Opensource.com

A guide to inter-process communication in Linux

Learn how processes synchronize with each other in Linux.

By Marty Kalin

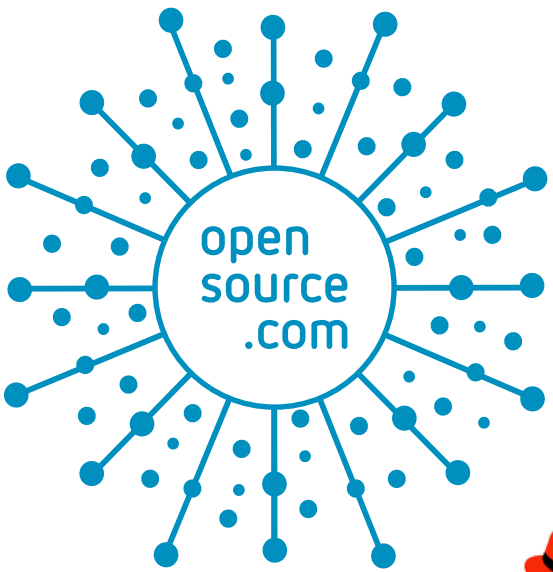
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

MARTY KALIN

I'M AN ACADEMIC IN COMPUTER SCIENCE (College of Computing and Digital Media, DePaul University) with wide experience in software development, mostly in production planning and scheduling (steel industry) and product configuration (truck and bus manufacturing). Details on books and other publications are available at:

Marty Kalin's hompage

FOLLOW MARTY KALIN

Twitter: @kalin_martin

INTRODUCTION

CHAPTERS

GET INVOLVED | ADDITIONAL RESOURCES

Introduction

THIS GUIDE IS ABOUT INTERPROCESS COMMUNICATION (IPC) in Linux. The guide uses code examples in C to clarify the following IPC mechanisms:

- Shared files
- Shared memory (with semaphores)
- Pipes (named and unnamed)
- Message queues
- Sockets
- Signals

I'll introduce you to some core concepts before moving on to the first two of these mechanisms: shared files and shared memory.

Core concepts

A process is a program in execution, and each process has its own address space, which comprises the memory locations that the process is allowed to access. A process has one or more *threads* of execution, which are sequences of executable instructions: a *single-threaded* process has just one thread, whereas a *multi-threaded* process has more than one thread. *Threads* within a process share various resources, in particular, address space. Accordingly, threads within a process can communicate straightforwardly through shared memory, although some modern languages (e.g., Go) encourage a more disciplined approach such as the use of thread-safe channels. Of interest here is that different processes, by default, do *not* share memory.

There are various ways to launch processes that then communicate, and two ways dominate in the examples that follow:

- A terminal is used to start one process, and perhaps a different terminal is used to start another.
- The system function **fork** is called within one process (the parent) to spawn another process (the child).

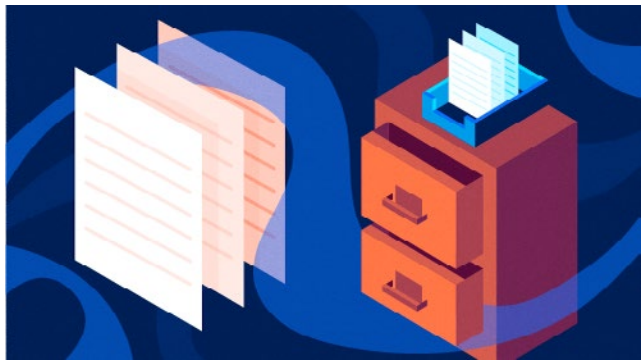
The first examples take the terminal approach. The code examples [1] are available in a ZIP file on my website.

Links

[1] http://condor.depaul.edu/mkalin

Shared storage

Learn how processes synchronize with each other in Linux.



PROGRAMMERS ARE ALL TOO FAMILIAR with file access, including the many pitfalls (non-existent files, bad file permissions, and so on) that beset the use of files in programs. Nonetheless, shared files may be the most basic IPC mechanism. Consider the relatively simple case in which one process (*producer*) creates and writes to a file, and another process (*consumer*) reads from this same file:

```
          writes  +-----------+  reads
producer-------->| disk file |<-------consumer
                 +-----------+
```

The obvious challenge in using this IPC mechanism is that a *race condition* might arise: the producer and the consumer might access the file at exactly the same time, thereby making the outcome indeterminate. To avoid a race condition, the file must be locked in a way that prevents a conflict between a *write* operation and any another operation, whether a *read* or a *write*. The locking API in the standard system library can be summarized as follows:

- A producer should gain an exclusive lock on the file before writing to the file. An *exclusive* lock can be held by one process at most, which rules out a race condition because no other process can access the file until the lock is released.
- A consumer should gain at least a shared lock on the file before reading from the file. Multiple *readers* can hold a *shared* lock at the same time, but no *writer* can access a file when even a single *reader* holds a shared lock.

A shared lock promotes efficiency. If one process is just reading a file and not

Example 1. The *producer* program

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FileName "data.dat"
#define DataString "Now is the winter of our discontent\nMade glorious summer by
#this sun of York\n"

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

int main() {
  struct flock lock;
  lock.l_type = F_WRLCK;    /* read/write (exclusive versus shared) lock */
  lock.l_whence = SEEK_SET; /* base for seek offsets */
  lock.l_start = 0;         /* 1st byte in file */
  lock.l_len = 0;           /* 0 here means 'until EOF' */
  lock.l_pid = getpid();    /* process id */

  int fd; /* file descriptor to identify a file within a process */
  if ((fd = open(FileName, O_RDWR | O_CREAT, 0666)) < 0)  /* -1 signals an error */
    report_and_exit("open failed...");

  if (fcntl(fd, F_SETLK, &lock) < 0) /** F_SETLK doesn't block, F_SETLKW does **/
    report_and_exit("fcntl failed to get lock...");
  else {
    write(fd, DataString, strlen(DataString)); /* populate data file */
    fprintf(stderr, "Process %d has written to data file...\n", lock.l_pid);
  }

  /* Now release the lock explicitly. */
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0)
    report_and_exit("explicit unlocking failed...");

  close(fd); /* close the file: would unlock if needed */
  return 0;  /* terminating the process would unlock as well */
}
```

changing its contents, there is no reason to prevent other processes from doing the same. Writing, however, clearly demands exclusive access to a file.

The standard I/O library includes a utility function named **fcntl** that can be used to inspect and manipulate both exclusive and shared locks on a file. The function works through a *file descriptor*, a non-negative integer value that, within a process, identifies a file. (Different file descriptors in different processes may identify the same physical file.) For file locking, Linux provides the library function **flock**, which is a thin wrapper around **fcntl**. The first example uses the **fcntl** function to expose API details (see Example 1).

The main steps in the *producer* program above can be summarized as follows:

- The program declares a variable of type **struct flock**, which represents a lock, and initializes the structure's five fields. The first initialization:

```
lock.l_type = F_WRLCK; /* exclusive lock */
```

  makes the lock an exclusive (*read-write*) rather than a shared (*read-only*) lock. If the *producer* gains the lock, then no other process will be able to write or read the file until the *producer* releases the lock, either explicitly with the appropriate call to **fcntl** or implicitly by closing the file. (When the process terminates, any opened files would be closed automatically, thereby releasing the lock.)
- The program then initializes the remaining fields. The chief effect is that the *entire* file is to be locked. However, the locking API allows only designated bytes to be locked. For example, if the file contains multiple text records, then a single record (or even part of a record) could be locked and the rest left unlocked.
- The first call to **fcntl**:

```
if (fcntl(fd, F_SETLK, &lock) < 0)
```

  tries to lock the file exclusively, checking whether the call succeeded. In general, the **fcntl** function returns **-1** (hence, less than zero) to indicate failure. The second argument **F_SETLK** means that the call to **fcntl** does *not* block: the function returns immediately, either granting the lock or indicating failure. If the flag **F_SETLKW** (the **W** at the end is for *wait*) were used instead, the call to **fcntl** would block until gaining the lock was possible. In the calls to **fcntl**, the first argument **fd** is the file descriptor, the second argument specifies the action to be taken (in this case, **F_SETLK** for setting the lock), and the third argument is the address of the lock structure (in this case, **&lock**).
- If the *producer* gains the lock, the program writes two text records to the file.
- After writing to the file, the *producer* changes the lock structure's **l_type** field to the *unlock* value:

```
lock.l_type = F_UNLCK;
```

  and calls **fcntl** to perform the unlocking operation. The program finishes up by closing the file and exiting (see Example 2).

Example 2. The *consumer* program

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define FileName "data.dat"

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

int main() {
  struct flock lock;
  lock.l_type = F_WRLCK;    /* read/write (exclusive) lock */
  lock.l_whence = SEEK_SET; /* base for seek offsets */
  lock.l_start = 0;         /* 1st byte in file */
  lock.l_len = 0;           /* 0 here means 'until EOF' */
  lock.l_pid = getpid();    /* process id */

  int fd; /* file descriptor to identify a file within a process */
  if ((fd = open(FileName, O_RDONLY)) < 0)  /* -1 signals an error */
    report_and_exit("open to read failed...");

  /* If the file is write-locked, we can't continue. */
  fcntl(fd, F_GETLK, &lock); /* sets lock.l_type to F_UNLCK if no write lock */
  if (lock.l_type != F_UNLCK)
    report_and_exit("file is still write locked...");

  lock.l_type = F_RDLCK; /* prevents any writing during the reading */
  if (fcntl(fd, F_SETLK, &lock) < 0)
    report_and_exit("can't get a read-only lock...");

  /* Read the bytes (they happen to be ASCII codes) one at a time. */
  int c; /* buffer for read bytes */
  while (read(fd, &c, 1) > 0)    /* 0 signals EOF */
    write(STDOUT_FILENO, &c, 1); /* write one byte to the standard output */

  /* Release the lock explicitly. */
  lock.l_type = F_UNLCK;
  if (fcntl(fd, F_SETLK, &lock) < 0)
    report_and_exit("explicit unlocking failed...");

  close(fd);
  return 0;
}
```

The *consumer* program is more complicated than necessary to highlight features of the locking API. In particular, the *consumer* program first checks whether the file is exclusively locked and only then tries to gain a shared lock. The relevant code is:

```
lock.l_type = F_WRLCK;
...
fcntl(fd, F_GETLK, &lock); /* sets lock.l_type to F_UNLCK if no
                              write lock */
if (lock.l_type != F_UNLCK)
  report_and_exit("file is still write locked...");
```

The **F_GETLK** operation specified in the **fcntl** call checks for a lock, in this case, an exclusive lock given as **F_WRLCK** in the first statement above. If the specified lock does not exist, then the **fcntl** call automatically changes the lock type field to **F_UNLCK** to indicate this fact. If the file is exclusively locked, the *consumer* terminates. (A more robust version of the program might have the *consumer* **sleep** a bit and try again several times.)

If the file is not currently locked, then the *consumer* tries to gain a shared (*read-only*) lock (**F_RDLCK**). To shorten the program, the **F_GETLK** call to **fcntl** could be dropped because the **F_RDLCK** call would fail if a *read-write* lock already were held by some other process. Recall that a *read-only* lock does prevent any other process from writing to the file, but allows other processes to read from the file. In short, a *shared* lock can be held by multiple processes. After gaining a shared lock, the *consumer* program reads the bytes one at a time from the file, prints the bytes to the standard output, releases the lock, closes the file, and terminates.

Here is the output from the two programs launched from the same terminal with **%** as the command line prompt:

```
% ./producer
Process 29255 has written to data file...

% ./consumer
Now is the winter of our discontent
Made glorious summer by this sun of York
```

In this first code example, the data shared through IPC is text: two lines from Shakespeare's play *Richard III*. Yet, the shared file's contents could be voluminous, arbitrary bytes (e.g., a digitized movie), which makes file sharing an impressively flexible IPC mechanism. The downside is that file access is relatively slow, whether the access involves reading or writing. As always, programming comes with tradeoffs. The next example has the upside of IPC through shared memory, rather than shared files, with a corresponding boost in performance.

Shared memory

Linux systems provide two separate APIs for shared memory: the legacy System V API and the more recent POSIX one. These APIs should never be mixed in a single application, however. A downside of the POSIX approach is that features are still in development and dependent upon the installed kernel version, which impacts code portability. For example, the POSIX API, by default, implements shared memory as a *memory-mapped file*: for a shared memory segment, the system maintains a *backing file* with corresponding contents. Shared memory under POSIX can be configured without a backing file, but this may impact portability. My example uses the POSIX API with a backing file, which combines the benefits of memory access (speed) and file storage (persistence).

The shared-memory example has two programs, named *memwriter* and *memreader*, and uses a *semaphore* to coordinate their access to the shared memory. Whenever shared memory comes into the picture with a *writer*, whether in multi-processing or multi-threading, so does the risk of a memory-based race condition; hence, the semaphore is used to coordinate (synchronize) access to the shared memory.

Example 3. Source code for the *memwriter* process

```
/** Compilation: gcc -o memwriter memwriter.c -lrt -lpthread **/
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include "shmem.h"

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

int main() {
  int fd = shm_open(BackingFile,      /* name from smem.h */
                    O_RDWR | O_CREAT, /* read/write, create if needed */
                    AccessPerms);     /* access permissions (0644) */
  if (fd < 0) report_and_exit("Can't open shared mem segment...");

  ftruncate(fd, ByteSize); /* get the bytes */
```

The *memwriter* program should be started first in its own terminal. The *memreader* program then can be started (within a dozen seconds) in its own terminal. The output from the *memreader* is:

```
This is the way the world ends...
```

Each source file has documentation at the top explaining the link flags to be included during compilation.

