CH15: IPC Roundup: Can We talk?

Objectives

Ideas and Skills

- Named pipes
- Shared memory
- File locks
- Semaphores
- IPC overview

System Calls, Functions, Commands

- mkfifo
- shmget, shmat, shmctl, shmdt
- semget, semctl
- lpr

15.3: Communication Choices

One Problem, Three Solutions

The problem

- Consider a situation where you have two unrelated processes.
- Let's assume they have a client, server relationship.
- Let's further assume both processes run on the same machine, for the sake of convenience.

The problem

- ***The Problem: Getting Data from Server to Client***.
- What methods do we have for getting data from one unrelated process to another?

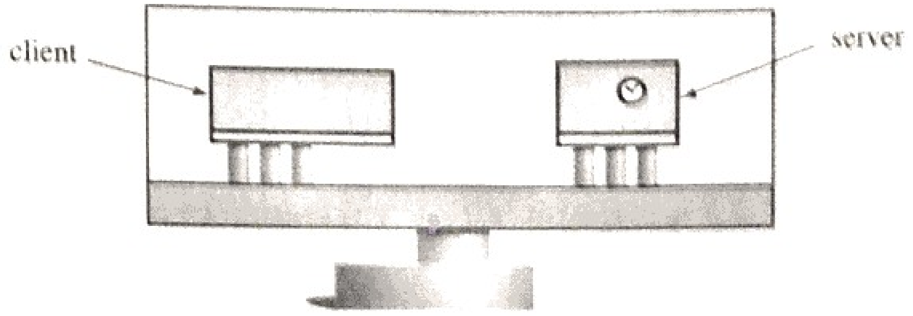


FIGURE 15.4

One process has information the other one wants.

Three solutions

- We have three methods to choose from, one we know (use files), and two new methods: *named pipe* and *shared memory segment.*
- These methods transfer data through the disk, through the kernel, and through user space.

Three Solutions

- So, our choices: File, Pipe, Shared Memory.

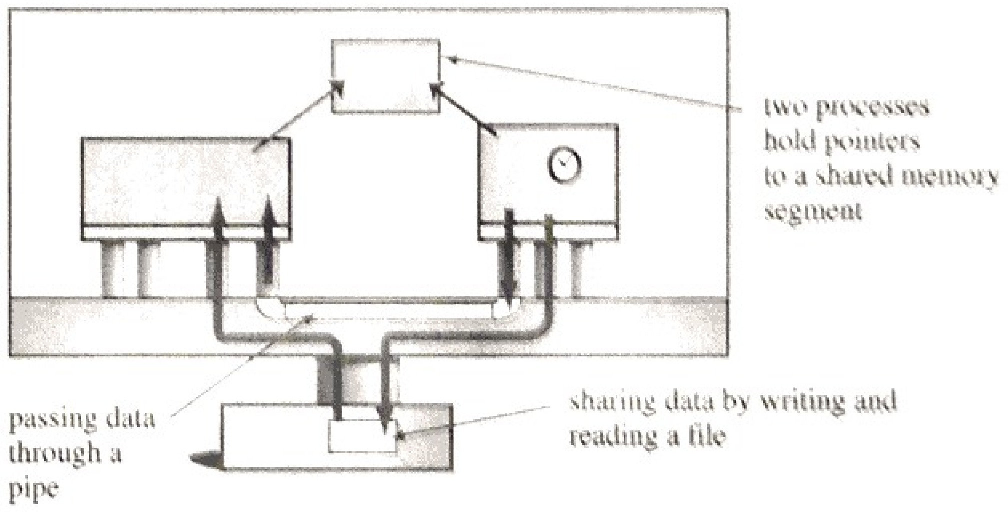


FIGURE 15.5

Three ways to transfer data.

IPC With Files

- Processes can communicate through files.
- One process writes to the file, the other processes read.
- Consider a simple Time/Data server.
- In such a setup, the server process will write the time to a file at regular intervals.
- When a client process needs to know the time, it will look inside the file.

IPC With Files

- Now, we don't have to write C, we can do this with a shell script.
- Let's look at the scripts, client and server.
- The server writes the date and file to the file every second.
- > truncates then rewrites the file.
- The client reads the contents.

Remarks on File IPC

- Access: clients have to be able to read the file. Give the server write access and limit clients to read access.
- Multiple Clients: Multiple clients can read from a file at the same time. Unix places no limits on the number of processes that can open a file at once.

Remarks on File IPC

- Race Conditions: The server updates the file by truncating and replacing the data with new data and time. Should the client read the file at the moment between truncate and rewrite, the client gets partial results.

Remarks on File IPC

- Avoiding Race Conditions: Server and clients could use a mutex (covered later).
- Or, if the server uses lseek and write instead of creat, not a problem because write is atomic.

Named Pipes

- Regular pipes only connect related processes.
- Regular pipes are created by a process and are gone when the last process closes.
- A *named pipe*, aka *FIFO*, can connect unrelated processes and are independent of processes.

Named Pipes

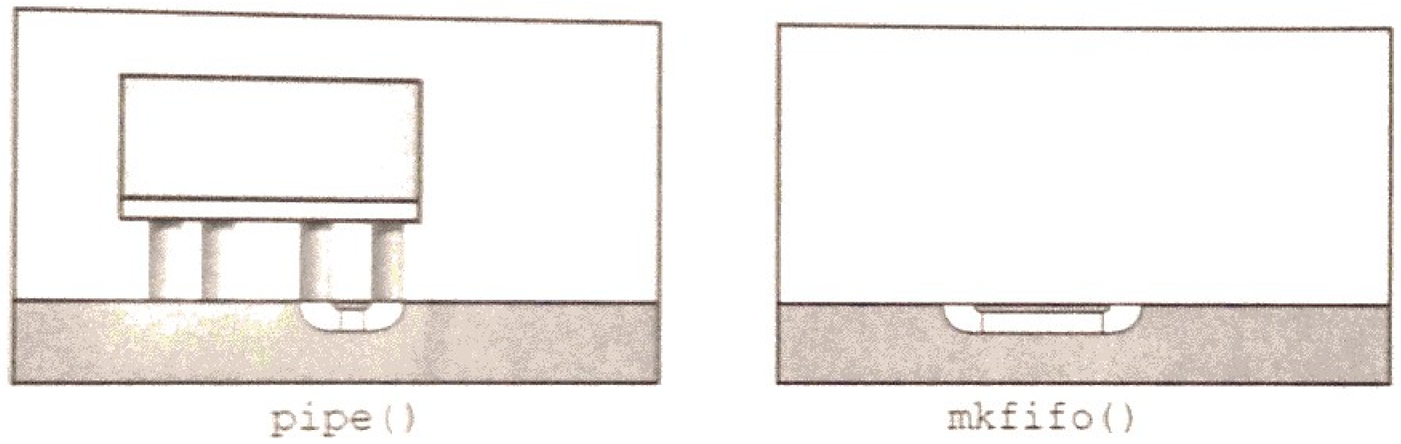


FIGURE 15.6

FIFOs are independent of processes.

