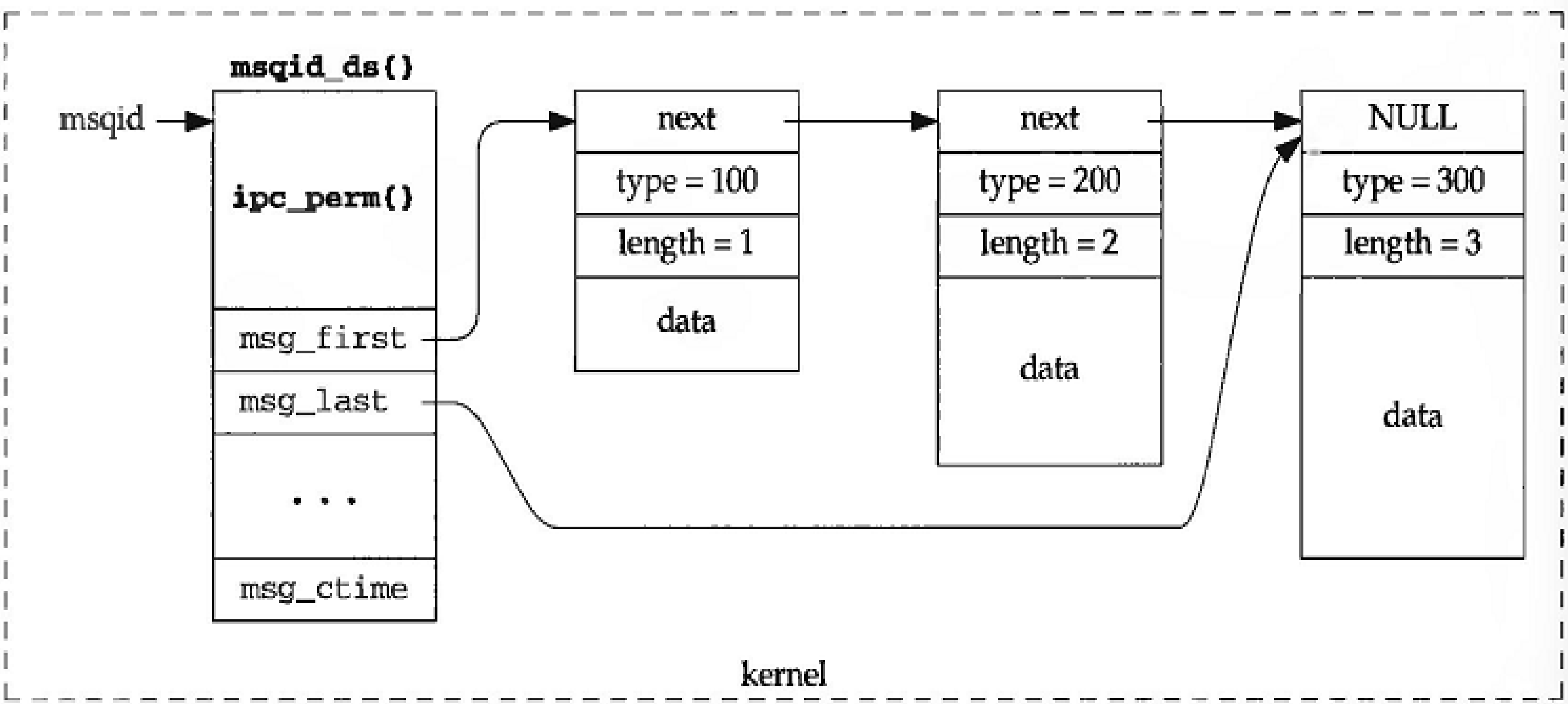
- A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier
    - Any process with adequate privileges can place the message into the queue and any process with adequate privileges can read from queue
- There is no requirement that some process must be waiting to receive message before sending the message
- Maintains boundaries between messages.
- Even if all processes referencing message queue terminate, still the message queue exists.

