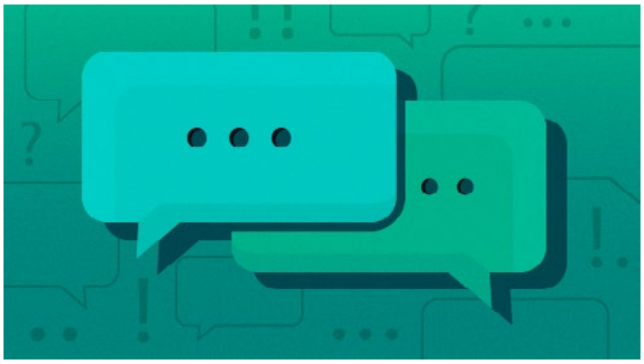
Using pipes and message queues

Learn how processes synchronize with each other in Linux.

THIS SECTION TURNS TO PIPES, which are channels that connect processes for communication. A channel has a *write end* for writing bytes, and a *read end* for reading these bytes in FIFO (first in, first out) order. In typical use, one process writes to the channel, and a different process reads from this same channel. The bytes themselves might represent anything: numbers, employee records, digital movies, and so on.



Pipes come in two flavors, named and unnamed, and can be used either interactively from the command line or within programs; examples are forthcoming. This section also looks at memory queues, which have fallen out of fashion—but undeservedly so.

The code examples in the first section acknowledged the threat of race conditions (either file-based or memory-based) in IPC that uses shared storage. The question naturally arises about safe concurrency for the channel-based IPC, which will be covered in this section. The code examples for pipes and memory queues use APIs with the POSIX stamp of approval, and a core goal of the POSIX standards is thread-safety.

Consider the man pages for the **mq_open** [1] function, which belongs to the memory queue API. These pages include a section on Attributes [2] with this small table:

Interface	Attribute	Value
mq_open()	Thread safety	MT-Safe

The value **MT-Safe** (with **MT** for multi-threaded) means that the **mq_open** function is thread-safe, which in turn implies process-safe: A process executes in precisely the sense that one of its threads executes, and if a race condition cannot arise among threads in the *same* process, such a condition cannot arise among threads in different processes. The **MT-Safe** attribute assures that a race condition does not arise in invocations of **mq_open**. In general, channel-based IPC is concurrent-safe, although a cautionary note is raised in the examples that follow.

Unnamed pipes

Let's start with a contrived command line example that shows how unnamed pipes work. On all modern systems, the vertical bar **|** represents an unnamed pipe at the command line. Assume **%** is the command line prompt, and consider this command:

```
% sleep 5 | echo "Hello, world!" ## writer to the left of |,
  reader to the right
```

The *sleep* and *echo* utilities execute as separate processes, and the unnamed pipe allows them to communicate. However, the example is contrived in that no communication occurs. The greeting *Hello, world!* appears on the screen; then, after about five seconds, the command line prompt returns, indicating that both the *sleep* and *echo* processes have exited. What's going on?

In the vertical-bar syntax from the command line, the process to the left (*sleep*) is the writer, and the process to the right (*echo*) is the reader. By default, the reader blocks until there are bytes to read from the channel, and the writer—after writing its bytes—finishes up by sending an end-of-stream marker. (Even if the writer terminates prematurely, an end-of-stream marker is sent to the reader.) The unnamed pipe persists until both the writer and the reader terminate.

In the contrived example, the *sleep* process does not write any bytes to the channel but does terminate after about five seconds, which sends an end-of-stream marker to the channel. In the meantime, the *echo* process immediately writes the greeting to the standard output (the screen) because this process does not read any bytes from the channel, so it does no waiting. Once the *sleep* and *echo* processes terminate, the unnamed pipe—not used at all for communication—goes away and the command line prompt returns.

Here is a more useful example using two unnamed pipes. Suppose that the file *test.dat* looks like this:

```
this
is
the
way
the
world
ends
```

The command:

```
% cat test.dat | sort | uniq
```

pipes the output from the *cat* (concatenate) process into the *sort* process to produce sorted output, and then pipes the sorted output into the *uniq* process to eliminate duplicate records (in this case, the two occurrences of **the** reduce to one):

```
ends
is
the
this
way
world
```

The scene now is set for a program with two processes that communicate through an unnamed pipe (see Example 1).