- Kind of like a hose on the ground....

Named Pipes

- Anyone can pick up one end and listen.
- Anyone else can come along and speak into the other end.
- Strangers can communicate through a hose, and the hose is there even if people aren't using it.
- A FIFO is a pipe identified by a filename.

Using FIFOs

- How do I create a FIFO?

  *mkfifo(char*name, mode_t mode)*

  This creates a FIFO with a specified permission mode.

Using FIFOs

- How do I remove a FIFO?
  use

  unlink(fifoname)

  to remove a FIFO just like a regular file.

Using FIFOs

- How do I wait to speak into a FIFO?

  open(fifoname, O_WRONLY)

- open blocks until a process opens the FIFO for reading.

Using FIFOs

- How do two processes speak through a FIFO?
    - Sending process uses write and listening process uses read.
    - When the writing process calls close, the reader sees end of file.

Using FIFOs

- The Unix command mkfifo is basically a wrapper for the mkfifo system call.
- Let's look at a couple shell scripts that are the client and server for our Time server example that uses FIFOs: client_fifo, server_fifo

Notes on FIFO IPC

- Access: FIFOs use the file access permission system files use. Servers can have write permissions. Clients can be limited to read only.
- Multiple Readers: Named pipes are queues, not files. Writers add bytes to the queue, and readers remove bytes from the queue. Each client pulls the date/time from the FIFO, so the server has to write it again.

Notes on FIFO IPC

- Race Conditions: No race condition problems with the FIFO version.
  → The read and write system calls are atomic if the message size is less than the size of the pipe.
  → Reading empties the pipe, and writing to the pipe fills it.
  → Kernel blocks processes until a writer and reader are connected, locks are not needed.

Servers That Read from FIFOs

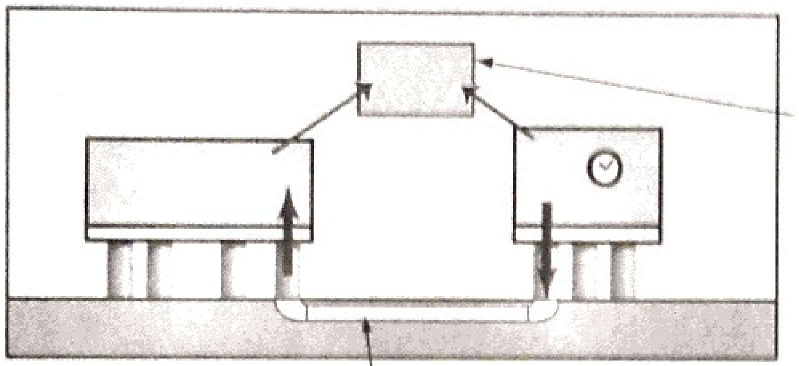
- Our time server writes into a FIFO, blocking until a client opens the FIFO for reading.
- Sometimes, a server reads from a FIFO, waiting for a client to write.

Shared Memory

Shared Memory

- How do bytes move through a file, or a FIFO?
- write copies data from process memory to a buffer in the kernel.
- read copies data from a buffer in the kernel to process memory.

Shared Memory



Using shared memory allows one process to put data directly into the memory space of another process.

Using a pipe requires copying data twice.

FIGURE 15.7

Two processes share a block of memory.

- But why do this if both processes are on the same machine in user space?

Shared Memory

- Two processes on the same machine can exchange or share data by using a *shared memory segment* – which is just a part of memory in user space both processes have pointers to.
- Subject to permissions, both can read and write the data in that memory location.
- Information is shared, not copied.

Facts about Shared Memory Segments

- Shared memory segments live in memory independent of a process.
- Shared memory segments have names, called a *key*.
- Keys are integers.
- Shared memory segments have an owner and permission bits.
- Processes can *attach* to a segment, gaining a pointer to the segment.

Using a Shared Memory Segment

- How do I create a shared memory segment?

  int seg_id = shmget(key, size-of-segment,
  flags)

  shmget locates the segment, if it exists.

Using a Shared Memory Segment

- If it doesn't exist, this can handled in the flags value, where you specify a request to create it and the inital mode (like open):

```
int sid = shmget(99,256,IPC_CREAT | 0777)
```

  - 99 is the key
  - 256 is the requested segment size in bytes
  - IPC_CREAT | 0777 specifies the segment should be created with an initial mode of 0777

Using a Shared Memory Segment

- How do I attach?

  void ptr = shmat(seg_id, NULL, flags)*

  shmat makes the segment part of the address space of the process, and returns a pointer to it.

  Use flags to specify if you want it to be read only. (SHM_RDONLY)

Using a Shared Memory Segment

Setting the second argument to NULL tells the kernel to pick an address. Which is what you want the majority of the time.

```
char* memptr = shmat(sid,NULL,0);
```

Setting the 3rd argument to 0 indicates no special flags. By default, segment is RW.

Attaching process must have correct permissions
.

Using a Shared Memory Segment

- How do I read and write to a shared memory segment?

```
    strcpy( memptr, "hello" );
```

or you can use other operations that use pointers, like mempcy() or ptr[i]

Using a Shared Memory Segment

- Once you are finished with the memory segment, time to detach:

```
shmdt(memptr);
```

- Then, we need to tell the kernel to free the segment:

```
shmctl(sid, IPC_RMID,NULL);
```

Using a Shared Memory Segment

- The memory segment is destroyed when the last process using it detaches.
- Let's look at a time/date server that uses shared memory: shtm_ts.c
- And let's look at the client: shm_tc.c.