Message Queues

- Every message queue has following structure in kernel

```c
struct msqid_ds {
    struct ipc_perm msg_perm;          /* Ownership and permissions */
    time_t          msg_stime;         /* Time of last msgsnd() */
    time_t          msg_rtime;         /* Time of last msgrcv() */
    time_t          msg_ctime;         /* Time of last change */
    unsigned long   __msg_cbytes;      /* Number of bytes in queue */
    msgqnum_t       msg_qnum;          /* Number of messages in queue */
    msglen_t        msg_qbytes;        /* Maximum bytes in queue */
    pid_t           msg_lspid;         /* PID of last msgsnd() */
    pid_t           msg_lrpid;         /* PID of last msgrcv() */
};
```

Message Queues



Message Queues

- First **`msgget`** is used to either open an existing queue or create a new queue

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

- Key value can be IPC_PRIVATE, key generated by ftok() or any key (long integer)
- Flag value must be
  - IPC_CREAT if a new queue has to be created
  - IPC_CREAT and IPC_EXCL if want to create a new a queue but don’t reference existing one .

Key values

- The server can create a new IPC structure by specifying a key of IPC_PRIVATE
    - Kernel generates a unique id
- The client and the server can agree on a key by defining the key in a common header.
- The client and the server can agree on a pathname and project ID and call the function *ftok* to convert these two values into a key.

```c
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

- The path argument must refer to an existing file. Only the lower 8 bits of id are used when generating the key.

Message Queues

- When a new queue is created, the following members of the msqid_ds structure are initialized.
- The ipc_perm structure is initialized
  - 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.

Messages

- Most applications define their own message structure according to the needs of the application.

```
struct mymsg {
    long mtype;                 /* Message type */
    char mtext[];               /* Message body */
}
```

```
#define MY_DATA  8

typedef struct my_msgbuf {
   long      mtype;     /* message type */
   int16_t   mshort;    /* start of message data */
   char      mchar[MY_DATA];
} Message;
```

Sending Messages

```c
#include <sys/types.h>          /* For portability */
#include <sys/msg.h>
int msgsnd(int  msqid , const void * msgp , size_t  msgsz , int  msgflg );
//Returns 0 on success, or -1 on error
```

- *msqid* is the id returned by msgget sys call
- The *msgp* argument is a pointer to a message structure
- *msgsz* is the length of the message without mtype field.
- A flag value of 0 or IPC_NOWAIT can be specified.
- mssnd() is blocked until one of the following occurs
    - Room exists for the message
    - Message queue is removed (EIDRM error is returned)
- I nterrupted by a signal (EINTR is returned)

Receiving Messages

```
#include <sys/types.h>          /* For portability */
#include <sys/msg.h>
ssize_t msgrcv(int  msqid , void * msgp , size_t  maxmsgsz , long  msgtyp , int  msgflg );
//Returns number of bytes copied into mtext field, or -1 on error
```

- msgp points to the message structure where message will be stored.
- maxmsgsz points to the size available on the message structure excluding size of (long)
- msgtype indicates the message desired on the message queue
- Flag can be 0 or IPC_NOWAIT or MSG_NOERROR

Receiving Messages

- The type argument 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, treat the waiting messages as a priority queue. The first message of the lowest mtype less than or equal to the absolute value of msgtyp is removed and returned to the calling process.
- A nonzero type is used to read the messages in an order other than first in, first out.
  - Priority to messages, Multiplexing

Receiving Messages

- IPC_NOWAIT flag makes the operation nonblocking, causing msgrcv to return -1 with errno set to ENOMSG if a message of the specified type is not available.
- If IPC_NOWAIT is not specified, the operation blocks until
  - a message of the specified type is available,
  - the queue is removed from the system (-1 is returned with errno set to EIDRM)
  - a signal is caught and the signal handler returns (causing msgrcv to return 1 with errno set to EINTR).

Receiving Messages

- If the returned message is larger than nbytes and the MSG_NOERROR bit in flag is set, the message is truncated.
  - no notification is given to us that the message was truncated, and the remainder of the message is discarded.
- If the message is too big and MSG_NOERROR is not specified, an error of E2BIG is returned instead (and the message stays on the queue).

Control Operations on Message Queues

```c
#include <sys/types.h>          /* For portability */
#include <sys/msg.h>
int msgctl(int  msqid , int  cmd , struct msqid_ds * buf );
//Returns 0 on success, or -1 on error
```

- IPC_STAT: Fetch the msqid_ds structure for this queue, storing it in the structure pointed to by buf.
- IPC_SET: Copy the following fields from the structure pointed to by buf to the msqid_ds structure associated with this queue: msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes.
- 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.
  - Above two commands can be executed only by a process whose effective user ID equals msg_perm.cuid or msg_perm.uid or by a process with superuser privileges

Server.c

