Linux Internals

Linux Inter Process Communication

Contents

- Module Coverage
  - What is IPC?
  - Different IPC mechanisms
  - Shared Memory
  - Mapped Memory (mmap)
  - Pipe and FIFO
  - Semaphores
  - Sockets

What is IPC?

- Interprocess communication (IPC) is the transfer of data among processes
- This transfer of data can be between,related processes (parent-child relationship),unrelated processes, and processes on different machines
- Example:
  - We use the command "ls | lpr" to print the filenames in a directory
  - The ls process writes data into the pipe, and the lpr process reads data from the pipe.

Different IPC mechanisms

- The different IPC mechanism available in Linux are:
  - Pipes
    - Pipes permit sequential communication from one process to a related process
  - FIFOs
    - FIFOs are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem
  - Message Queues
    - Message Queues can be used by processes to read and write messages
  - Shared memory
    - Shared memory allows processes to communicate by reading and writing to a specified memory location.

Different IPC mechanisms (Contd..)

- Mapped memory
  - Mapped memory is similar to shared memory, except that it is associated with a file in the filesystem
  - Contents of a file are mapped to memory
- Semaphores
  - Semaphores are used to synchronize access to shared resources
- Sockets
  - Sockets are used for communication between processes on same or different computers

Pipes

- A pipe is a communication device that permits unidirectional communication
- Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes (I.e related processes)
- In a shell, the symbol | presents a pipe. For example, this shell command causes the shell to produce two child processes, one for ls and one for less. The two child processes communicate using a pipe.
  - % ls | less
- A pipe's data capacity is limited. The pipe automatically synchronizes the two processes

Creating Pipes

- To create a pipe, invoke the pipe system call.
  - Supply an integer array of size 2
  - The call to pipe stores the reading file descriptor in array position 0 and the writing file descriptor in position 1
- For example, consider this code:

```
int pipe_fds[2];
int read_fd;
int write_fd;
pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Communication Between Parent and Child Processes

- Parent creates pipe
- A fork spawns a child process
- The child inherits the pipe file descriptors
- The parent writes a data to the pipe
- and the child reads it out

FIFOs

- A *first-in, first-out (FIFO)* file is a pipe that has a name in the filesystem.
- Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other
- FIFOs are also called *named pipes*
- You can make a FIFO using the mkfifo command or system call

```
% mkfifo /tmp/fifo
% ls -l /tmp/fifo
    prw-rw-rw- 1 samuel users 0 Jan 16 14:04 /tmp/fifo
```

- The first character of the output from ls is p, indicating that this file is actually a FIFO (named pipe)

Creating FIFO

- Create a FIFO programmatically using the mkfifo function.
- The first argument is the path at which to create the FIFO; the second parameter specifies the pipe's owner, group, and world permissions
- If the pipe cannot be created (for instance, if a file with that name already exists), mkfifo returns –1
- Include <sys/types.h> and <sys/stat.h> if you call mkfifo

Accessing FIFO

- A FIFO is accessed just like an ordinary file
  - To communicate through a FIFO, one program must open it for writing, and another program must open it for reading
- Either low-level I/O functions (open, write, read, close, and so on, or C library I/O functions (fopen, fprintf, fscanf, fclose, and so on) may be used
  - For example to write a buffer of data to a FIFO using low-level I/O

```
int fd = open (fifo_path, O_WRONLY);
write (fd, data, data_length);
close (fd);
```

  - To read a string from the FIFO using C library I/O functions

```
FILE* fifo = fopen (fifo_path, “r”);
fscanf (fifo, “%s”, buffer);
fclose (fifo);
```

Shared Memory

- Shared memory allows two or more processes to access the same memory.
- Access to the shared memory is as fast as accessing a process’s nonshared memory, and it does not require a system call or entry to the kernel.
- All processes share the same piece of memory.
- The kernel does not synchronize accesses to shared memory, we must provide our own synchronization mechanisms. A common strategy is to use semaphores.

Shared Memory Issues

- Shared memory segments allow fast communication among any number of processes
- Each process can both read and write, but must follow a protocol to avoid race conditions
- Processes must make arrangements to use the same key to use a shared segment

The memory model

- To use a shared memory segment, one process must allocate the segment
- Then each process desiring to access the segment must attach the segment
- After finishing its use of the segment, each process detaches the segment.
- At some point, one process must deallocate the segment

Shared memory Data Structure