Shared Memory IPC Remarks

- To Access: Clients must be able to read the shared memory segment.
  → The segments have file permissions that work similar to files.
  → The segment has an owner and access bits for user, group, and other.
  → A shared memory block can be protected—for example, the server has write permission, and clients can only read.

Shared Memory IPC Remarks

- Multiple Clients: Any number of clients can read from the segment at once.
- Race Conditions:
  → The server updates the memory by calling strcpy.
  → If a client reads from the segment when the server is copying a new string to it, the client may read a combo of the old and new strings.

Shared Memory IPC Remarks

- Avoiding Race Conditions:
  → The sever and clients must use some system to lock the resource.
  → The kernel provides interprocess locking mechanisms known as *semaphores*. (we'll study them later).
- For more info: man shmget, shmat, shmdt, shmctl.

Comparing Communication Methods

- Our original problem was to get a string from one process to the other.
- All three ways work: they get the data from the server.
- So, we have three versions of this system.
- Which method should we use? What are the criteria that matter?

Speed Considerations

- Sending data through files and named pipes requires more operations.
- Kernel copies data to kernel space then to user space.
- With files, the kernel could copy data to disk then later could copy from the disk.
- Worse, virtual memory allows user space to be swapped to disk, so shared memory segments could be written/read from disk.

Connected or Unconnected?

- Shared memory segments and files are like billboards.
  - Data producers post information then leave.
  - Data consumers can read at any time, and many can read it at the same time.

Connected or Unconnected?

- FIFOs require a connection.
  - Readers and Writers have to have the FIFO open before the kernel does anything, and only one client can read the message.

Range?

- How far should the message go?
- Shared memory and named pipes only work with processes on the same computer.
- Files can be stored on a shared file server, thus connecting processes across machines.

Restricted Access?

- Should everyone be able to speak with the server?
- Files, FIFOs, shared memory provide standard Unix file system permissions.

Race Conditions?

- Shared memory and shared files are trickier than pipes.
- Pipes are kernel managed queues.
- Writers put data into an end, a reader takes it from the other end.
- Processes don't care about internal structure.

Race Conditions?

- Shared files and shared memory access isn't kernel managed.
- If a process reads a file while another is rewriting it, the reading process can get bad data. We'll look at file locks and semaphores next.

Interproccess Cooperation and Coordination

File Locks

Two Kinds of Locks

- Consider two problems....
- What happens if the server is rewriting a file when a client reads it?
- The client finds it empty or incomplete.
- So, when a server is rewriting the file, clients need to wait for the server to finish.

Two Kinds of Locks

- What about the opposite situation?
- What if the client is reading the file when the server takes the file from the client then truncates and starts writing to it?
- The client sees the file changing before its very eyes.
- So, when a client is reading, the server needs to wait.

Two Kinds of Locks

- To stop these problems, two kinds of file locks are needed.
- The first, a *write* lock, says "I'm writing the file, everyone wait.“.
- The second, a *read lock*, says, "I'm reading the file. Writers wait, but everyone else can read if they want to!"

Programing with File Locks

- There are three ways to apply locks to open files: *flock*, *lockf*, and *fcntl*.
- The most flexible and portable of these is *fcntl*.

Using fcntl to Lock Files

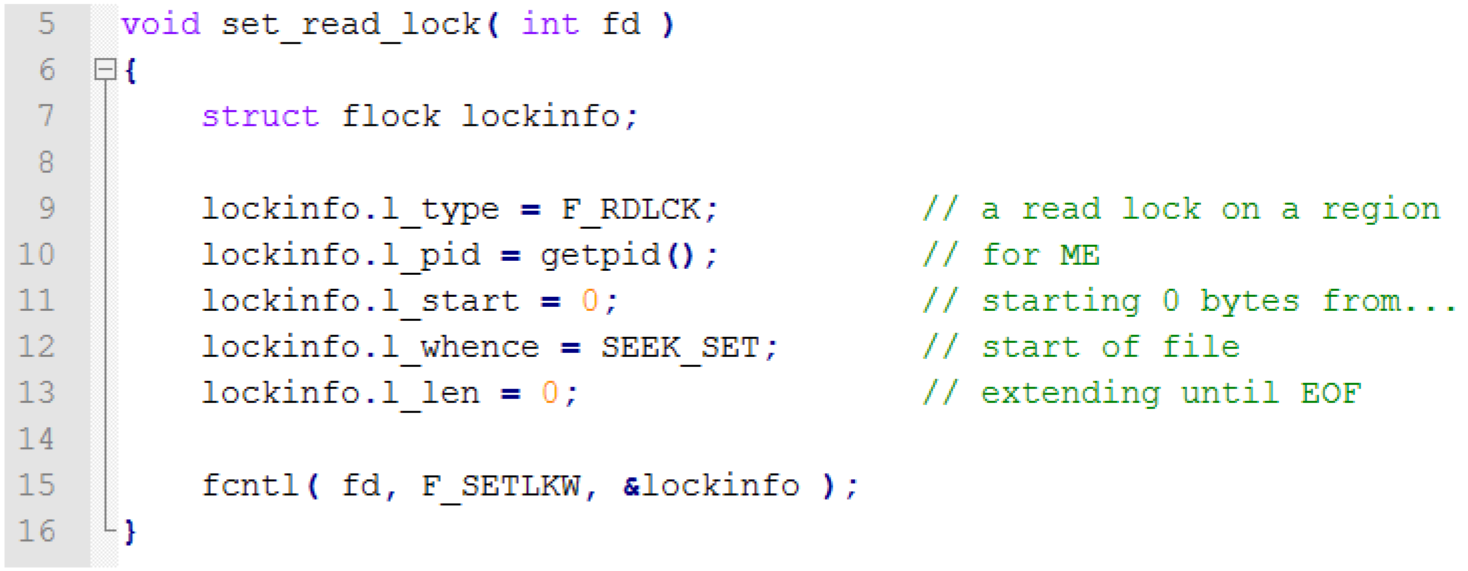
- How do I set a read lock on an open file?

```
fcntl(fd, F_SETLKW, &lockinfo)
```

→ The first argument is the file descriptor.
→ The second says you will wait for some other process to release a lock if needs be.
→ The third points to a struct flock variable.