```
/*key.h*/
#define MSGQ_PATH
"/home/students/f2007045/msgq_server.c "

struct my_msgbuf
{
  long mtype;
  char mtext[200];
};

int main (void)
{
  struct my_msgbuf buf;
  int msqid;
  key_t key;
   if ((key = ftok (MSGQ_PATH, 'B')) == -1)
    {
      perror ("ftok");
      exit (1);
    }
```

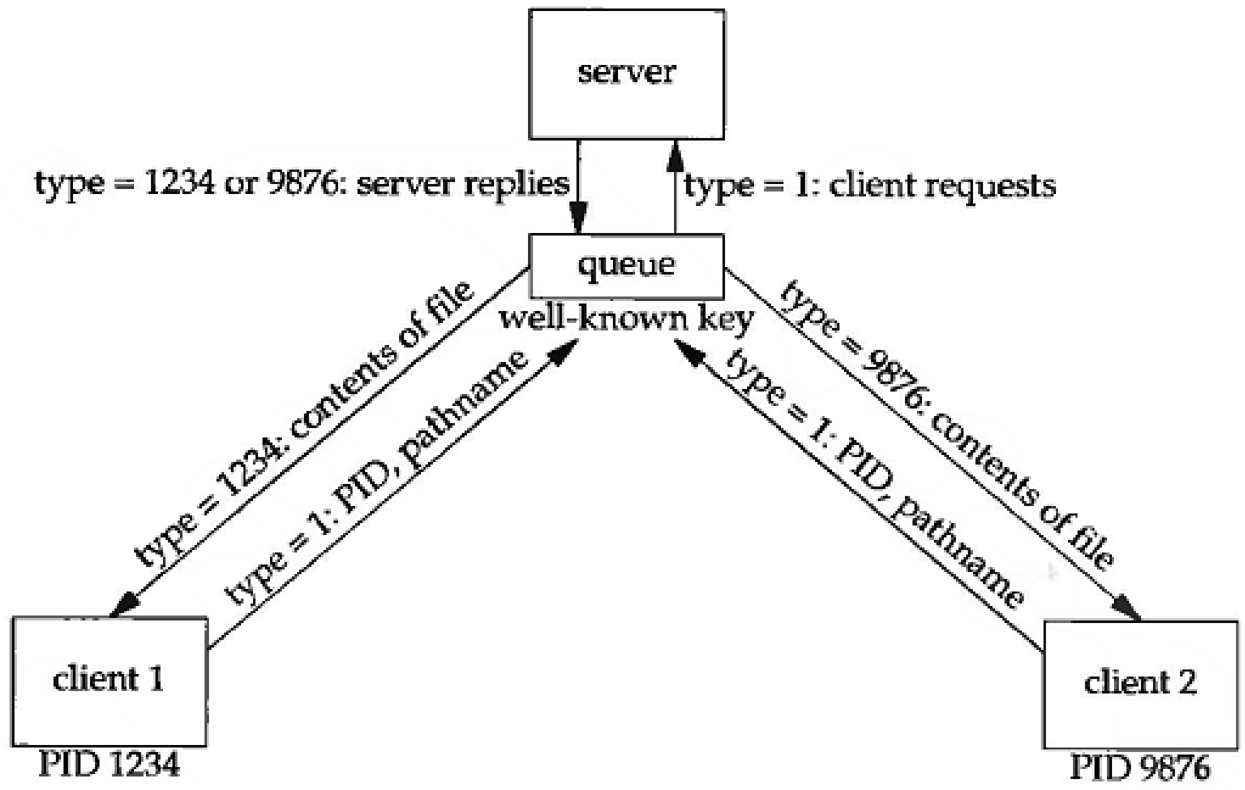
```
  if ((msqid = msgget (key, IPC_CREAT | 0644)) == -1)
    {
      perror ("msgget");
      exit (1);
    }
   printf ("server: ready to receive messages\n");
 for (;;)
    {
      if (msgrcv (msqid, &(buf.mtype), sizeof (buf), 0, 0)
== -1)
                  {
                    perror ("msgrcv");
                    exit (1);
                  }
      printf ("server: \"%s\"\n", buf.mtext);
    }
   return 0;
}
```

Client.c

