

IPC in Linux

Ngo Duy Hop

Agenda

- ❑ IPC Mechanisms on Linux – introduction
- ❑ Introduce some IPC mechanisms:
- ❑ IPC commands

IPC MECHANISMS ON LINUX - INTRODUCTION

- Inter-Process-Communication (or IPC for short) are mechanisms provided by the kernel to allow processes to communicate with each other.
- Communication can be of two types:
 - Between related processes initiating from only one process, such as parent and child processes.
 - Between unrelated processes or two or more different processes.
- There are two standards:
 - System V (AT&T):
 - AT & T introduced (1983) three new forms of IPC facilities namely message queues, shared memory, and semaphores.
 - SYSTEM V IPC covers all the IPC mechanisms viz., pipes, named pipes, message queues, signals, semaphores, and shared memory. It also covers socket and Unix Domain sockets.
 - POSIX(IEEE):
 - Portable Operating System Interface standards specified by IEEE to define application programming interface (API). POSIX covers all the three forms of IPC


Introduce some IPC mechanisms:

1. Signals
2. Pipes (named and unnamed)
3. Message queues
4. Shared memory

1.Signals

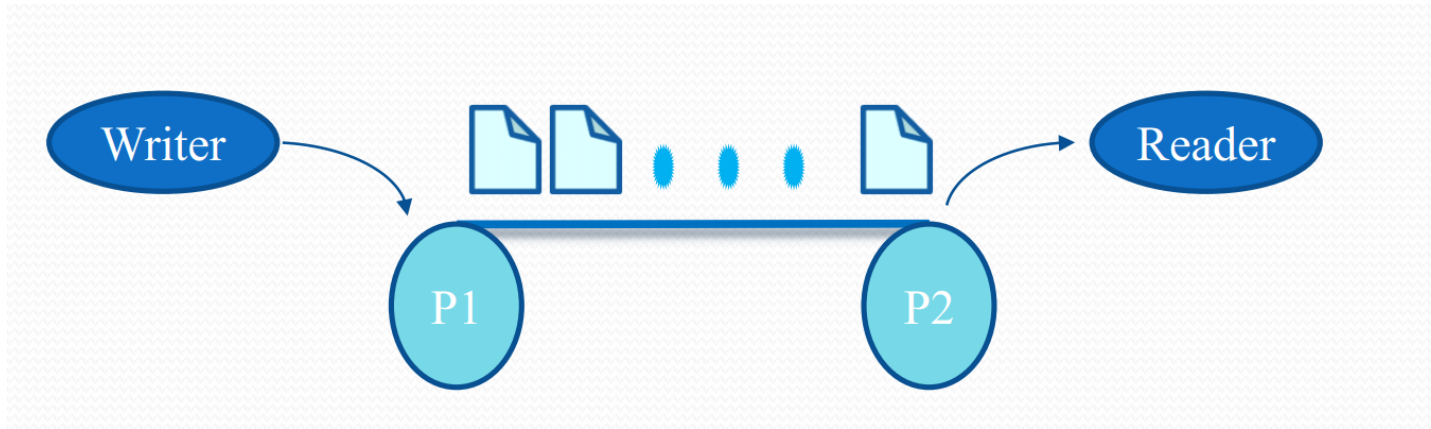
- Signal is a mechanism to communication between multiple processes by way of signaling. This means a source process will send a signal (recognized by number) and the destination process will handle it accordingly.
- Signals are software interrupts that are delivered to a process by the kernel.
- The signals are identified by integers.
- The signals in Linux are defined in /usr/include/signal.h
- Signal can be specified with a number or a name, usually signal names start with SIG. The available signals can be checked with the command `kill -l` (l for Listing signal names), as follows

```
bostney@ubuntu:~/Demo_CBL/Signal$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS     8) SIGFPE     9) SIGKILL    10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF    28) SIGWINCH   29) SIGIO      30) SIGPWR
31) SIGSYS     34) SIGRTMIN   35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

- 
- ▶ The actions performed for the signals are as follows:
 - Default Action
 - Handle the signal: Signal handling can be done in either of the two ways: through system calls `signal()` and `sigaction()`.
 - Ignore the signal: `SIG_IGN` function of OS
 - ▶ Note: the signals which can't be either ignored or handled/caught are `SIGSTOP` and `SIGKILL`.

- ▶ While a signal arrives on a single threaded process, the thread complete the current instruction, jump to the signal handler and return when it finish.
- ▶ On a multithreaded application – the signal handler execute in one of the thread contexts. We can't predict the thread that will be chosen to run the signal handler
- ▶ Inside the kernel, each thread has a `task_struct` object defines in `sched.h`:
- ▶ All the signals fields are stored per thread. Actually , there is no structure for the process , all the threads on the same process points to the same memory and files tables so the kernel need to choose a thread to deliver the signal to

2. Pipes



- Pipe is a communication medium between two or more related or interrelated processes. We can think of the pipe as a special file that can store a limited amount of data in a first in, first out (FIFO) manner.
 - ✓ Pipe in (pipe write).
 - ✓ Pipe out (pipe read).
 - ✓ Stream data with FIFO mechanism.
 - ✓ Fixed in size and is usually at least 4,096 bytes.
- There are two types of pipe:
 - Name pipe.
 - Unnamed pipe.

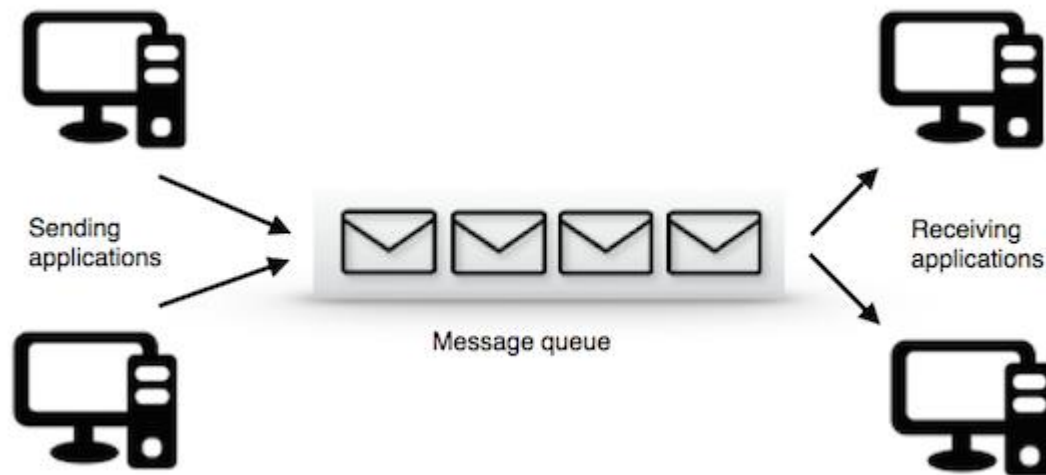
- **Unnamed pipe:**
 - Used only with related processes(child and it's parent process).
 - The pipe exists only as long as the processes using it are alive
- **Named pipe:**
 - Actually exist as directory entries
 - Have file access permissions
 - Can be used by unrelated processes

► Pipe Interfaces:

- `int pipe(int fildes[2]):` create pipe for unnamed pipe, Descriptor `pipedes[0]` is for reading and `pipedes[1]` is for writing
- `Mkfifo(const char *filename, mode_t mode):` create pipe for name pipe
- `int close(int fd):` call closing already opened file descriptor
- `ssize_t read(int fd, void *buf, size_t count):` Read from pipe
- `ssize_t write(int fd, void *buf, size_t count):` Write into pipe

3. Message queues

- ▶ Message Queues are synonymous to mailboxes. One process writes a message packet on the message queue and exits. Another process can access the message packet from the same message queue at a latter point in time. The advantage of message queues over pipes/FIFOs are that the sender (or writer) processes do not have to wait for the receiver (or reader) processes to connect. Think of communication using pipes as similar to two people communicating over phone, while message queues are similar to two people communicating using mail or other messaging services.



► There are two standard specifications for message queues.

- **SysV message queues.**

The AT&T SysV message queues support message channeling. Each message packet sent by senders carry a message number.

- **POSIX message queues.**

The POSIX message queues support message priorities. Each message packet sent by the senders carry a priority number along with the message payload.

- Linux support both of the above standards for message queues.

SysV message queues.

- ▶ System V message queues are identified using keys obtained with the `ftok` function call
- ▶ There are three system wide limits regarding the message queues. These are, `MSGMNI` is maximum number of queues in the system, `MSGMAX` is maximum size of a message in bytes and `MSGMNB`, which is the maximum size of a message queue. We can see these limits with the `ipcs -l` command