Using fcntl to Lock Files

- Here's some code to set a read lock on a file descriptor:



```c
void set_read_lock( int fd )
{
    struct flock lockinfo;

    lockinfo.l_type = F_RDLCK;          // a read lock on a region
    lockinfo.l_pid = getpid();          // for ME
    lockinfo.l_start = 0;               // starting 0 bytes from...
    lockinfo.l_whence = SEEK_SET;       // start of file
    lockinfo.l_len = 0;                 // extending until EOF

    fcntl( fd, F_SETLKW, &lockinfo );
}
```

Using fcntl to Lock Files

- How do I set a write lock on an open file?

  fcntl(fd, F_SETLKW, &lockinfo)

  with lockinfo.l_type = F_WRLCK

Using fcntl to Lock Files

- How do I release a lock I hold?

  fcntl(fd, F_SETLKW, &lockinfo)

  with lockinfo.l_type = F_UNLCK;

Using fcntl to Lock Files

- How do I lock only part of a file?

  fcntl(fd, F_SETLKW, &lockinfo)

  with lockinfo.l_start equal to the offset of the start and lockinfo.l_len equal to length of the region.
- Setting lockinfo.l_start and lockinfo.l_len to 0 means lock the entire file.

Using fcntl to Lock Files

- Let's examine the code for a File-Based Time Server: file_ts.c
- Then, let's exame it for the client: file_tc.c.

File Locks: Summary

- Calls to fcntl with F_SETLKW block until the kernel allows the process to set the lock.
- Clients must set a read lock before reading data.
- If a server has a write lock, clients wait until the server finishes.

File Locks: Summary

- Servers must lock before updating the file.
- If a client holds a read lock, the server waits until all clients release their locks.

File Locks: Summary

- Note: Processes can Ignore Locks!!!
- Unix locks allow processes to cooperate, but doesn't enforce it!
- Locks in Unix are *advisory locks*.

Semaphores

Semaphores

- How can we prevent interference in reads and writes with shared memory?
- There are no read/write locks for memory segments, but there is a more flexible mechanism for cooperation: *semaphores*.

Semaphores

- A semaphore is a kernel variable accessible to all processes on the system.
- Processes can use them to coordinate access to resources.
- How can we use them for the time server and clients?

Counters and Operations

- The server writes to a shared segment and should wait until no clients are reading.
- The clients read from it and should wait until no server is writing.

Counters and Operations

- Let's translate these rules into statements about variable values. We want the:

  → clients to wait until number_of_writers==0

  → server to wait until number_of_readers==0

Counters and Operations

- Semaphores are systemwide global variables. Let's use one to represent the number of readers and one for number of writers.
- Managing them requires two operations.
- A reader waits until number of writers is 0, then increments the number of readers.

Counters and Operations

- When the reader finishes, it decrements the number of readers.
- The operation is similar for writers.
- The operations of waiting for the reader count to be 0 and incrementing the writer count has to happen indivisibly. (atomic)
- Processes communicating through semaphores can use several variables and apply several operations atomically.

Counters and Operations

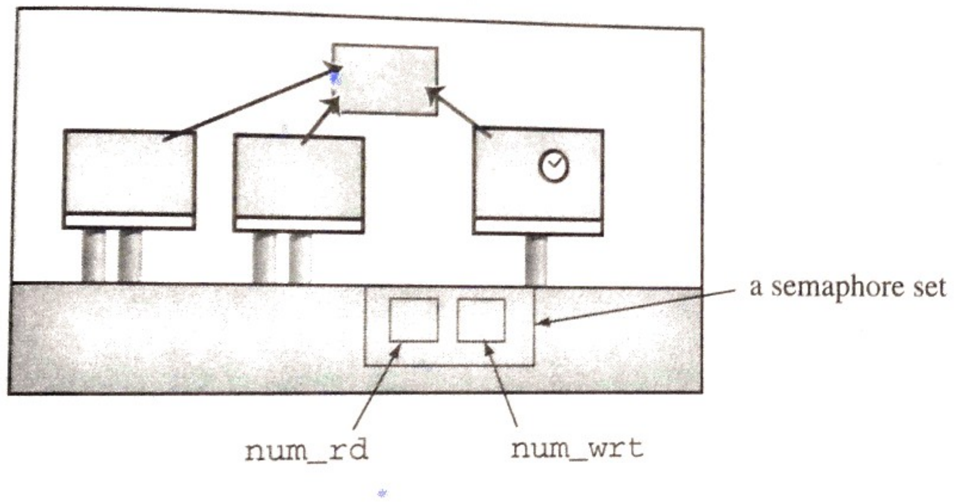


FIGURE 15.8

A semaphore set: num_readers, num_writers.

- A process can perform a set of actions on a set of semaphores all at once.

Sets of Semaphores, Sets of Actions

- The time-server system uses two semaphores (fig 15.8), and the readers/writers perform two action sets.
- Before modifying shared memory, the server must perform this semaphore set:
  - [0] wait for num_readers to be 0, and
  - [1] add 1 to num_writers

Sets of Semaphores, Sets of Actions

- When the server is done, it must perform this set:

  [0] subtract 1 from num_writers

- Before reading from the shared memory, a client must:

  [0] wait for num_writers to be 0, and
  [1] add 1 to num_readers

Sets of Semaphores, Sets of Actions

- When the client finishes, it peforms:
  [0] subtract 1 from num_readers
- Let's look at the server code: shm_ts2.c

The server does five things with the set:

- **Thing 1: Create the semaphore set:**

```
semset_id = semget( key_t key,
   int numsems,
   int flags )
```

  - creates a semaphore set containing *numsems* semaphores.
  - shm_ts2 creates a set with 2 semaphores.
  - the set has permission mode 0666.
  - semget returns an ID for the set.

The server does five things with the set:

- **Thing 2: Set both semaphores to 0:**

```
semctl( int semset_id,
   int semnum,
   int cmd,
   union semun arg )
```

We use semctl to control a semaphore set:

- First argument is the set ID
- Second is the number of a specific semaphore in the set.

The server does five things with the set:

- The third is the control command. If the control command requires an argument, the fourth argument is that argument.
- In shm_ts2, SETVAL command is used to initialize each semaphore to 0.

