Inter-Process Communication

Introduction

We can divide process interactions into a two broad categories:

1. the coordination of operations with other processes:
    - synchronization (e.g. mutexes and condition variables)
    - the exchange of signals (e.g. *kill(2)*)
    - control operations (e.g. *fork(2)*, *wait(2)*, *ptrace(2)*)
2. the exchange of data between processes:
    - uni-directional data processing pipelines
    - bi-directional interactions

The first of these are discussed in other readings and lectures. This is an introduction to the exchange of data between processes.

Simple Uni-Directional Byte Streams

These are easy to create and trivial to use. A pipe can be opened by a parent and inherited by a child, who simply reads standard input and writes standard output. Such pipelines can be used as part of a standard processing model:

```
macro-processor | compiler | assembler > output
```

or as custom constructions for one-off tasks:

```
find . -newer $TIMESTAMP | grep -v '*.o' | tar cfz archive.tgz -T -
```

All such uses have a few key characteristics in common:

- Each program accepts a byte-stream input, and produces a byte-stream output that is a well defined function of the input.
- Each program in the pipe-line operates independently, and is unaware that the others exist or what they might do.
- Byte streams are inherently unstructured. If there is any structure to the data they carry (e.g. newline-delimited lines or comma-separated-values) these conventions are implemented by parsers in the affected applications.
- Ensuring that the output of one program is suitable input to the next is the responsibility of the agent who creates the pipeline.
- Similar results could be obtained by writing the output of each program into a (temporary) file, and then using that file as input to the next program.

Pipes are temporary files with a few special features, that recognize the difference between a file (whose contents are relatively static) and an inter-process data stream:

- If the reader exhausts all of the data in the pipe, but there is still an open write file descriptor, the reader does not get an *End of File*(EOF). Rather the reader is blocked until more data becomes available, or the write side is closed.
- The available buffering capacity of the pipe may be limited. If the writer gets too far ahead of the reader, the operating system may block the writer until the reader catches up. This is called *flow control*.
- Writing to a pipe that no longer has an open read file descriptor is illegal, and the writer will be sent an exception signal (SIGPIPE).

- When the read and write file descriptors are both closed, the file is automatically deleted.

Because a pipeline (in principle) represents a closed system, the only data privacy mechanisms tend to be the protections on the initial input files and final output files. There is generally no authentication or encryption of the data being passed between successive processes in the pipeline.

Named Pipes and Mail-Boxes

A named-pipe (*fifo(7)*) is a baby-step towards explicit connections. It can be thought of as a persistent pipe, whose reader and writer(s) can open it by name, rather than inheriting it from a *pipe(2)* system call. A normal pipe is custom-plumbed to interconnect processes started by a single user. A named pipe can be used as a rendezvous point for unrelated processes. Named pipes are almost as simple to use as ordinary pipes, but ...

- Readers and writers have no way of authenticating one-another's identities.
- Writes from multiple writers may be interspersed, with no indications of which bytes came from whom.
- They do not enable clean fail-overs from a failed reader to its successor.
- All readers and writers must be running on the same node.

Recognizing these limitations, some operating systems have crated more general inter-process communication mechanisms, often called *mailboxes*. While implementations differ, common features include:

- Data is not a byte-stream. Rather each write is stored and delivered as a distinct message.
- Each write is accompanied by authenticated identification information about its sender.
- Unprocessed messages remain in the mailbox after the death of a reader and can be retrieved by the next reader.

But mailboxes still subject to single node/single operating system restrictions, and most distributed applications are now based on general and widely standardized network protocols.

General Network Connections

Most operating systems now provide a fairly standard set of network communications APIs. The associated Linux APIs ar:

- *socket(2)* ... create an inter-process communication end-point with an associated protocol and data model.
- *bind(2)* ... associate a *socket* with a local network address.
- *connect(2)* ... establish a connection to a remote network address.
- *listen(2)* ... await an incoming connection request.
- *accept(2)* ... accept an incoming connection request.
- *send(2)* ... send a message over a socket.
- *recv(2)* ... receive a message from a socket.

These APIs directly provide a range of different communications options:

- byte streams over reliable connections (e.g. TCP)
- best effort data-grams (e.g. UDP)

But they also form a foundation for higher level communication/service models. A few examples include:

- Remote Procedure Calls ... distribued request/response APIs.
- RESTful service models ... layered on top of HTTP GETs and PUTs.
- Publish/Subscribe services ... content based information flow.

0

Using more general networking models enables processes to interact with services all over the world, but this adds considerable complexity:

- Ensuring interoperability with software running under differerent operating systems on computers with different instruction set architectures.
- Dealing with the security issues associated with exchanging data and services with unknown systems over public networks.
- Discovering the addresses of (a constantly changing set of) servers.
- Detecting and recovering from (relatively common) connection and node failures.

Applications are forced to choose between a simple but strictly local model (pipes) or a general but highly complex model (network communications). But there is yet another issue: performance. Protocol stacks may be many layers deep, and data may be processed and copied many times. Network communication may have limited throughput, and high latencies.