The *pipeUN* program above uses the system function **fork** to create a process. Although the program has but a single source file, multi-processing occurs during (successful) execution. Here are the particulars in a quick review of how the library function **fork** works:

- The **fork** function, called in the *parent* process, returns **-1** to the parent in case of failure. In the *pipeUN* example, the call is:

```
pid_t cpid = fork(); /* called in parent */
```

The returned value is stored, in this example, in the variable **cpid** of integer type **pid_t**. (Every process has its own *process ID*, a non-negative integer that identifies the process.) Forking a new process could fail for several reasons, including a full *process table*, a structure that the system maintains to track processes. Zombie processes, clarified shortly, can cause a process table to fill if these are not harvested.

Example 1. Two processes communicating through an unnamed pipe.

```c
#include <sys/wait.h> /* wait */
#include <stdio.h>
#include <stdlib.h>   /* exit functions */
#include <unistd.h>   /* read, write, pipe, _exit */
#include <string.h>

#define ReadEnd  0
#define WriteEnd 1

void report_and_exit(const char* msg) {
  perror(msg);
  exit(-1);    /** failure **/
}

int main() {
  int pipeFDs[2]; /* two file descriptors */
  char buf;       /* 1-byte buffer */
  const char* msg = "Nature's first green is gold\n"; /* bytes to write */

  if (pipe(pipeFDs) < 0) report_and_exit("pipeFD");
  pid_t cpid = fork();                                /* fork a child process */
  if (cpid < 0) report_and_exit("fork");              /* check for failure */

  if (0 == cpid) {    /*** child ***/                 /* child process */
    close(pipeFDs[WriteEnd]);                         /* child reads, doesn't write */

    while (read(pipeFDs[ReadEnd], &buf, 1) > 0)       /* read until end of byte stream */
      write(STDOUT_FILENO, &buf, sizeof(buf));        /* echo to the standard output */

    close(pipeFDs[ReadEnd]);                          /* close the ReadEnd: all done */
    _exit(0);                                         /* exit and notify parent at once  */
  }
  else {              /*** parent ***/
    close(pipeFDs[ReadEnd]);                          /* parent writes, doesn't read */

    write(pipeFDs[WriteEnd], msg, strlen(msg));       /* write the bytes to the pipe */
    close(pipeFDs[WriteEnd]);                         /* done writing: generate eof */

    wait(NULL);                                       /* wait for child to exit */
    exit(0);                                          /* exit normally */
  }
  return 0;
}
```

- If the **fork** call succeeds, it thereby spawns (creates) a new child process, returning one value to the parent but a different value to the child. Both the parent and the child process execute the *same* code that follows the call to **fork**. (The child inherits copies of all the variables declared so far in the parent.) In particular, a successful call to **fork** returns:
  - Zero to the child process
  - The child's process ID to the parent
- An *if/else* or equivalent construct typically is used after a successful **fork** call to segregate code meant for the parent from code meant for the child. In this example, the construct is:

```
if (0 == cpid) {    /*** child ***/
...
}
else {              /*** parent ***/
...
}
```

If forking a child succeeds, the *pipeUN* program proceeds as follows. There is an integer array:

```
int pipeFDs[2]; /* two file descriptors */
```

to hold two file descriptors, one for writing to the pipe and another for reading from the pipe. (The array element **pipeFDs[0]** is the file descriptor for the read end, and the array element **pipeFDs[1]** is the file descriptor for the write end.) A successful call to the system **pipe** function, made immediately before the call to **fork**, populates the array with the two file descriptors:

```
if (pipe(pipeFDs) < 0) report_and_exit("pipeFD");
```

The parent and the child now have copies of both file descriptors, but the *separation of concerns* pattern means that each process requires exactly one of the descriptors. In this example, the parent does the writing and the child does the reading, although the roles could be reversed. The first statement in the child *if*-clause code, therefore, closes the pipe's write end:

```
close(pipeFDs[WriteEnd]); /* called in child code */
```

and the first statement in the parent *else*-clause code closes the pipe's read end:

```
close(pipeFDs[ReadEnd]);  /* called in parent code */
```

The parent then writes some bytes (ASCII codes) to the unnamed pipe, and the child reads these and echoes them to the standard output.