The server does five things with the set:

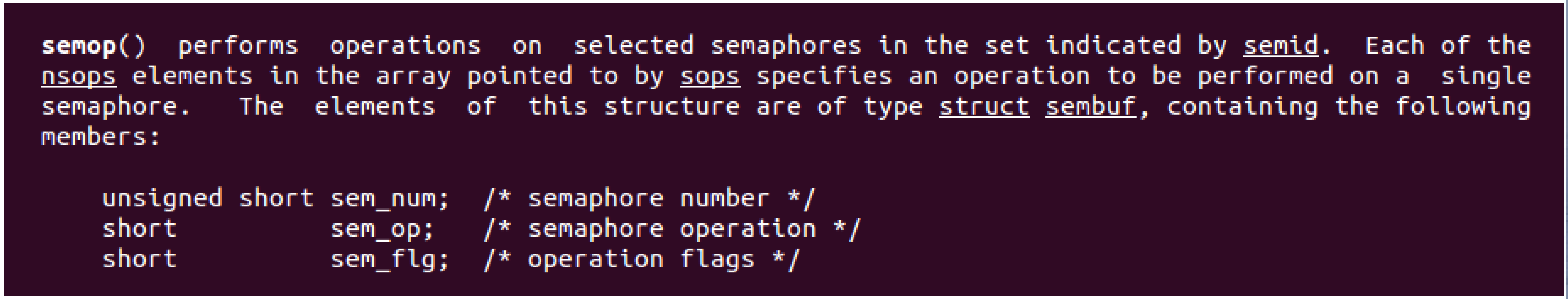
Thing 3: Wait until no readers, then increment num_writers:

```
semop( int semid,
  struct sembuf* actions,
   size_t numactions )
```

- semop applies a set of actions to a semaphore set.
- First argument identifies the set.
- Second is an array of actions.

The server does five things with the set:

- Last argument is size of the array of actions.
- Each action in the set is a struct (man semop) saying:
  "Do sem_op to semaphore sem_num with options sem_flg“



```
semop()  performs  operations  on  selected semaphores in the set indicated by semid.  Each of the
nsops elements in the array pointed to by sops specifies an operation to be performed on a  single
semaphore.   The  elements  of  this structure are of type struct sembuf, containing the following
members:

    unsigned short sem_num;  /* semaphore number */
    short          sem_op;   /* semaphore operation */
    short          sem_flg;  /* operation flags */
```

The server does five things with the set:

- The entire actions set is done as a group.
- The wait_and_lock function performs two actions:
  1. wait for the number of readers to equal 0
  2. increment the number of writers.

The server does five things with the set:

- So, an array of two actions is created: action 0 says wait for zero on semaphore 0. Action 1 says add 1 to semaphore 1.
- The process blocks until both actions can happen.
- As soon as reader count is 0, the writer count increments.

The server does five things with the set:

- SEM_UNDO tells the kernel to undo these operations when the process exits.
- Why do this? In this program, incrementing the writer count locks the memory segment – if the process dies before decrementing, other processes could never use that segment.

The server does five things with the set:

- **Thing 4. Decrement num_writers:**
  - in function release_lock, only one thing to do: decrement the writers count. So,
  - Call semop with an array containing this action.
  - if a client is waiting for the count to hit 0, the client resumes execution

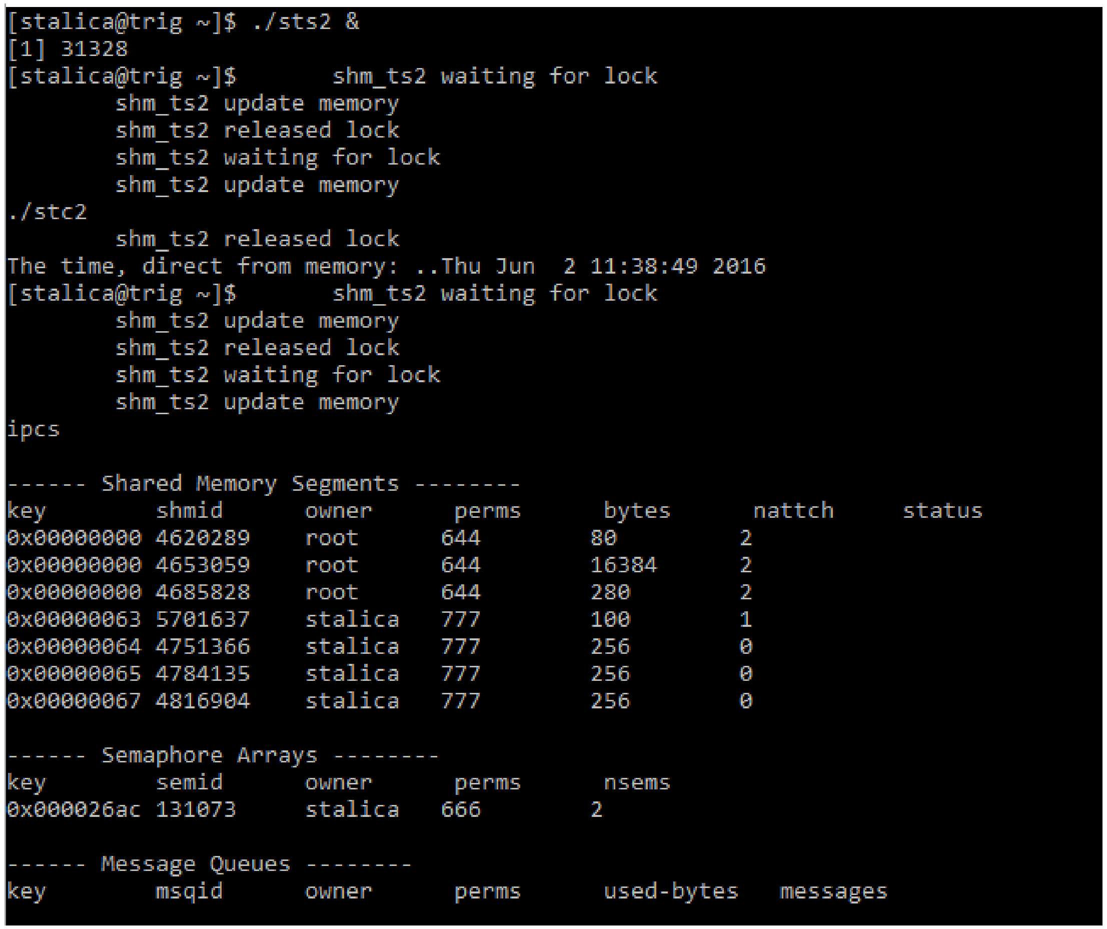
The server does five things with the set:

- **Thing 5. Deletes the semaphore:**

```
semctl( semset_id, 0, IPC_RMID, 0)
```

  - At the end, call semctl again, this time deleting the semaphore, specified with IPC_RMID.