```
kien@ubuntu:~/workspace/Demo/IPC/MsgQueue/SysV$ ipcs -l  
  
----- Messages Limits -----  
max queues system wide = 32000  
max size of message (bytes) = 8192  
default max size of queue (bytes) = 16384
```

► System V Message Queue Interfaces:

- `int msgget()`: Create a message queue or connect to an already existing message queue
- `msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`: Write into message queue
- `msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)`: Read from the message queue
- `msgctl()`: Perform control operations on the message queue (This call would return the number of bytes actually received in mtext array on success and -1 in case of failure)

POSIX message queues

- ▶ POSIX message queues are identified using name strings.
- ▶ POSIX queues are named as string starting with a forward slash (/) followed by one or more characters, none of which is a slash and ending with the null character. Any process knowing the queue name and having appropriate permissions can send or receive messages from the queue and also do other operations on it.
- ▶ Programs using POSIX message queues on Linux must link with the real-time library, `librt` using the compiler option `-lrt`

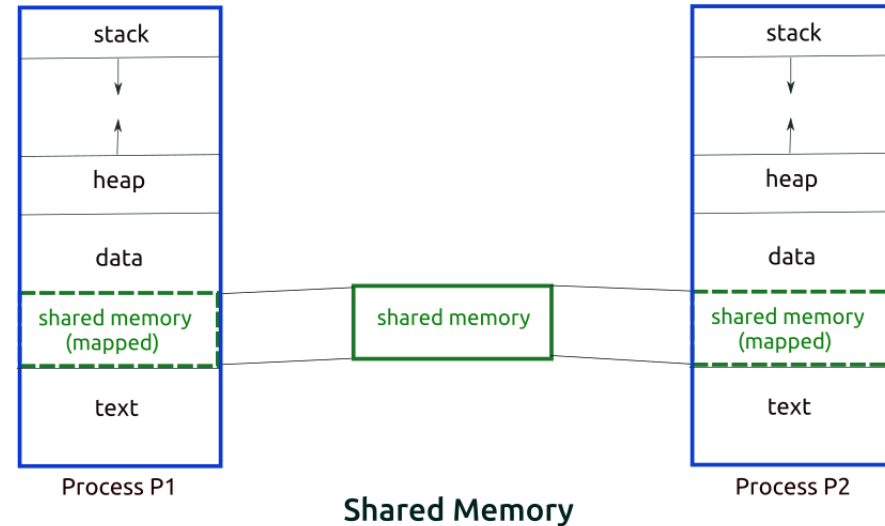
► POSIX Message Queue Interfaces:

- `mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr)`: Create a message queue or connect to an already existing message queue (`/dev/mqueue/`)
- `mq_send()`: Write into message queue
- `mq_receive()`: Read from the message queue
- `mq_unlink()`: to destroy the message queue
- `mq_getattr()`: gets the attribute structure (`struct mq_attr`)
- `mq_setattr()`: setting the attributes (`struct mq_attr`) of a queue

```
struct mq_attr {  
    long mq_flags;    /* Flags: 0 or O_NONBLOCK */  
    long mq_maxmsg;   /* Max. # of messages on queue */  
    long mq_msgsize;  /* Max. message size (bytes) */  
    long mq_curmsgs;  /* # of messages currently in queue */  
};
```


4. Shared memory

- ▶ As the name implies, this IPC mechanism allows one process to share a region of memory in its address space with another. This allows two or more processes to communicate data more efficiently amongst themselves with minimal kernel intervention.
- ▶ This is the fastest method of inter process communication.
- ▶ There are two standard specifications for Shared memory:
 - **SysV Shared memory.**
 - **POSIX Shared memory:** a cleaner interface and are easier than SysV.



POSIX share memory

- ▶ Programs using POSIX message queues on Linux must link with the real-time library, `librt` using the compiler option `-lrt`.
- ▶ POSIX shared memory files are provided from a `tmpfs` filesystem mounted at `/dev/shm`.
- ▶ **Share memory Interfaces:**
 - `int shm_open (const char *name, int oflag, mode_t mode):` open or creat a POSIX shared memory object
 - `int shm_unlink (const char *name):` `shm_unlink` removes the previously created POSIX shared memory object
 - `int ftruncate (int fd, off_t length):` When a POSIX shared memory is created, it is of size zero bytes. Using `ftruncate`, we can make the POSIX shared memory object of size `length` bytes.
 - `void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset):`
 - `int munmap (void *addr, size_t length):`

Sys V share memory

► Share memory Interfaces:

- To use a System V IPC mechanism, we need a System V IPC key. The `ftok` function, which does the job
 - `key_t ftok (const char *pathname, int proj_id)`: is use to generate a unique key.
 - `int shmget (key_t key, size_t size,int shmflg)`: upon successful completion, `shmget()` returns an identifier for the shared memory segment.
 - `void *shmat (int shmid, const void *shmaddr, int shmflg)`: Before you can use a shared memory segment, you have to attach yourself
 - `int shmdt (const void *shm_addr)`:When you're done with the shared memory segment, your program should detach itself
 - `void shmctl (void *addr, size_t length, int prot, int flags, int fd, off_t offset)`: when you detach from shared memory,it is not destroyed. So, to destroy

IPC commands

- ▶ `ipcs` will provide all ipc mechanism stats.
- ▶ `ipcs -q`: Show only message queues
- ▶ `ipcs -s`: Show only semaphores
- ▶ `ipcs -m`: Show only shared memory
- ▶ `ipcs --help`: Additional arguments
- ▶ `ipcrm --all=msg`: Remove all message queue
- ▶ The `ipcrm` command can be used to remove an IPC object from the kernel:

`ipcrm <msg | sem | shm> <IPC ID>`

Q & A

Thank you