One more aspect of the program needs clarification: the call to the **wait** function in the parent code. Once spawned, a child process is largely independent of its parent, as even the short *pipeUN* program illustrates. The child can execute arbitrary code that may have nothing to do with the parent. However, the system does notify the parent through a signal—if and when the child terminates.

What if the parent terminates before the child? In this case, unless precautions are taken, the child becomes and remains a *zombie* process with an entry in the process table. The precautions are of two broad types. One precaution is to have the parent notify the system that the parent has no interest in the child's termination:

```
signal(SIGCHLD, SIG_IGN); /* in parent: ignore notification */
```

A second approach is to have the parent execute a **wait** on the child's termination, thereby ensuring that the parent outlives the child. This second approach is used in the *pipeUN* program, where the parent code has this call:

```
wait(NULL); /* called in parent */
```

This call to **wait** means *wait* until the termination of any child occurs, and in the *pipeUN* program, there is only one child process. (The **NULL** argument could be replaced with the address of an integer variable to hold the child's exit status.) There is a more flexible **waitpid** function for fine-grained control, e.g., for specifying a particular child process among several.

The *pipeUN* program takes another precaution. When the parent is done waiting, the parent terminates with the call to the regular **exit** function. By contrast, the child terminates with a call to the **_exit** variant, which fast-tracks notification of termination. In effect, the child is telling the system to notify the parent ASAP that the child has terminated.

If two processes write to the same unnamed pipe, can the bytes be interleaved? For example, if process P1 writes:

```
foo bar
```

to a pipe and process P2 concurrently writes:

```
baz baz
```

to the same pipe, it seems that the pipe contents might be something arbitrary, such as:

```
baz foo baz bar
```

The POSIX standard ensures that writes are not interleaved so long as no write exceeds **PIPE_BUF** bytes. On Linux systems, **PIPE_BUF** is 4,096 bytes in size. My preference with pipes is to have a single writer and a single reader, thereby sidestepping the issue.

6.4.4 Shared Memory

Basic Concepts

Shared memory can best be described as the mapping of an area (segment) of memory that will be mapped and shared by more than one process. This is by far the fastest form of IPC, because there is no intermediation (i.e. a pipe, a message queue, etc). Instead, information is mapped directly from a memory segment, and into the addressing space of the calling process. A segment can be created by one process, and subsequently written to and read from by any number of processes.

Internal and User Data Structures

Let's briefly look at data structures maintained by the kernel for shared memory segments.

Kernel `shmid_ds` structure As with message queues and semaphore sets, the kernel maintains a special internal data structure for each shared memory segment which exists within its addressing space. This structure is of type `shmid_ds`, and is defined in `linux/shm.h` as follows:

```
        /* One shmid data structure for each shared memory segment in the syste
        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 */

                                                 /* the following are private *

                unsigned short   shm_npages;     /* size of segment (pages) */
                unsigned long   *shm_pages;      /* array of ptrs to frames ->
                struct vm_area_struct *attaches; /* descriptors for attaches */
        };
```

Operations on this structure are performed by a special system call, and should not be tinkered with directly. Here are descriptions of the more pertinent fields:

`shm_perm`

> This is an instance of the `ipc_perm` structure, which is defined for us in `linux/ipc.h`. This holds the permission information for the segment, including the access permissions, and information about the creator of the segment (uid, etc).

`shm_segsz`

> Size of the segment (measured in bytes).

`shm_atime`

> Time the last process attached the segment.

`shm_dtime`

> Time the last process detached the segment.

`shm_ctime`

Time of the last change to this structure (mode change, etc).

`shm_cpid`

The PID of the creating process.

`shm_lpid`

The PID of the last process to operate on the segment.

`shm_nattch`

Number of processes currently attached to the segment.