Compile and Test



```
[stalica@trig ~]$ ./sts2 &
[1] 31328
[stalica@trig ~]$       shm_ts2 waiting for lock
        shm_ts2 update memory
        shm_ts2 released lock
        shm_ts2 waiting for lock
        shm_ts2 update memory
./stc2
        shm_ts2 released lock
The time, direct from memory: ..Thu Jun  2 11:38:49 2016
[stalica@trig ~]$       shm_ts2 waiting for lock
        shm_ts2 update memory
        shm_ts2 released lock
        shm_ts2 waiting for lock
        shm_ts2 update memory
ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 4620289    root       644        80         2
0x00000000 4653059    root       644        16384      2
0x00000000 4685828    root       644        280        2
0x00000063 5701637    stalica    777        100        1
0x00000064 4751366    stalica    777        256        0
0x00000065 4784135    stalica    777        256        0
0x00000067 4816904    stalica    777        256        0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x000026ac 131073     stalica    666        2

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

Waiting for a Semaphore to Become Positive

- The client waits until the number-of-writers semaphore is 0.
- Server waits until number-of-readers is 0.
- Semaphores can never be negative: the kernel will block calls using a negative semaphore until it is 0.

Waiting for a Semaphore to Become Positive

- sem_op could look like this:
  - sem_op's value is *positive*:
    Action: increment semaphore by sem_op
  - sem_op's value is *zero*:
    Action: block until semaphore equals 0
  - sem_op's value is *negative*:
    Action: Block until adding sem_op to the semaphore causes it to not be negative

Further reading

- man semctl, semget, semop

A Print Spooler

A print spooler

- Some apps work opposite of how the time server/client work: multiple clients sending data to a single server.
- An example is a print spooler.
- What questions apply?

Many Writers, One Reader

- Multiple users share a single printer.
- How do we use client/server design to create a printer-sharing program?
- A printing program has to accept multiple print requests but produce only a single output stream to a printer.
- What do servers do? Clients?
- How do they communicate?

Many Writers, One Reader

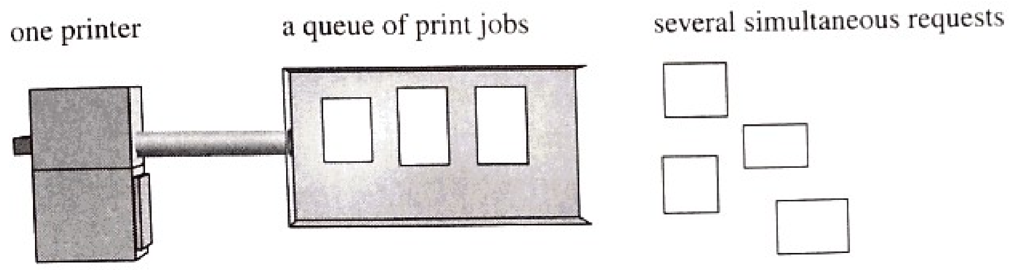


FIGURE 15.9

Many sources of data, one printer.

- What are the functional units? What data/messages are exchanged?

Many Writers, One Reader

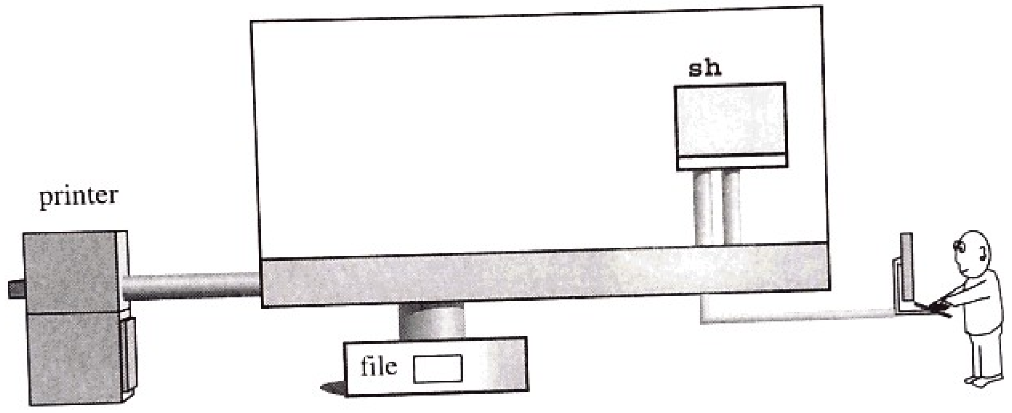


FIGURE 15.10
Getting a file to the printer.

- Simplest way to print a file is the command:

```
cat filename > /dev/lp1       or,
cp filename /dev/lp1
```

How About Write Locks?

- A lock-based file-copy program works.
- If one program has a lock on the printer, other file-copy programs block until the first job releases the lock.
- Which goes next?
- The kernel will unblock one, but it might not be the next in line.
- Fair?

How About Write Locks?

- Some people could cheat and not use our special program.
- Some files might require special processing, an image file for example, might need conversion to graphics.
- Clients might not know how to do this, resulting in garbage prints.
- Solution? Centralization.

A Client/Server Model

- This model solves the problems discussed.
- A server program known as the *line printer daemon* has permission to write to the printer and others don't.

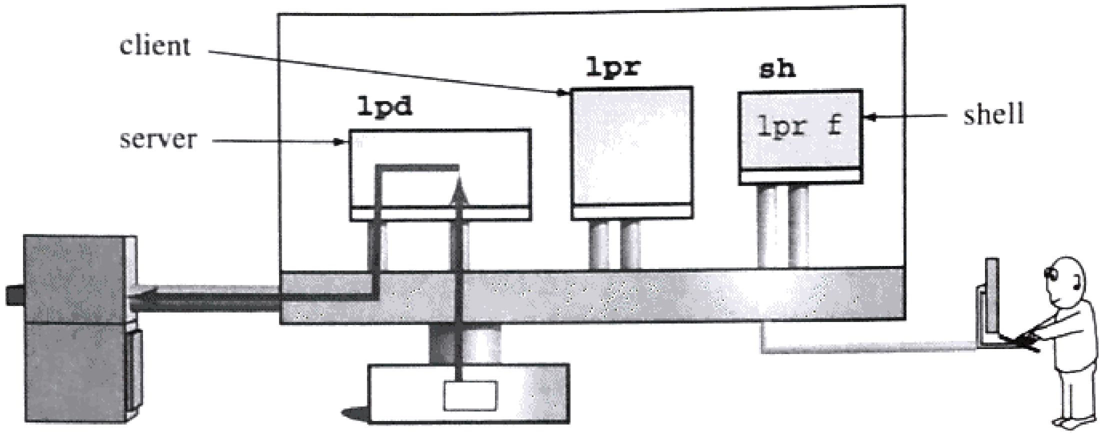


FIGURE 15.11

Client/server printing system.

A Client/Server Model

- Users run a client called *lpr* when they want to print.
- *lpr* makes a file copy then places the copy in the print jobs queue.
- The user can then edit/delete the original, since the copy gets printed.
- Additionally, the printer daemon can apply conversion programs so images and fonts print correctly.

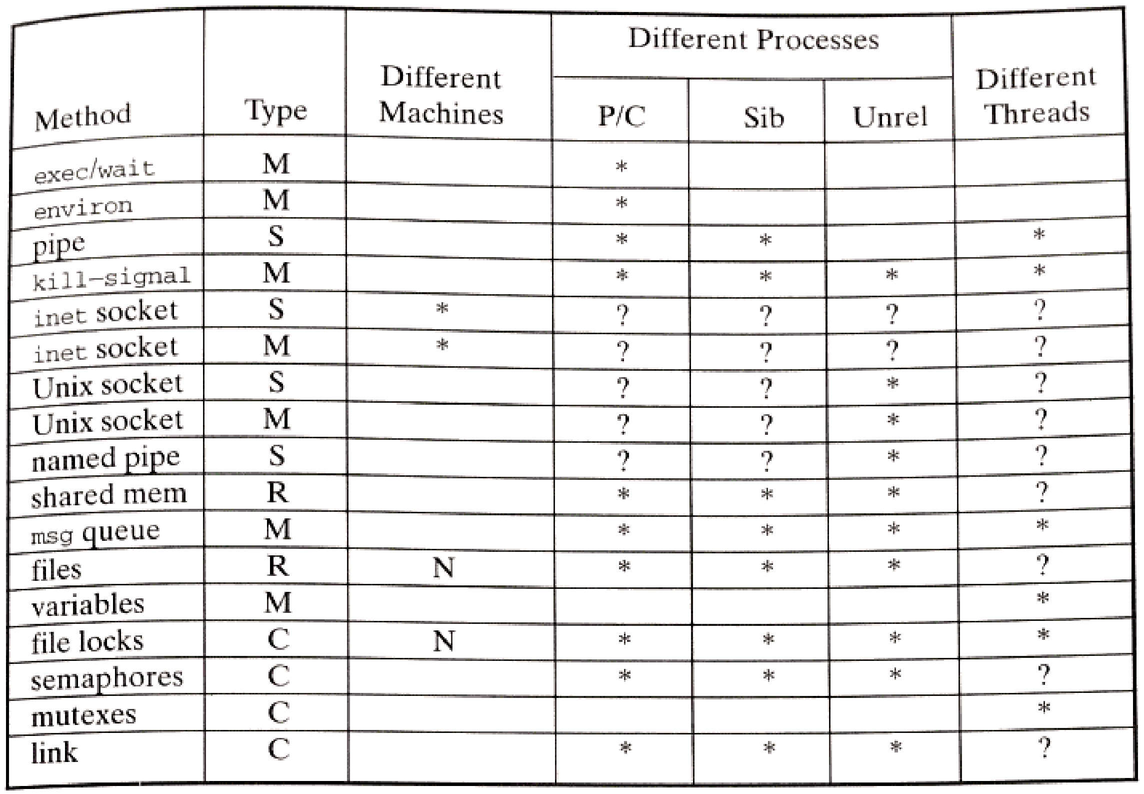
A Client/Server Model

- Some questions:
- How do the client/server communicate?
- What data is exchanged?
- What if server is on a different coputer?
- What about choice of communication methods?
- Locks?
- Different solutions have been found, resulting in different implementations.

15.6: IPC Overview

IPC Overview

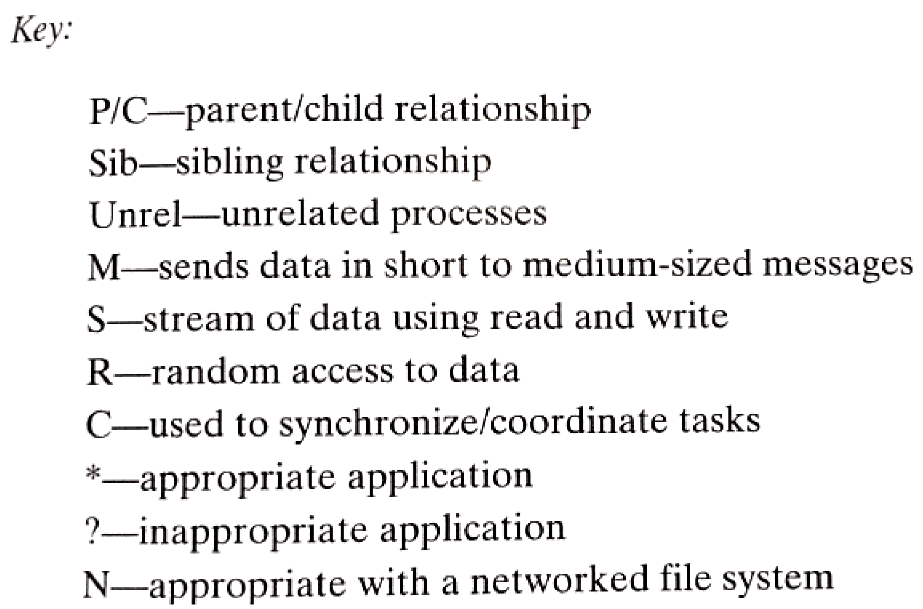
- We've seen a lot of forms of ipc. Here's a summary:



Method	Type	Different Machines	Different Processes			Different Threads
			P/C	Sib	Unrel	
exec/wait	M		*			
environ	M		*			
pipe	S		*	*		*
kill-signal	M		*	*	*	*
inet socket	S	*	?	?	?	?
inet socket	M	*	?	?	?	?
Unix socket	S		?	?	*	?
Unix socket	M		?	?	*	?
named pipe	S		?	?	*	?
shared mem	R		*	*	*	?
msg queue	M		*	*	*	*
files	R	N	*	*	*	?
variables	M					*
file locks	C	N	*	*	*	*
semaphores	C		*	*	*	?
mutexes	C					*
link	C		*	*	*	?

IPC Overview

- Note, we didn't cover everything in this table. Let's summarize what we did.



Key:

P/C—parent/child relationship
Sib—sibling relationship
Unrel—unrelated processes
M—sends data in short to medium-sized messages
S—stream of data using read and write
R—random access to data
C—used to synchronize/coordinate tasks
*—appropriate application
?—inappropriate application
N—appropriate with a networked file system

Explanations

fork-execv-argv, exit-wait

- Used to call a program with arguments list and for called program to return an integer value to the caller.
- Parent calls *fork* to create a new process
- The program, in the new process, calls *execv* to run a new program, passing the array of strings.
- Child sends a value back with *exit.*
- Parent receives it with *exit*.
- Message oriented, related processes only, same machine.

pipe

- One-way stream of data created by a process.