Let's start with a review of how semaphores work as a synchronization mechanism. A general semaphore also is called a *counting semaphore*, as it has a value (typically initialized to zero) that can be incremented. Consider a shop that rents bicycles, with a hundred of them in stock, with a program that clerks use to do the rentals. Every time a bike is rented, the semaphore is incremented by one; when a bike is returned, the semaphore is decremented by one. Rentals can continue until the value hits 100 but then must halt until at least one bike is returned, thereby decrementing the semaphore to 99.

A *binary semaphore* is a special case requiring only two values: 0 and 1. In this situation, a semaphore acts as a *mutex*: a mutual exclusion construct. The shared-memory example uses a semaphore as a mutex. When the semaphore's value is 0, the *memwriter* alone can access the shared memory. After writing, this process increments the semaphore's value, thereby allowing the *memreader* to read the shared memory (see Example 3).

Here's an overview of how the *memwriter* and *memreader* programs communicate through shared memory:

- The *memwriter* program, shown above, calls the **shm_open** function to get a file descriptor for the backing file that the system coordinates with the shared memory. At this point, no memory has been allocated. The subsequent call to the misleadingly named function **ftruncate**:

```
ftruncate(fd, ByteSize); /* get the bytes */
```

  allocates **ByteSize** bytes, in this case, a modest 512 bytes. The *memwriter* and *memreader* programs access the shared memory only, not the backing file. The system is responsible for synchronizing the shared memory and the backing file.

- The *memwriter* then calls the **mmap** function:

```
caddr_t memptr = mmap(NULL,       /* let system pick where to
                                     put segment */
                      ByteSize,   /* how many bytes */
                      PROT_READ | PROT_WRITE, /* access
                                                 protections */
                      MAP_SHARED, /* mapping visible to other
                                     processes */
                      fd,         /* file descriptor */
                      0);         /* offset: start at 1st byte */
```

  to get a pointer to the shared memory. (The *memreader* makes a similar call.) The pointer type **caddr_t** starts with a **c** for **calloc**, a system function that initializes dynamically allocated storage to zeroes. The *memwriter* uses the **memptr** for the later *write* operation, using the library **strcpy** (string copy) function.

Example 3. Source code for the *memwriter* process (continued)

```
  caddr_t memptr = mmap(NULL,       /* let system pick where to put segment */
                        ByteSize,   /* how many bytes */
                        PROT_READ | PROT_WRITE, /* access protections */
                        MAP_SHARED, /* mapping visible to other processes */
                        fd,         /* file descriptor */
                        0);         /* offset: start at 1st byte */
  if ((caddr_t) -1  == memptr) report_and_exit("Can't get segment...");

  fprintf(stderr, "shared mem address: %p [0..%d]\n", memptr, ByteSize - 1);
  fprintf(stderr, "backing file:       /dev/shm%s\n", BackingFile );

  /* semaphore code to lock the shared mem */
  sem_t* semptr = sem_open(SemaphoreName, /* name */
                           O_CREAT,       /* create the semaphore */
                           AccessPerms,   /* protection perms */
                           0);            /* initial value */
  if (semptr == (void*) -1) report_and_exit("sem_open");

  strcpy(memptr, MemContents); /* copy some ASCII bytes to the segment */

  /* increment the semaphore so that memreader can read */
  if (sem_post(semptr) < 0) report_and_exit("sem_post");

  sleep(12); /* give reader a chance */

  /* clean up */
  munmap(memptr, ByteSize); /* unmap the storage */
  close(fd);
  sem_close(semptr);
  shm_unlink(BackingFile); /* unlink from the backing file */
  return 0;
}
```

- At this point, the *memwriter* is ready for writing, but it first creates a semaphore to ensure exclusive access to the shared memory. A race condition would occur if the *memwriter* were writing while the *memreader* was reading. If the call to **sem_open** succeeds:

```
sem_t* semptr = sem_open(SemaphoreName, /* name */
                         O_CREAT,       /* create the semaphore */
                         AccessPerms,   /* protection perms */
                         0);            /* initial value */
```

  then the writing can proceed. The **SemaphoreName** (any unique non-empty name will do) identifies the semaphore in both the *memwriter* and the *memreader*. The initial value of zero gives the semaphore's creator, in this case, the *memwriter*, the right to proceed, in this case, to the *write* operation.
- After writing, the *memwriter* increments the semaphore value to 1:

```
if (sem_post(semptr) < 0) ..
```

  with a call to the **sem_post** function. Incrementing the semaphore releases the mutex lock and enables the *memreader* to perform its *read* operation. For good measure, the *memwriter* also unmaps the shared memory from the *memwriter* address space:

```
munmap(memptr, ByteSize); /* unmap the storage *
```

  This bars the *memwriter* from further access to the shared memory (see Example 4).

In both the *memwriter* and *memreader* programs, the shared-memory functions of main interest are **shm_open** and **mmap**: on success, the first call returns a file descriptor for the backing file, which the second call then uses to get a pointer to the shared memory segment. The calls to **shm_open** are similar in the two programs except that the *memwriter* program creates the shared memory, whereas the *memreader* only accesses this already created memory:

```
int fd = shm_open(BackingFile, O_RDWR | O_CREAT, AccessPerms);
         /* memwriter */
int fd = shm_open(BackingFile, O_RDWR, AccessPerms);
         /* memreader */
```

With a file descriptor in hand, the calls to **mmap** are the same:

```
caddr_t memptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_
  SHARED, fd, 0);
```

The first argument to **mmap** is **NULL**, which means that the system determines where to allocate the memory in virtual address space. It's possible (but tricky) to specify an address

Example 4. Source code for the *memreader* process

```
/** Compilation: gcc -o memreader memreader.c -lrt -lpthread **/
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>
#include <string.h>
#include "shmem.h"

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

int main() {
  int fd = shm_open(BackingFile, O_RDWR, AccessPerms);  /* empty to begin */
  if (fd < 0) report_and_exit("Can't get file descriptor...");

  /* get a pointer to memory */
  caddr_t memptr = mmap(NULL,       /* let system pick where to put segment */
                        ByteSize,   /* how many bytes */
                        PROT_READ | PROT_WRITE, /* access protections */
                        MAP_SHARED, /* mapping visible to other processes */
                        fd,         /* file descriptor */
                        0);         /* offset: start at 1st byte */
```

instead. The **MAP_SHARED** flag indicates that the allocated memory is shareable among processes, and the last argument (in this case, zero) means that the offset for the shared memory should be the first byte. The **size** argument specifies the number of bytes to be allocated (in this case, 512), and the protection argument indicates that the shared memory can be written and read.

When the *memwriter* program executes successfully, the system creates and maintains the backing file; on my system, the file is */dev/shm/shMemEx*, with *shMemEx* as my name (given in the header file *shmem.h*) for the shared storage. In the current version of the *memwriter* and *memreader* programs, the statement:

```
shm_unlink(BackingFile); /* removes backing file */
```

removes the backing file. If the **unlink** statement is omitted, then the backing file persists after the program terminates.