```
#include "key.h“
struct my_msgbuf
{
  long mtype;
  char mtext[200];
};

main (void)
{
  struct my_msgbuf buf;
  int msqid;
  key_t key;
   if ((key = ftok (MSGQ_PATH, 'B')) == -1)
    {
      perror ("ftok");
      exit (1);
    }
   if ((msqid = msgget (key, 0) == -1)
    {
      perror ("msgget");
      exit (1);
    }
```

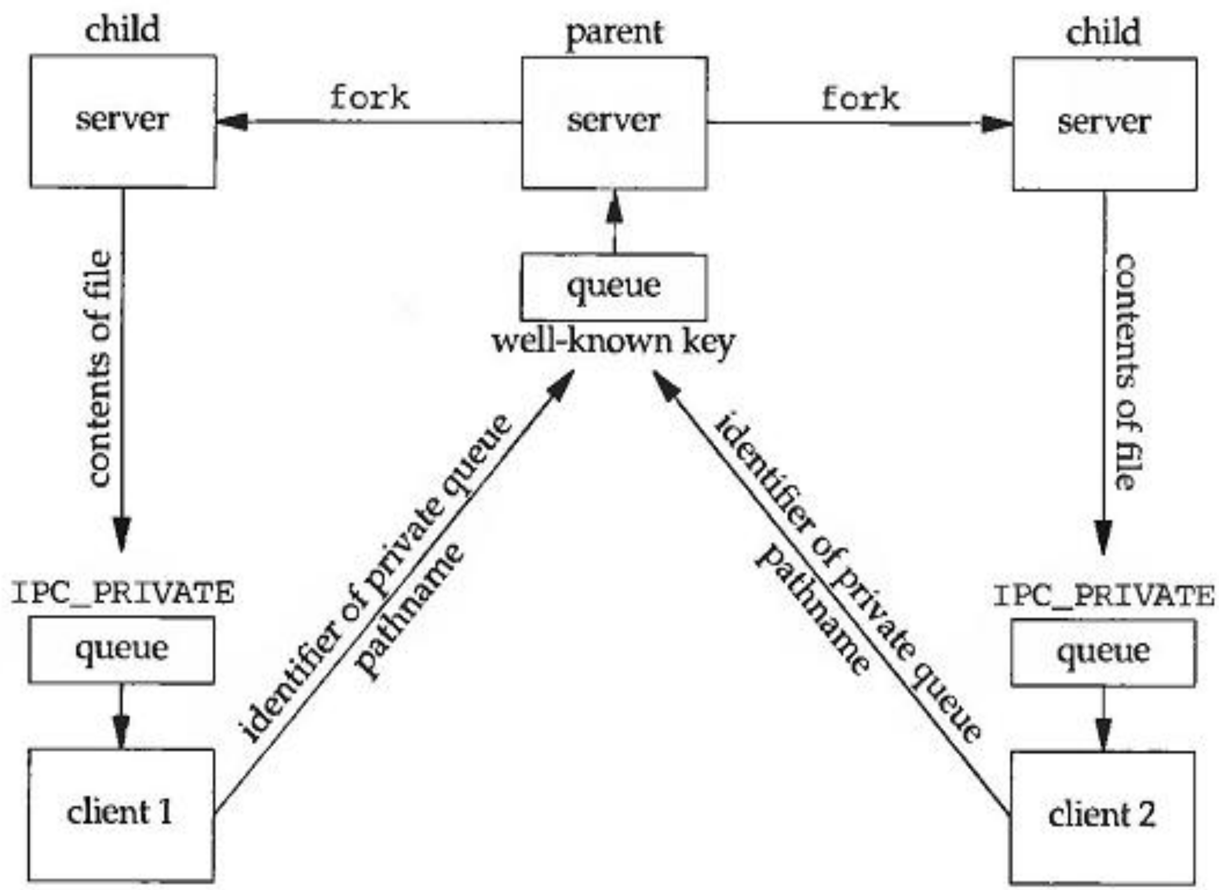
```
 printf ("Enter lines of text, ^D to quit:\n");
   buf.mtype = 1;

 while (gets (buf.mtext), !feof (stdin))
    {
      if (msgsnd (msqid, &(buf.mtype), sizeof (buf), 0) == -
1)
perror ("msgsnd");
    }

  if (msgctl (msqid, IPC_RMID, NULL) == -1)
    {
      perror ("msgctl");
      exit (1);
    }

  return 0;
}
```

Multiplexing Messages



- Possibility of dead lock

Multiplexing Messages



Q&A

Next Time

- Please read through R1: chapters 47-48

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You