- Two file descriptors connected in the kernel.
- Data written to one of the fds can be read from the other.
- A process calling *fork* after creating a pipe allows for the new process to read/write to the same pipe.
- Stream oriented, usually one-way, related processes, one machine.

Named Pipes (FIFOs)

- Like a regular pipe, but can connect unrelated processes.
- Identified by a filename
- Writer opens the file for writing using *open*.
- Reader opens the file for reading using *open*.
- Only go one way.
- Stream oriented, one-way, unrelated, within one machine.

File Locks

- Processes can set locks on file sections.
- One process locks a file section to modify it.
- Another process tries to lock the file, but is suspended or told the section is locked.
- The locks let one process know who is changing or reading a file.
- *flock*, *lockf*, *fcntl* can be used to set and test locks.
- Message oriented, *n*-way, unrelated, within one machine.

Shared Memory

- Each process has its own data space.
- Variables a program creates are available only to that process.
- A process may use *shmget* and *shmat* to create a memory chunk usable my multiple processes.
- Data written to the chunk can be read by another process.
- Most efficient, no data transfer required.
- Random access, *n*-way, unrelated, within one machine.

Semaphores

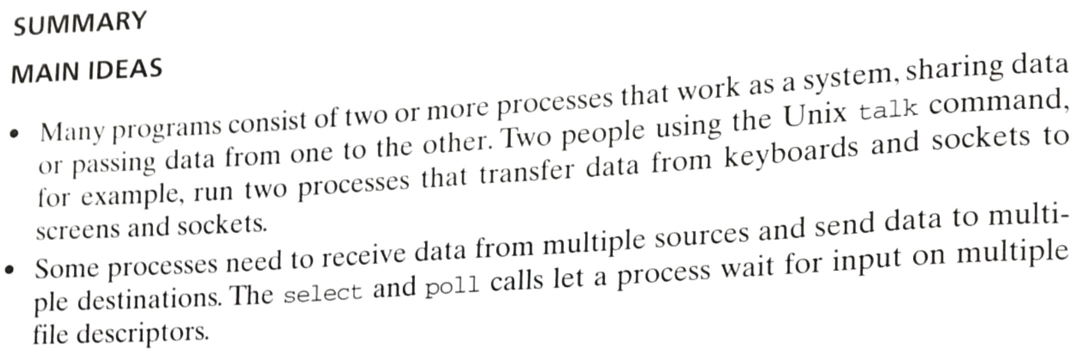
- Systemwide variables programs use to coordinate work.
- Processes can increment, decrement, and wait for semaphores to reach a value.
- Message oriented, *n*-way, unrelated, within one machine.

Files

- Can be opened by multiple processes at once.
- If a process writes to a file, another process can read the data.
- With careful planning complex communication can be implemented using plain text files.
- Random access, *n*-way, unrelated, NFS allows cross-machine communication

Summary

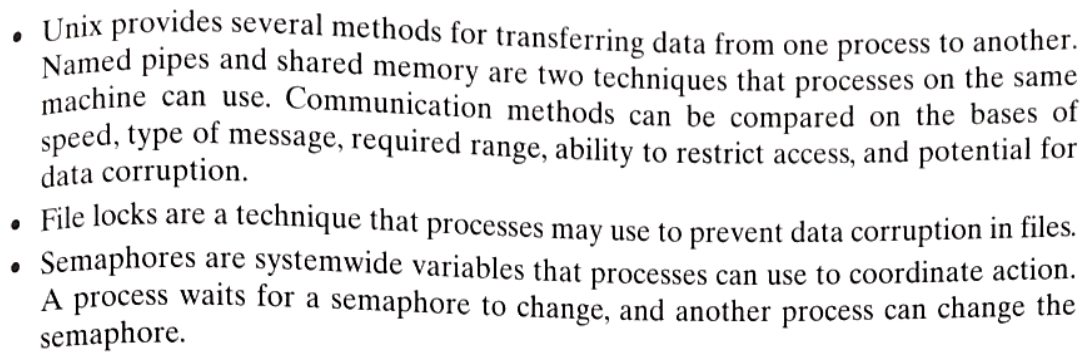
Main Ideas



SUMMARY

MAIN IDEAS

- Many programs consist of two or more processes that work as a system, sharing data or passing data from one to the other. Two people using the Unix `talk` command, for example, run two processes that transfer data from keyboards and sockets to screens and sockets.
- Some processes need to receive data from multiple sources and send data to multiple destinations. The `select` and `poll` calls let a process wait for input on multiple file descriptors.



- Unix provides several methods for transferring data from one process to another. Named pipes and shared memory are two techniques that processes on the same machine can use. Communication methods can be compared on the bases of speed, type of message, required range, ability to restrict access, and potential for data corruption.
- File locks are a technique that processes may use to prevent data corruption in files.
- Semaphores are systemwide variables that processes can use to coordinate action. A process waits for a semaphore to change, and another process can change the semaphore.