```
struct shmid_ds
{
   struct ipc_perm shm_perm;      /* operation perms */
   int    shm_segsz;              /* size of segment (bytes) */
   time_t shm_atime;              /* last attach time */
   time_t  shm_dtime;             /* last detach time */
   time_t  shm_ctime;             /* last change time */
   unsigned short  shm_cpid;      /* pid of creator */
   unsigned short  shm_lpid;      /* pid of last operator */
   short   shm_nattch;            /* no. of current attaches
  */
.....
};
```

Shared Memory Allocation

- A process allocates a shared memory segment using shmget(key,size,options) system call
- Its first parameter is an integer key that specifies which segment to create
- Its second parameter specifies the number of bytes in the segment.
- The third parameter is the bitwise or of flag values that specify options to shmget
  - IPC_CREAT
  - IPC_EXCL
  - Mode flags

shmget() system call

int shmget (key_t key, int size, int shmflg);

returns: shared memory segment identifier on success

Arg1 : keyvalue

Arg2 : size

Arg3 :

IPC_CREAT

Create the segment if it doesn't already exist in the kernel.

IPC_EXCL

When used with IPC_CREAT, fail if segment already exists.

Shared Memory Allocation (Contd..)

- For example

```
int segment_id = shmget (shm_key, getpagesize (),
  IPC_CREAT | S_IRUSR | S_IWUSR );
```

- If the call succeeds, shmget returns a segment identifier
- If the shared memory segment already exists, the access permissions are verified and a check is made to ensure that the segment is not marked for destruction.

Attachment and Detachment

- To make the shared memory segment available, a process must use shmat(shmid,address,flags)
  - Pass it the shared memory segment identifier shmid returned by shmget
  - The second argument is a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address
  - The third argument is a flag, which can include the following:
    - SHM_RND indicates that the address specified for the second parameter should be rounded down to a multiple of the page size. If you don't specify this flag, you must page-align the second argument to shmat yourself
    - SHM_RDONLY indicates that the segment will be only read, not written

Attachment of a Segment

- SYSTEM CALL: shmat();
  PROTOTYPE: int shmat (int shmid, char *shmaddr, int shmflg);
- RETURNS: address at which segment was attached to the process, or -1 on error
- char *attach_segment(int shmid)
- {
- return(shmat(shmid, 0, 0));
- }

Attachment and Detachment (Contd..)

- If the call succeeds, it returns the address of the attached shared segment
- Children created by calls to fork inherit attached shared segments; they can detach the shared memory segments, if desired.
- When you’re finished with a shared memory segment, the segment should be detached using shmdt(address)
- Pass it the address returned by shmat
- If the segment has been deallocated and this was the last process using it, it is removed. Calls to exit and any of the exec family automatically detach segments

Controlling and Deallocating Shared Memory

- The shmctl call returns information about a shared memory segment and can remove it.
  - The first parameter is a shared memory segment identifier
  - To obtain information about a shared memory segment, pass IPC_STAT as the second argument and a pointer to a struct shmid_ds
  - To remove a segment, pass IPC_RMID as the second argument, and pass NULL as the third argument.The segment is removed when the last process that has attached it finally detaches it.
  - Each shared memory segment should be explicitly deallocated using shmctl when you’re finished with it, to avoid violating the systemwide limit on the total number of shared memory segments. Invoking exit and exec detaches memory segments but does not deallocate them.

Controlling and Deallocating Shared Memory (contd)

- The ipcs command provides information on IPC facilities.
- Use -m flag to obtain information about shared memory
- % ipcs -m

  ------ Shared Memory Segments --------

  key shmid owner perms bytes nattch

  0x00000000 1627649 user 640 25600 0
- Use ipcrm command to remove a shm.
- % ipcrm shm 1627649

Semaphore

A semaphore is a protected variable and constitutes the classic method for restricting access to shared resources (e.g. storage) in a multiprogramming environment

It supports 2 operations

P(sv)-decrement sv

V(sv)-increment sv

Semaphore definitions

#include<sys/sem.h>

Int semctl(int sem_id,int sem_num,int command…)

Int semget(key_t key,int num_sems,int sem_flags);

Int semop(int sem_id,struct sembuf *sem_ops,size_t num_sem_ops);

Processes Semaphores

- The calls semget and semctl allocate and deallocate semaphores
- Invoke semget with a key specifying a semaphore set, the number of semaphores in the set, and permission flags as for shmget
- The return value is a semaphore set identifier.
- The last process to use a semaphore set must explicitly remove it to ensure that the operating system does not run out of semaphores.
- To do so, invoke semctl with the semaphore identifier, the number of semaphores in the set, IPC_RMID as the third argument, and any union semun value as the fourth argument (which is ignored)

Processes Semaphores (Contd..)