Example 4. Source code for the *memreader* process (continued)

```
  if ((caddr_t) -1 == memptr) report_and_exit("Can't access segment...");

  /* create a semaphore for mutual exclusion */
  sem_t* semptr = sem_open(SemaphoreName, /* name */
                           O_CREAT,       /* create the semaphore */
                           AccessPerms,   /* protection perms */
                           0);            /* initial value */
  if (semptr == (void*) -1) report_and_exit("sem_open");

  /* use semaphore as a mutex (lock) by waiting for writer to increment it */
  if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
    int i;
    for (i = 0; i < strlen(MemContents); i++)
      write(STDOUT_FILENO, memptr + i, 1); /* one byte at a time */
    sem_post(semptr);
  }

  /* cleanup */
  munmap(memptr, ByteSize);
  close(fd);
  sem_close(semptr);
  unlink(BackingFile);
  return 0;
}
```

The *memreader*, like the *memwriter*, accesses the semaphore through its name in a call to **sem_open**. But the *memreader* then goes into a wait state until the *memwriter* increments the semaphore, whose initial value is 0:

```
if (!sem_wait(semptr)) { /* wait until semaphore != 0 */
```

Once the wait is over, the *memreader* reads the ASCII bytes from the shared memory, cleans up, and terminates.

The shared-memory API includes operations explicitly to synchronize the shared memory segment and the backing file. These operations have been omitted from the example to reduce clutter and keep the focus on the memory-sharing and semaphore code.

The *memwriter* and *memreader* programs are likely to execute without inducing a race condition even if the semaphore code is removed: the *memwriter* creates the shared memory segment and writes immediately to it; the *memreader* cannot even access the shared memory until this has been created. However, best practice requires that shared-memory access is synchronized whenever a *write* operation is in the mix, and the semaphore API is important enough to be highlighted in a code example.

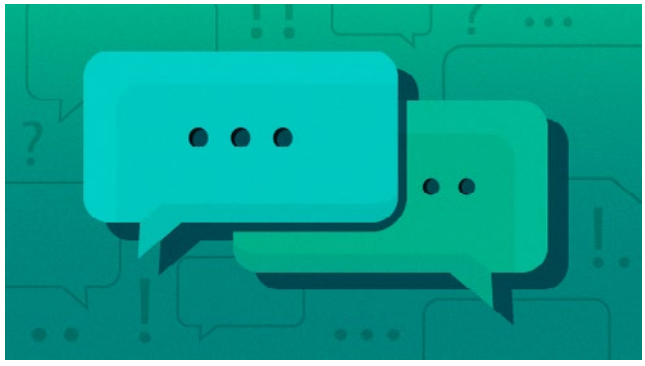
Wrapping up

The shared-file and shared-memory examples show how processes can communicate through *shared storage*, files in one case and memory segments in the other. The APIs for both approaches are relatively straightforward. Do these approaches have a common downside? Modern applications often deal with streaming data, indeed, with massively large streams of data. Neither the shared-file nor the shared-memory approaches are well suited for massive data streams. Channels of one type or another are better suited. Part 2 thus introduces channels and message queues, again with code examples in C.

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.

```
#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.

Named pipes

An unnamed pipe has no backing file: the system maintains an in-memory buffer to transfer bytes from the writer to the reader. Once the writer and reader terminate, the buffer is reclaimed, so the unnamed pipe goes away. By contrast, a named pipe has a backing file and a distinct API.

Let's look at another command line example to get the gist of named pipes. Here are the steps:

- Open two terminals. The working directory should be the same for both.
- In one of the terminals, enter these two commands (the prompt again is **%,** and my comments start with ##):

```
% mkfifo tester  ## creates a backing file named tester
% cat tester    ## type the pipe's contents to stdout
```

At the beginning, nothing should appear in the terminal because nothing has been written yet to the named pipe.

- In the second terminal, enter the command:

```
% cat > tester  ## redirect keyboard input
to the pipe
hello, world!   ## then hit Return key
bye, bye        ## ditto
<Control-C>     ## terminate session with a
                ## Control-C
```

Whatever is typed into this terminal is echoed in the other. Once **Ctrl+C** is entered, the regular command line prompt returns in both terminals: the pipe has been closed.

- Clean up by removing the file that implements the named pipe:

```
% unlink tester
```

As the utility's name *mkfifo* implies, a named pipe also is called a FIFO because the first byte in is the first byte out, and so on. There is a library function named **mkfifo** that creates a named pipe in programs and is used in the next example, which consists of two processes: one writes to the named pipe and the other reads from this pipe (see Example 2).

The *fifoWriter* program above can be summarized as follows:

- The program creates a named pipe for writing:

```
mkfifo(pipeName, 0666); /* read/write perms for user/group/others */
int fd = open(pipeName, O_CREAT | O_WRONLY);
```

where **pipeName** is the name of the backing file passed to **mkfifo** as the first argument. The named pipe then is opened with the by-now familiar call to the **open** function, which returns a file descriptor.

- For a touch of realism, the *fifoWriter* does not write all the data at once, but instead writes a chunk, sleeps a random number of microseconds, and so on. In total, 768,000 4-byte integer values are written to the named pipe.
- After closing the named pipe, the *fifoWriter* also unlinks the file:

```
close(fd);    /* close pipe: generates end-of-stream marker */
unlink(pipeName); /* unlink from the implementing file */
```

Example 2. The *fifoWriter* program

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>

#define MaxLoops        12000   /* outer loop */
#define ChunkSize          16   /* how many written at a time */
#define IntsPerChunk        4   /* four 4-byte ints per chunk */
#define MaxZs             250   /* max microseconds to sleep */

int main() {
  const char* pipeName = "./fifoChannel";
  mkfifo(pipeName, 0666);                      /* read/write for user/group/others */
  int fd = open(pipeName, O_CREAT | O_WRONLY); /* open as write-only */
  if (fd < 0) return -1;                       /* can't go on */

  int i;
  for (i = 0; i < MaxLoops; i++) {          /* write MaxWrites times */
    int j;
    for (j = 0; j < ChunkSize; j++) {       /* each time, write ChunkSize bytes */
      int k;
      int chunk[IntsPerChunk];
      for (k = 0; k < IntsPerChunk; k++)
        chunk[k] = rand();
      write(fd, chunk, sizeof(chunk));
    }
    usleep((rand() % MaxZs) + 1);           /* pause a bit for realism */
  }

  close(fd);           /* close pipe: generates an end-of-stream marker */
  unlink(pipeName);    /* unlink from the implementing file */
  printf("%i ints sent to the pipe.\n", MaxLoops * ChunkSize * IntsPerChunk);

  return 0;
}
```

The system reclaims the backing file once every process connected to the pipe has performed the unlink operation. In this example, there are only two such processes: the *fifoWriter* and the *fifoReader*, both of which do an *unlink* operation.