SYSTEM CALL: shmget()

In order to create a new message queue, or access an existing queue, the `shmget()` system call is used.

```
SYSTEM CALL: shmget();

PROTOTYPE: int shmget ( key_t key, int size, int shmflg );
  RETURNS: shared memory segment identifier on success
           -1 on error: errno = EINVAL (Invalid segment size specified)
                                EEXIST (Segment exists, cannot create)
                                EIDRM (Segment is marked for deletion, or was r
                                ENOENT (Segment does not exist)
                                EACCES (Permission denied)
                                ENOMEM (Not enough memory to create segment)
NOTES:
```

This particular call should almost seem like old news at this point. It is strikingly similar to the corresponding `get` calls for message queues and semaphore sets.

The first argument to `shmget()` is the key value (in our case returned by a call to `ftok()`). This key value is then compared to existing key values that exist within the kernel for other shared memory segments. At that point, the open or access operation is dependent upon the contents of the `shmflg` argument.

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.

If `IPC_CREAT` is used alone, `shmget()` either returns the segment identifier for a newly created segment, or returns the identifier for a segment which exists with the same key value. If `IPC_EXCL` is used along with `IPC_CREAT`, then either a new segment is created, or if the segment exists, the call fails with -1. `IPC_EXCL` is useless by itself, but when combined with `IPC_CREAT`, it can be used as a facility to guarantee that no existing segment is opened for access.

Once again, an optional octal mode may be OR'd into the mask.

Let's create a wrapper function for locating or creating a shared memory segment :

```
int open_segment( key_t keyval, int segsize )
{
        int     shmid;

        if((shmid = shmget( keyval, segsize, IPC_CREAT | 0660 )) == -1)
        {
                return(-1);
        }

        return(shmid);
}
```

Note the use of the explicit permissions of `0660`. This small function either returns a shared memory segment identifier (`int`), or -1 on error. The key value and requested segment size (in bytes) are passed as arguments.

Once a process has a valid IPC identifier for a given segment, the next step is for the process to `attach` or map the segment into its own addressing space.

SYSTEM CALL: shmat()

```
  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: errno = EINVAL (Invalid IPC ID value or attach addres
                                  ENOMEM (Not enough memory to attach segment)
                                  EACCES (Permission denied)
  NOTES:
```

If the addr argument is zero (0), the kernel tries to find an unmapped region. This is the recommended method. An address can be specified, but is typically only used to facilitate proprietary hardware or to resolve conflicts with other apps. The SHM_RND flag can be OR'd into the flag argument to force a passed address to be page aligned (rounds down to the nearest page size).

In addition, if the SHM_RDONLY flag is OR'd in with the flag argument, then the shared memory segment will be mapped in, but marked as readonly.

This call is perhaps the simplest to use. Consider this wrapper function, which is passed a valid IPC identifier for a segment, and returns the address that the segment was attached to:

```
char *attach_segment( int shmid )
{
        return(shmat(shmid, 0, 0));
}
```

Once a segment has been properly attached, and a process has a pointer to the start of that segment, reading and writing to the segment become as easy as simply referencing or dereferencing the pointer! Be careful not to lose the value of the original pointer! If this happens, you will have no way of accessing the base (start) of the segment.

SYSTEM CALL: shmctl()

```
  SYSTEM CALL: shmctl();
  PROTOTYPE: int shmctl ( int shmqid, int cmd, struct shmid_ds *buf );
    RETURNS: 0 on success
```

```
             -1 on error: errno = EACCES (No read permission and cmd is IPC_STAT)
                                  EFAULT (Address pointed to by buf is invalid wi
                                          IPC_STAT commands)
                                  EIDRM  (Segment was removed during retrieval)
                                  EINVAL (shmqid invalid)
                                  EPERM  (IPC_SET or IPC_RMID command was issued,
                                          calling process does not have write (al
                                          access to the segment)
      NOTES:
```

This particular call is modeled directly after the *msgctl* call for message queues. In light of this fact, it won't be discussed in too much detail. Valid command values are:

IPC_STAT

Retrieves the shmid_ds structure for a segment, and stores it in the address of the buf argument

IPC_SET

Sets the value of the ipc_perm member of the shmid_ds structure for a segment. Takes the values from the buf argument.