```
/* We must define union semun ourselves. */
union semun {
int val;
struct semid_ds *buf;
unsigned short int *array;
struct seminfo *__buf;
};
/* Obtain a binary semaphore’s ID, allocating if necessary. */
int binary_semaphore_allocation (key_t key, int sem_flags) {
return semget (key, 1, sem_flags);
}
/* Deallocate a binary semaphore. All users must have finished their
use. Returns -1 on failure. */
int binary_semaphore_deallocate (int semid) {
union semun ignored_argument;
return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

Initializing Semaphores

```
/* We must define union semun ourselves. */
union semun {
int val;
struct semid_ds *buf;
unsigned short int *array;
struct seminfo *__buf;
};
/* Initialize a binary semaphore with a value of 1. */
int binary_semaphore_initialize (int semid)
{
union semun argument;
unsigned short values[1];
values[0] = 1;
argument.array = values;
return semctl (semid, 0, SETALL, argument);
}
```

Wait and Post Operations

- The fields of struct sembuf are listed here:
  - **sem_num**
    - This is the semaphore number in the semaphore set on which the operation is performed
  - **sem_op**
    - This is an integer that specifies the semaphore operation
  - sem_flg
    - This is a flag value. Specify IPC_NOWAIT to prevent the operation from blocking; if the operation would have blocked, the call to semop fails instead.

Mapped Memory

- Mapped memory permits different processes to communicate via a shared file
- Although you can think of mapped memory as using a shared memory segment with a name, you should be aware that there are technical differences
- Mapped memory can be used for interprocess communication or as an easy way to access the contents of a file
- Mapped memory forms an association between a file and a process’s memory
- You can think of mapped memory as allocating a buffer to hold a file’s entire contents, and then reading the file into the buffer and (if the buffer is modified) writing the buffer back out to the file afterward

Mapping an Ordinary File

- mmap call is used to map an ordinary file to a process's memory
  - file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,MAP_SHARED, fd, 0);
- Arguments
  - The first argument is the address at which you would like Linux to map the file into your process's address space; the value NULL allows Linux to choose an available start address.
  - The second argument is the length of the map in bytes.
  - The third argument specifies the protection on the mapped address range.

Mapping an Ordinary File (Contd..)

- The fourth argument is a flag value that specifies additional options(MAP_FIXED, MAP_PRIVATE, MAP_SHARED)
  - MAP_FIXED—If you specify this flag, Linux uses the address you request to map the file.This address must be page-aligned.
  - MAP_PRIVATE—Writes to the memory range should not be written back to the attached file, but to a private copy of the file. No other process sees these writes. This mode may not be used with MAP_SHARED.
  - MAP_SHARED—Writes are immediately reflected in the underlying file rather than buffering writes. Use this mode when using mapped memory for IPC.This mode may not be used with MAP_PRIVATE
- The fifth argument is a file descriptor opened to the file to be mapped.
- The last argument is the offset from the beginning of the file from which to start the map

Mapping an Ordinary File (Contd..)

- If mmap call succeeds, it returns a pointer to the beginning of the memory. On failure, it returns MAP_FAILED.
- When you’re finished with a memory mapping, release it by using munmap.
  - Pass it the start address and length of the mapped memory region.
  - Linux automatically unmaps mapped regions when a process terminates

MAP_SHARED flag

- Different processes can communicate using memory-mapped regions associated with the same file.
- Specify the MAP_SHARED flag so that any writes to these regions are immediately transferred to the underlying file and made visible to other processes.
- If you don't specify this flag, Linux may buffer writes before transferring them to the file.
- As with shared memory segments, users of memory-mapped regions must establish and follow a protocol to avoid race conditions

Sockets

- A socket is a bidirectional communication mechanism that can be used to communicate with another process on the same machine or with a process running on other machine.

Socket Concepts

- When data is sent through a socket, it is packaged into chunks called packets.
- A socket address struct identifies one end of a socket connection.
- Client is the process initiating the connection, and server is the process waiting to accept connections
- We can read from or write to the socket using read, write calls as with files.
- We can also use functions like send and recv instead of read and write. Send and Recv are specific to socket IO and provide some advanced options.

Socket Concepts (contd)

- Similar to a file, a Socket is represented by file descriptors.
- A port number distinguishes among multiple sockets on the same host.