The two programs should be executed in different terminals with the same working directory. However, the *fifoWriter* should be started before the *fifoReader*, as the former creates the pipe. The *fifoReader* then accesses the already created named pipe (see Example 3).

The *fifoReader* program above can be summarized as follows:

- Because the *fifoWriter* creates the named pipe, the *fifoReader* needs only the standard call **open** to access the pipe through the backing file:

```
const char* file = "./fifoChannel";
int fd = open(file, O_RDONLY);
```

The file opens as read-only.

- The program then goes into a potentially infinite loop, trying to **read** a 4-byte chunk on each iteration. The read call:

```
ssize_t count = read(fd, &next, sizeof(int));
```

returns 0 to indicate end-of-stream, in which case the *fifoReader* breaks out of the loop, closes the named pipe, and unlinks the backing file before terminating.

- After reading a 4-byte integer, the *fifoReader* checks whether the number is a prime. This represents the business logic that a production-grade reader might perform on the received bytes. On a sample run, there were 37,682 primes among the 768,000 integers received.

On repeated sample runs, the *fifoReader* successfully read all of the bytes that the *fifoWriter* wrote. This is not surprising. The two processes execute on the same host, taking network issues out of the equation. Named pipes are a highly reliable and efficient IPC mechanism and, therefore, in wide use.

Here is the output from the two programs, each launched from a separate terminal but with the same working directory:

```
% ./fifoWriter
768000 ints sent to the pipe.
###
% ./fifoReader
Received ints: 768000, primes: 37682
```

Example 3. The *fifoReader* program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

unsigned is_prime(unsigned n) { /* not pretty, but efficient */
  if (n <= 3) return n > 1;
  if (0 == (n % 2) || 0 == (n % 3)) return 0;

  unsigned i;
  for (i = 5; (i * i) <= n; i += 6)
    if (0 == (n % i) || 0 == (n % (i + 2))) return 0;

  return 1; /* found a prime! */
}

int main() {
  const char* file = "./fifoChannel";
  int fd = open(file, O_RDONLY);
  if (fd < 0) return -1; /* no point in continuing */
  unsigned count = 0, total = 0, primes_count = 0;

  while (1) {
    int next;
    int i;

    ssize_t count = read(fd, &next, sizeof(int));
    if (0 == count) break;                  /* end of stream */
    else if (count == sizeof(int)) {        /* read a 4-byte int value */
      total++;
      if (is_prime(next)) primes_count++;
    }
  }

  close(fd);       /* close pipe from read end */
  unlink(file);    /* unlink from the underlying file */
  printf("Received ints: %u, primes: %u\n", total, primes_count);

  return 0;
}
```

Message queues

Pipes have strict FIFO behavior: the first byte written is the first byte read, the second byte written is the second byte read, and so forth. Message queues can behave in the same way but are flexible enough that byte chunks can be retrieved out of FIFO order.

As the name suggests, a message queue is a sequence of messages, each of which has two parts:

- The payload, which is an array of bytes (**char** in C)
- A type, given as a positive integer value; types categorize messages for flexible retrieval

Consider the following depiction of a message queue, with each message labeled with an integer type:

```
          +-+    +-+    +-+    +-+
sender--->|3|--->|2|--->|2|--->|1|--->receiver
          +-+    +-+    +-+    +-+
```

Of the four messages shown, the one labeled 1 is at the front, i.e., closest to the receiver. Next come two messages with label 2, and finally, a message labeled 3 at the back. If strict FIFO behavior were in play, then the messages would be received in the order 1-2-2-3. However, the message queue allows other retrieval orders. For example, the messages could be retrieved by the receiver in the order 3-2-1-2.

The *mqueue* example consists of two programs, the *sender* that writes to the message queue and the *receiver* that reads from this queue. Both programs include the header file *queue.h* shown below in Example 4:

The header file defines a structure type named **queuedMessage**, with **payload** (byte array) and **type** (integer) fields. This file also defines symbolic constants (the **#define** statements), the first two of which are used to generate a key that, in turn, is used to get a message queue ID. The **ProjectId** can be any positive integer value, and the **PathName** must be an existing, accessible file—in this case, the file *queue.h*. The setup statements in both the *sender* and the *receiver* programs are:

```
key_t key = ftok(PathName, ProjectId);   /* generate key
                                            */
int qid = msgget(key, 0666 | IPC_CREAT); /* use key
                                            to get
                                            queue id */
```

The ID **qid** is, in effect, the counterpart of a file descriptor for message queues (see Example 5).

The *sender* program above sends out six messages, two each of a specified type: the first messages are of type 1, the next two of type 2, and the last two of type 3. The sending statement:

```
msgsnd(qid, &msg, sizeof(msg), IPC_NOWAIT);
```

is configured to be non-blocking (the flag **IPC_NOWAIT**) because the messages are so small. The only danger is that a full queue, unlikely in this example, would result in a sending failure. The *receiver* program below also receives messages using the **IPC_NOWAIT** flag (see Example 6).

The *receiver* program does not create the message queue, although the API suggests as much. In the *receiver*, the call:

```
int qid = msgget(key, 0666 | IPC_CREAT);
```

is misleading because of the **IPC_CREAT** flag, but this flag really means *create if needed, otherwise access*. The *sender* program calls **msgsnd** to send messages, whereas the *receiver* calls **msgrcv** to retrieve them. In this example, the *sender* sends the messages in the order 1-1-2-2-3-3, but the *receiver* then retrieves them in the order 3-1-2-1-3-2, showing that message queues are not bound to strict FIFO behavior:

Example 4. The header file *queue.h*

```
#define ProjectId 123
#define PathName  "queue.h" /* any existing, accessible file would do */
#define MsgLen    4
#define MsgCount  6

typedef struct {
  long type;                /* must be of type long */
  char payload[MsgLen + 1];  /* bytes in the message */
} queuedMessage;
```

Example 5. The message *sender* program

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <string.h>
#include "queue.h"

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

int main() {
  key_t key = ftok(PathName, ProjectId);
  if (key < 0) report_and_exit("couldn't get key...");

  int qid = msgget(key, 0666 | IPC_CREAT);
  if (qid < 0) report_and_exit("couldn't get queue id...");

  char* payloads[] = {"msg1", "msg2", "msg3", "msg4", "msg5", "msg6"};
  int types[] = {1, 1, 2, 2, 3, 3}; /* each must be > 0 */
  int i;
  for (i = 0; i < MsgCount; i++) {
    /* build the message */
    queuedMessage msg;
    msg.type = types[i];
    strcpy(msg.payload, payloads[i]);

    /* send the message */
    msgsnd(qid, &msg, sizeof(msg), IPC_NOWAIT); /* don't block */
    printf("%s sent as type %i\n", msg.payload, (int) msg.type);
  }
  return 0;
}
```

```
% ./sender
msg1 sent as type 1
msg2 sent as type 1
msg3 sent as type 2
msg4 sent as type 2
msg5 sent as type 3
msg6 sent as type 3
```

```
% ./receiver
msg5 received as type 3
msg1 received as type 1
msg3 received as type 2
msg2 received as type 1
msg6 received as type 3
msg4 received as type 2
```