IPC_RMID

Marks a segment for removal.

The IPC_RMID command doesn't actually remove a segment from the kernel. Rather, it `marks` the segment for removal. The actual removal itself occurs when the last process currently attached to the segment has properly detached it. Of course, if no processes are currently attached to the segment, the removal seems immediate.

To properly detach a shared memory segment, a process calls the *shmdt* system call.

SYSTEM CALL: shmdt()

```
  SYSTEM CALL: shmdt();

  PROTOTYPE: int shmdt ( char *shmaddr );
    RETURNS: -1 on error: errno = EINVAL (Invalid attach address passed)
```

After a shared memory segment is no longer needed by a process, it should be detached by calling this system call. As mentioned earlier, this is not the same as removing the segment from the kernel! After a detach is successful, the shm_nattch member of the associates shmid_ds structure is decremented by one. When this value reaches zero (0), the kernel will physically remove the segment.

shmtool: An interactive shared memory manipulator

Background Our final example of System V IPC objects will be `shmtool`, which is a command line tool for creating, reading, writing, and deleting shared memory segments. Once again, like the previous examples, the segment is created during any operation, if it did not previously exist.

Command Line Syntax

Writing strings to the segment

```
shmtool w "text"
```

Retrieving strings from the segment

```
shmtool r
```

Changing the Permissions (mode)

```
shmtool m (mode)
```

Deleting the segment

```
shmtool d
```

Examples

```
shmtool  w   test
shmtool  w   "This is a test"
shmtool  r
shmtool  d
shmtool  m   660
```

The Source

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SEGSIZE 100

main(int argc, char *argv[])
{
        key_t key;
        int   shmid, cntr;
        char  *segptr;

        if(argc == 1)
                usage();

        /* Create unique key via call to ftok() */
        key = ftok(".", ’S’);

        /* Open the shared memory segment - create if necessary */
        if((shmid = shmget(key, SEGSIZE, IPC_CREAT|IPC_EXCL|0666)) == -1)
        {
                printf("Shared memory segment exists - opening as client\n");

                /* Segment probably already exists - try as a client */
                if((shmid = shmget(key, SEGSIZE, 0)) == -1)
                {
                        perror("shmget");
                        exit(1);
                }
        }
        else
        {
                printf("Creating new shared memory segment\n");
        }

        /* Attach (map) the shared memory segment into the current process */
        if((segptr = shmat(shmid, 0, 0)) == -1)
        {
                perror("shmat");
                exit(1);
        }

        switch(tolower(argv[1][0]))
        {
                case ’w’: writeshm(shmid, segptr, argv[2]);
                          break;
                case ’r’: readshm(shmid, segptr);
                          break;
                case ’d’: removeshm(shmid);
                          break;
                case ’m’: changemode(shmid, argv[2]);
                          break;
                 default: usage();

        }
}

writeshm(int shmid, char *segptr, char *text)
{
        strcpy(segptr, text);
        printf("Done...\n");
}

readshm(int shmid, char *segptr)
{
        printf("segptr: %s\n", segptr);
}

removeshm(int shmid)
{
        shmctl(shmid, IPC_RMID, 0);
        printf("Shared memory segment marked for deletion\n");
}

changemode(int shmid, char *mode)
{
        struct shmid_ds myshmds;

        /* Get current values for internal data structure */
        shmctl(shmid, IPC_STAT, &myshmds);

        /* Display old permissions */
        printf("Old permissions were: %o\n", myshmds.shm_perm.mode);

        /* Convert and load the mode */
        sscanf(mode, "%o", &myshmds.shm_perm.mode);
        /* Update the mode */
        shmctl(shmid, IPC_SET, &myshmds);

        printf("New permissions are : %o\n", myshmds.shm_perm.mode);
}

usage()
{
        fprintf(stderr, "shmtool - A utility for tinkering with shared memory\
        fprintf(stderr, "\nUSAGE:  shmtool (w)rite <text>\n");
        fprintf(stderr, "                (r)ead\n");
        fprintf(stderr, "                (d)elete\n");
        fprintf(stderr, "                (m)ode change <octal mode>\n");
        exit(1);
}
```