  Eg:

  Web servers use port number 80

  SMTP servers use port number 25

- For your own server applications, use port numbers greater than 1024

Socket Concepts (contd)

- DNS Names
  - It is easier to remember names than numbers
  - The Domain Name Service (DNS) associates names such as www.xyz.com with unique IP numbers
  - To convert human-readable hostnames, either numbers in standard dot notation (such as 10.0.0.1) or DNS names (such as www.xyz.com) into 32-bit IP numbers, you can use gethostbyname() function

Socket System Calls

- socket — Creates a socket
- Close — Destroys a socket
- connect — Creates a connection between two sockets
- bind — Labels a server socket with an address
- listen — Configures a socket to accept connections
- accept — Accepts a connection and creates a new socket for the connection

Creating and destroying Sockets

- The socket() and close() functions create and destroy sockets, respectively
- When you create a socket, you must specify three parameters:
  - Connection type,
  - namespace (local or internet), and
  - Protocol
- Connection type
  - Connection type controls how the socket treats transmitted data and specifies the number of communication partners
  - Connection-oriented type guarantees delivery of all packets in the order they were sent
  - If packets are lost or reordered by problems in the network, the receiver automatically requests their retransmission from the sender

Creating and destroying Sockets (contd)

- Connection type (contd)
  - Connectionless (Datagram) type does not guarantee delivery or arrival order
  - Packets may be lost or reordered in transit due to network errors or other conditions
  - The sender specifies the receiver’s address for each individual message
- For specifying connection type parameter, use SOCK_STREAM for connection-oriented type, and SOCK_DGRAM for Connectionless (Datagram) type

Creating and destroying Sockets (contd)

- Socket Namespace
  - A socket namespace specifies how socket addresses are specified
  - For specifying namespace parameter, use
    - PF_LOCAL or PF_UNIX for local namespace, and PF_INET for Internet namespace.
  - Socket addresses in the “local namespace” are ordinary filenames
  - In “Internet namespace”, a socket address is composed of host IP address and a port number.

Creating and destroying Sockets (contd)

- Protocol
  - A protocol specifies how data is transmitted. Eg: TCP/IP, UDP or UNIX local communication protocol, etc.
  - Each protocol is valid for a particular namespace-style combination
  - There is usually one best protocol for each such pair, specifying 0 is recommended
- If call to socket() succeeds, it returns a file descriptor for the socket
- For closing the connection, call close()

Connect( ) function

- Client calls connect() function, specifying the address of a server socket to which it wants to connect.
- connect(sock_fd, socket address struct, length of socket address struct)
- Socket address formats differ according to the socket namespace.

Server Socket Calls

- A server does the following :
  - creates a socket,
  - binds an address to its socket,
  - calls listen(), that enables connections to the socket,
  - calls accept() to accept incoming connections,
  - and then closes the socket
- Data isn’t read and written directly via the server socket
- Instead, each time a server program accepts a new connection, Linux creates a separate socket (connected socket) for transferring data
- bind( )function
  - An address must be bound to the server’s socket using bind.
  - bind(sock_fd, socket address struct, length of socket address struct)

Server Socket Calls (contd)

- After an address is bound to the socket, server must call listen( )
  - Listen indicates that server is ready to accept connections
  - listen(sock_fd, num of conn)
  - The second argument specifies how many pending connections can be queued
  - If the queue is full, additional connections will be rejected
  - Note, this does not limit the total number of connections that a server can handle; it limits just the number of clients attempting to connect that have not yet been accepted

Server Socket Calls (contd)

- A server accepts a connection request from a client by invoking accept( )
  - accept(sock_fd, socket address struct, length of socket address struct)
  - Second argument contains the address of the client
  - The call to accept creates a new socket for communicating with the client and returns the corresponding file descriptor
  - The original server socket continues to accept new client connections

LOCAL or UNIX domain sockets

- Sockets connecting processes on the same computer
- They are called local sockets or UNIX-domain sockets
- Their socket addresses are specified by filenames
- The socket's name is specified in struct sockaddr_un :
    - set the sun_family field to AF_LOCAL
    - sun_path field specifies the filename
- Use SUN_LEN macro to get the length of this socket address struct
- The only permissible protocol value for the local namespace is 0

LOCAL or UNIX domain sockets (contd)

- Because it resides in file system, a local socket is listed as a file

  For example,

  % ls -l /tmp/socket

  srwxrwx--x 1 user group 0 Nov 13 19:18 /tmp/socket

- Notice the initial s
- Call unlink to remove a local socket file

Internet-Domain Sockets

- Internet-domain sockets are used to connect processes on different machines connected by a network
- The most common protocols for Internet namespace are TCP/IP
- Internet socket addresses are stored in struct sockaddr_in :
  - Set the sin_family field to AF_INET
  - The sin_addr field stores the Internet address of the desired machine as a 32-bit IP address
- Specify port number to differentiate between sockets on the machine.

Review

- Topics Covered
  - What is IPC?
  - Different IPC mechanisms
  - Shared Memory
  - Mapped Memory (mmap)
  - Semaphores
  - Sockets