Example 6. The message *receiver* program

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include "queue.h"

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

int main() {
  key_t key= ftok(PathName, ProjectId); /* key to identify the queue */
  if (key < 0) report_and_exit("key not gotten...");

  int qid = msgget(key, 0666 | IPC_CREAT); /* access if created already */
  if (qid < 0) report_and_exit("no access to queue...");

  int types[] = {3, 1, 2, 1, 3, 2}; /* different than in sender */
  int i;
  for (i = 0; i < MsgCount; i++) {
    queuedMessage msg; /* defined in queue.h */
    if (msgrcv(qid, &msg, sizeof(msg), types[i], MSG_NOERROR | IPC_NOWAIT) < 0)
      puts("msgrcv trouble...");
    printf("%s received as type %i\n", msg.payload, (int) msg.type);
  }

  /** remove the queue **/
  if (msgctl(qid, IPC_RMID, NULL) < 0)  /* NULL = 'no flags' */
    report_and_exit("trouble removing queue...");

  return 0;
}
```

The output above shows that the *sender* and the *receiver* can be launched from the same terminal. The output also shows that the message queue persists even after the *sender* process creates the queue, writes to it, and exits. The queue goes away only after the *receiver* process explicitly removes it with the call to **msgctl**:

```
if (msgctl(qid, IPC_RMID, NULL) < 0) /* remove
                                        queue */
```

Wrapping up

The pipes and message queue APIs are fundamentally *unidirectional*: one process writes and another reads. There are implementations of bidirectional named pipes, but my two cents is that this IPC mechanism is at its best when it is simplest. As noted earlier, message queues have fallen in popularity—but without good reason; these queues are yet another tool in the IPC toolbox. Part 3 completes this quick tour of the IPC toolbox with code examples of IPC through sockets and signals.

Links

[1] http://man7.org/linux/man-pages/man2/mq_open.2.html
[2] http://man7.org/linux/man-pages/man2/mq_open.2.html#ATTRIBUTES

Sockets and signals

Learn how processes synchronize with each other in Linux.

THE FIRST SECTION focused on IPC through shared storage (files and memory segments), and the second section does the same for basic channels: pipes (named and unnamed) and message queues. This section moves from IPC at the high end (sockets) to IPC at the low end (signals). Code examples flesh out the details.

Sockets

Just as pipes come in two flavors (named and unnamed), so do sockets. IPC sockets (aka Unix domain sockets) enable channel-based communication for processes on the same physical device (*host*), whereas network sockets enable this kind of IPC for processes that can run on different hosts, thereby bringing networking into play. Network sockets need support from an underlying protocol such as TCP (Transmission Control Protocol) or the lower-level UDP (User Datagram Protocol).

By contrast, IPC sockets rely upon the local system kernel to support communication; in particular, IPC sockets communicate using a local file as a socket address. Despite these implementation differences, the IPC socket and network socket APIs are the same in the essentials. The forthcoming example covers network sockets, but the sample server and client programs can run on the same machine because the server uses network address *localhost* (127.0.0.1), the address for the local machine on the local machine.

Sockets configured as streams (discussed below) are bidirectional, and control follows a client/server pattern: the client initiates the conversation by trying to connect to a server, which tries to accept the connection. If everything works, requests from the client and responses from the server then can flow through the channel until this is closed on either end, thereby breaking the connection.

An *iterative* server, which is suited for development only, handles connected clients one at a time to completion: the first client is handled from start to finish, then the second, and so on. The downside is that the handling of a particular client may hang, which then starves all the clients waiting behind. A production-grade server would be *concurrent*, typically using some mix of multi-processing and multi-threading. For example, the Nginx web server on my desktop machine has a pool of four worker processes that can handle client requests concurrently. The following code example keeps the clutter to a minimum by using an iterative server; the focus thus remains on the basic API, not on concurrency.

Finally, the socket API has evolved significantly over time as various POSIX refinements have emerged. The current sample code for server and client is deliberately simple but underscores the bidirectional aspect of a stream-based socket connection. Here's a summary of the flow of control, with the server started in a terminal then the client started in a separate terminal:

- The server awaits client connections and, given a successful connection, reads the bytes from the client.
- To underscore the two-way conversation, the server echoes back to the client the bytes received from the client. These bytes are ASCII character codes, which make up book titles.
- The client writes book titles to the server process and then reads the same titles echoed from the server. Both the server and the client print the titles to the screen. Here is the server's output, essentially the same as the client's:

Example 1. The socket server

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/tcp.h>
#include <arpa/inet.h>
#include "sock.h"

void report(const char* msg, int terminate) {
  perror(msg);
  if (terminate) exit(-1); /* failure */
}

int main() {
  int fd = socket(AF_INET,     /* network versus AF_LOCAL */
                  SOCK_STREAM, /* reliable, bidirectional, arbitrary payload size */
                  0);          /* system picks underlying protocol (TCP) */
  if (fd < 0) report("socket", 1); /* terminate */

  /* bind the server's local address in memory */
  struct sockaddr_in saddr;
  memset(&saddr, 0, sizeof(saddr));          /* clear the bytes */
  saddr.sin_family = AF_INET;                /* versus AF_LOCAL */
  saddr.sin_addr.s_addr = htonl(INADDR_ANY); /* host-to-network endian */
  saddr.sin_port = htons(PortNumber);        /* for listening */

  if (bind(fd, (struct sockaddr *) &saddr, sizeof(saddr)) < 0)
    report("bind", 1); /* terminate */

  /* listen to the socket */
  if (listen(fd, MaxConnects) < 0) /* listen for clients, up to MaxConnects */
    report("listen", 1); /* terminate */

  fprintf(stderr, "Listening on port %i for clients...\n", PortNumber);
  /* a server traditionally listens indefinitely */
  while (1) {
    struct sockaddr_in caddr; /* client address */
    int len = sizeof(caddr);  /* address length could change */

    int client_fd = accept(fd, (struct sockaddr*) &caddr, &len); /* accept blocks */
    if (client_fd < 0) {
      report("accept", 0); /* don't terminate, though there's a problem */
      continue;
    }

    /* read from client */
    int i;
    for (i = 0; i < ConversationLen; i++) {
      char buffer[BuffSize + 1];
      memset(buffer, '\0', sizeof(buffer));
      int count = read(client_fd, buffer, sizeof(buffer));
      if (count > 0) {
        puts(buffer);
        write(client_fd, buffer, sizeof(buffer)); /* echo as confirmation */
      }
    }
    close(client_fd); /* break connection */
  }  /* while(1) */
  return 0;
}
```

```
Listening on port 9876 for clients...
War and Peace
Pride and Prejudice
The Sound and the Fury
```

See Example 1.

The server program above performs the classic four-step to ready itself for client requests and then to accept individual requests. Each step is named after a system function that the server calls:

1. **socket(…)**: get a file descriptor for the socket connection
2. **bind(…)**: bind the socket to an address on the server's host
3. **listen(…)**: listen for client requests
4. **accept(…)**: accept a particular client request

The **socket** call in full is:

```
int sockfd = socket(AF_INET,  /* versus AF_LOCAL */
                    SOCK_STREAM,  /* reliable,
                                     bidirectional */
                    0);           /* system picks
                                     protocol (TCP) */
```

The first argument specifies a network socket as opposed to an IPC socket. There are several options for the second argument, but **SOCK_STREAM** and **SOCK_DGRAM** (datagram) are likely the most used. A *stream-based* socket supports a reliable channel in which lost or altered messages are reported; the channel is bidirectional, and the payloads from one side to the other can be arbitrary in size. By contrast, a datagram-based socket is unreliable (*best try*), unidirectional, and requires fixed-sized payloads. The third argument to **socket** specifies the protocol. For the stream-based socket in play here, there is a single choice, which the zero represents: TCP. Because a successful call to **socket** returns the familiar file descriptor, a socket is written and read with the same syntax as, for example, a local file.

The **bind** call is the most complicated, as it reflects various refinements in the socket API. The point of interest is that this call binds the socket to a memory address on the server machine. However, the **listen** call is straightforward:

```
if (listen(fd, MaxConnects) < 0)
```

The first argument is the socket's file descriptor and the second specifies how many client connections can be accommodated before the server issues a *connection refused* error on an attempted connection. (**MaxConnects** is set to 8 in the header file *sock.h*.)

The **accept** call defaults to a *blocking wait*: the server does nothing until a client attempts to connect and then proceeds. The **accept** function returns **-1** to indicate an error. If the call succeeds, it returns another file descriptor—for a *read/write* socket in contrast to the *accepting socket* referenced by the first argument in the **accept** call. The server uses the read/write socket to read requests from the client and to write responses back. The accepting socket is used only to accept client connections.

By design, a server runs indefinitely. Accordingly, the server can be terminated with a **Ctrl+C** from the command line (see Example 2).

The client program's setup code is similar to the server's. The principal difference between the two is that the client neither listens nor accepts, but instead connects:

```
if (connect(sockfd, (struct sockaddr*) &saddr,
    sizeof(saddr)) < 0)
```

The **connect** call might fail for several reasons; for example, the client has the wrong server address or too many clients are already connected to the server. If the **connect** operation succeeds, the client writes requests and then reads the echoed responses in a **for** loop. After the conversation, both the server and the client **close** the read/write socket, although a close operation on either side is sufficient to close the connection. The client exits thereafter but, as noted earlier, the server remains open for business.

The socket example, with request messages echoed back to the client, hints at the possibilities of arbitrarily rich conversations between the server and the client. Perhaps this is the chief appeal of sockets. It is common on modern systems for client applications (e.g., a database client) to communicate with a server through a socket. As noted earlier, local IPC sockets and network sockets differ only in a few implementation details; in general, IPC sockets have lower overhead and better performance. The communication API is essentially the same for both.

Signals

A *signal* interrupts an executing program and, in this sense, communicates with it. Most signals can be either ignored (blocked) or handled (through desig-

Example 2. The socket client

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>
#include "sock.h"

const char* books[] = {"War and Peace",
                       "Pride and Prejudice",
                       "The Sound and the Fury"};

void report(const char* msg, int terminate) {
  perror(msg);
  if (terminate) exit(-1); /* failure */
}

int main() {
  /* fd for the socket */
  int sockfd = socket(AF_INET,      /* versus AF_LOCAL */
                      SOCK_STREAM,  /* reliable, bidirectional */
                      0);           /* system picks protocol (TCP) */
  if (sockfd < 0) report("socket", 1); /* terminate */

  /* get the address of the host */
  struct hostent* hptr = gethostbyname(Host); /* localhost: 127.0.0.1 */
  if (!hptr) report("gethostbyname", 1); /* is hptr NULL? */
  if (hptr->h_addrtype != AF_INET)       /* versus AF_LOCAL */
    report("bad address family", 1);

  /* connect to the server: configure server's address 1st */
  struct sockaddr_in saddr;
  memset(&saddr, 0, sizeof(saddr));
  saddr.sin_family = AF_INET;
  saddr.sin_addr.s_addr =
     ((struct in_addr*) hptr->h_addr_list[0])->s_addr;
  saddr.sin_port = htons(PortNumber); /* port number in big-endian */

  if (connect(sockfd, (struct sockaddr*) &saddr, sizeof(saddr)) < 0)
    report("connect", 1);

  /* Write some stuff and read the echoes. */
  puts("Connect to server, about to write some stuff...");
  int i;
  for (i = 0; i < ConversationLen; i++) {
    if (write(sockfd, books[i], strlen(books[i])) > 0) {
      /* get confirmation echoed from server and print */
      char buffer[BuffSize + 1];
      memset(buffer, '\0', sizeof(buffer));
      if (read(sockfd, buffer, sizeof(buffer)) > 0)
        puts(buffer);
    }
  }
  puts("Client done, about to exit...");
  close(sockfd); /* close the connection */
  return 0;
}
```

nated code), with **SIGSTOP** (pause) and **SIGKILL** (terminate immediately) as the two notable exceptions. Symbolic constants such as **SIGKILL** have integer values, in this case, 9.

Example 3. The graceful shutdown of a multi-processing system

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

void graceful(int signum) {
  printf("\tChild confirming received signal: %i\n", signum);
  puts("\tChild about to terminate gracefully...");
  sleep(1);
  puts("\tChild terminating now...");
  _exit(0); /* fast-track notification of parent */
}

void set_handler() {
  struct sigaction current;
  sigemptyset(&current.sa_mask);    /* clear the signal set */
  current.sa_flags = 0;             /* enables setting sa_handler, not sa_action */
  current.sa_handler = graceful;        /* specify a handler */
  sigaction(SIGTERM, &current, NULL);    /* register the handler */
}

void child_code() {
  set_handler();

  while (1) {   /** loop until interrupted **/
    sleep(1);
    puts("\tChild just woke up, but going back to sleep.");
  }
}

void parent_code(pid_t cpid) {
  puts("Parent sleeping for a time...");
  sleep(5);

  /* Try to terminate child. */
  if (-1 == kill(cpid, SIGTERM)) {
    perror("kill");
    exit(-1);
  }
  wait(NULL); /** wait for child to terminate **/
  puts("My child terminated, about to exit myself...");
}

int main() {
  pid_t pid = fork();
  if (pid < 0) {
    perror("fork");
    return -1; /* error */
  }
  if (0 == pid)
    child_code();
  else
    parent_code(pid);
  return 0;  /* normal */
}
```

Signals can arise in user interaction. For example, a user hits **Ctrl+C** from the command line to terminate a program started from the command-line; **Ctrl+C** generates a **SIGTERM** signal. **SIGTERM** for *terminate*, unlike **SIGKILL**, can be either blocked or handled. One process also can signal another, thereby making signals an IPC mechanism.

Consider how a multi-processing application such as the Nginx web server might be shut down gracefully from another process. The **kill** function:

```
int kill(pid_t pid, int signum);
  /* declaration */
```

can be used by one process to terminate another process or group of processes. If the first argument to function **kill** is greater than zero, this argument is treated as the *pid* (process ID) of the targeted process; if the argument is zero, the argument identifies the group of processes to which the signal sender belongs.

The second argument to **kill** is either a standard signal number (e.g., **SIGTERM** or **SIGKILL**) or 0, which makes the call to **signal** a query about whether the *pid* in the first argument is indeed valid. The graceful shutdown of a multi-processing application thus could be accomplished by sending a *terminate* signal—a call to the **kill** function with **SIGTERM** as the second argument—to the group of processes that make up the application. (The Nginx master process could terminate the worker processes with a call to **kill** and then exit itself.) The **kill** function, like so many library functions, houses power and flexibility in a simple invocation syntax (see Example 3).

The *shutdown* program above simulates the graceful shutdown of a multi-processing system, in this case, a simple one consisting of a parent process and a single child process. The simulation works as follows:

- The parent process tries to fork a child. If the fork succeeds, each process executes its own code: the child executes the function **child_code**, and the parent executes the function **parent_code**.
- The child process goes into a potentially infinite loop in which the child sleeps for a second, prints a message, goes back to sleep, and so on. It is precisely a **SIGTERM** signal from the parent that

causes the child to execute the signal-handling callback function **graceful**. The signal thus breaks the child process out of its loop and sets up the graceful termination of both the child and the parent. The child prints a message before terminating.

- The parent process, after forking the child, sleeps for five seconds so that the child can execute for a while; of course, the child mostly sleeps in this simulation. The parent then calls the **kill** function with **SIGTERM** as the second argument, waits for the child to terminate, and then exits.

  Here is the output from a sample run:

```
% ./shutdown
Parent sleeping for a time...
        Child just woke up, but going back to sleep.
        Child just woke up, but going back to sleep.
        Child just woke up, but going back to sleep.
        Child just woke up, but going back to sleep.
        Child confirming received signal: 15  ## SIGTERM is 15
        Child about to terminate gracefully...
        Child terminating now...
My child terminated, about to exit myself...
```

  For the signal handling, the example uses the **sigaction** library function (POSIX recommended) rather than the legacy **signal** function, which has portability issues. Here are the code segments of chief interest:

- If the call to **fork** succeeds, the parent executes the **parent_code** function and the child executes the **child_code** function. The parent waits for five seconds before signaling the child:

```
puts("Parent sleeping for a time...");
sleep(5);
if (-1 == kill(cpid, SIGTERM)) {
...
```

  If the **kill** call succeeds, the parent does a **wait** on the child's termination to prevent the child from becoming a permanent zombie; after the wait, the parent exits.

- The **child_code** function first calls **set_handler** and then goes into its potentially infinite sleeping loop. Here is the **set_handler** function for review:

```
void set_handler() {
  struct sigaction current;          /* current setup */
  sigemptyset(&current.sa_mask);     /* clear the signal set */
  current.sa_flags = 0;              /* for setting sa_handler,
                                        not sa_action */
  current.sa_handler = graceful;      /* specify a handler */
  sigaction(SIGTERM, &current, NULL); /* register the handler */
}
```

The first three lines are preparation. The fourth statement sets the handler to the function **graceful**, which prints some messages before calling **_exit** to terminate. The fifth and last statement then registers the handler with the system through the call to **sigaction**. The first argument to **sigaction** is **SIGTERM** for *terminate*, the second is the current **sigaction** setup, and the last argument (**NULL** in this case) can be used to save a previous **sigaction** setup, perhaps for later use.

Using signals for IPC is indeed a minimalist approach, but a tried-and-true one at that. IPC through signals clearly belongs in the IPC toolbox.

Wrapping up this guide

THIS GUIDE ON IPC has covered the following mechanisms through code examples:

- Shared files
- Shared memory (with semaphores)
- Pipes (named and unnamed)
- Message queues
- Sockets
- Signals

Even today, when thread-centric languages such as Java, C#, and Go have become so popular, IPC remains appealing because concurrency through multi-processing has an obvious advantage over multi-threading: every process, by default, has its own address space, which rules out memory-based race conditions in multi-processing unless the IPC mechanism of shared memory is brought into play. (Shared memory must be locked in both multi-processing and multi-threading for safe concurrency.) Anyone who has written even an elementary multi-threading program with communication via shared variables knows how challenging it can be to write thread-safe yet clear, efficient code. Multi-processing with single-threaded processes remains a viable—indeed, quite appealing—way to take advantage of today's multi-processor machines without the inherent risk of memory-based race conditions.

There is no simple answer, of course, to the question of which among the IPC mechanisms is the best. Each involves a trade-off typical in programming: simplicity versus functionality. Signals, for example, are a relatively simple IPC mechanism but do not support rich conversations among processes. If such a conversion is needed, then one of the other choices is more appropriate. Shared files with locking is reasonably straightforward, but shared files may not perform well enough if processes need to share massive data streams; pipes or even sockets, with more complicated APIs, might be a better choice. Let the problem at hand guide the choice.

Although the sample code (available on my website [1]) is all in C, other programming languages often provide thin wrappers around these IPC mechanisms. The code examples are short and simple enough, I hope, to encourage you to experiment.

Links

[1] https://condor.depaul.edu/mkalin/

WRITE FOR US

In 2010, Red Hat CEO Jim Whitehurst announced the launch of Opensource.com in a post titled *Welcome to the conversation on Opensource.com*. He explained, "This site is one of the ways in which Red Hat gives something back to the open source community. Our desire is to create a connection point for conversations about the broader impact that open source can have—and is having—even beyond the software world." he wrote, adding, "All ideas are welcome, and all participants are welcome. This will not be a site for Red Hat, about Red Hat. Instead, this will be a site for open source, about the future."

By 2013, Opensource.com was publishing an average of 46 articles per month, and in March 2016, Opensource.com surpassed 1-million page views for the first time. In 2019, Opensource.com averages more than 1.5 million page views and 90 articles per month.

More than 60% of our content is contributed by members of open source communities, and additional articles are written by the editorial team and other Red Hat contributors. A small, international team of staff editors and Community Moderators work closely with contributors to curate, polish, publish, and promote open source stories from around the world.

Would you like to write for us? Send pitches and inquiries to open@opensource.com.

To learn more, read 7 big reasons to contribute to Opensource.com.