Shared Memory

Sometimes performance is more important than generality.

- The network drivers, MPEG decoders, and video rendering in a set top box are guaranteed to be local. Making these operations more efficient can greatly reduce the required processing power, resulting in a smaller form-factor, reduced heat dissipation, and a lower product cost.
- The network drivers, protocol interpreters, write-back cache, RAID implementation, and back-end drivers in a storage array are guaranteed to be local. Significantly reducing the execution time per write operation is the difference between an industry leader and road-kill.

High performance for Inter-Process Communication means generally means:

- efficiency ... low cost (instructions, nano-seconds, watts) per byte transferred.
- throughput ... maximum number of bytes (or messages) per second that can be transferred.
- latency ... minimum delay between sender write and receiver read.

If we want ultra high performance Inter-Process Communication between two local processes, buffering the data through the operating system and/or protocol stacks is not the way to get it. The fastest and most efficient way to move data between processes is through shared memory:

- create a file for communication.
- each process maps that file into its virtual address space.
- the shared segment might be locked-down, so that it is never paged out.
- the communicating processes agree on a set of data structures (e.g. polled lock-free circular buffers) in the shared segment.
- anything written into the shared memory segment will be immediately visible to all of the processes that have it mapped in to their address spaces.

Once the shared segment has been created and mapped into the participating process' address spaces, the Operating System plays no role in the subsequent data exchanges. Moving data in this way is extremely efficient and blindingly fast ... but (like all good things) this performance comes at a price:

- This can only be used between processes on the same memory bus.
- A bug in one of the processes can easily destroy the communications data structures.
- There is no authentication (beyond access control on the shared file) of which data came from which process.

Network Connections and Out-of-Band Signals

0

In most cases, event completions can be reported simply by sending a message (announcing the completion) to the waiter. But what if there are megabytes of queued requests, and we want to send a message to abort those queued requests? Data sent down a network connection is FIFO ... and one of the problems with FIFO schdeduling is the delays waiting for the processing of earlier but longer messages. Occasionally, we would like to make it possible for an important message to go directly to front of the line.

If the recipient was local, we might consider sending a signal that could invoke a registered handler, and flush (without processing) all of the bufferred data. This works because the signals travel over a different channel than the buffered data. Such communication is often called *out-of-band*, because it does not travel over the normal data path.

We can achieve a similar effect with network based services by opening multiple communications channels:

- one heavily used channel for normal requests
- another reserved for out-of-band requests

The server on the far end periodically polls the out-of-band channel before taking requests from the normal communications channel. This adds a little overhead to the processing, but makes it possible to preempt queued operations. The choosen polling interal represents a trade-off between added overhead (to check for out-of-band messages) and how long we might go (how much wasted work we might do) before noticing an out-of-band message.

Named Pipes (FIFOs - First In First Out)

6.3.1 Basic Concepts

A named pipe works much like a regular pipe, but does have some noticeable differences.

- Named pipes exist as a device special file in the file system.
- Processes of different ancestry can share data through a named pipe.
- When all I/O is done by sharing processes, the named pipe remains in the file system for later use.

6.3.2 Creating a FIFO

There are several ways of creating a named pipe. The first two can be done directly from the shell.

```
mknod MYFIFO p
mkfifo a=rw MYFIFO
```

The above two commands perform identical operations, with one exception. The mkfifo command provides a hook for altering the permissions on the FIFO file directly after creation. With mknod, a quick call to the chmod command will be necessary.

FIFO files can be quickly identified in a physical file system by the ``p'' indicator seen here in a long directory listing:

```
$ ls -l MYFIFO
prw-r--r--   1 root     root            0 Dec 14 22:15 MYFIFO|
```

Also notice the vertical bar (``pipe sign'') located directly after the file name. Another great reason to run Linux, eh?

To create a FIFO in C, we can make use of the mknod() system call:

```
  LIBRARY FUNCTION: mknod();

  PROTOTYPE: int mknod( char *pathname, mode_t mode, dev_t dev);
    RETURNS: 0 on success,
             -1 on error: errno = EFAULT (pathname invalid)
                                  EACCES (permission denied)
                                  ENAMETOOLONG (pathname too long)
```

```
                                    ENOENT (invalid pathname)
                                    ENOTDIR (invalid pathname)
                                    (see man page for mknod for others)

  NOTES: Creates a filesystem node (file, device file, or FIFO)
```

I will leave a more detailed discussion of mknod() to the man page, but let's consider a simple example of FIFO creation from C:

```
                mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In this case, the file ``/tmp/MYFIFO'' is created as a FIFO file. The requested permissions are ``0666'', although they are affected by the umask setting as follows:

```
                final_umask = requested_permissions & ~original_umask
```

A common trick is to use the umask() system call to temporarily zap the umask value:

```
                umask(0);
                mknod("/tmp/MYFIFO", S_IFIFO|0666, 0);
```

In addition, the third argument to mknod() is ignored unless we are creating a device file. In that instance, it should specify the major and minor numbers of the device file.

6.3.3 FIFO Operations

I/O operations on a FIFO are essentially the same as for normal pipes, with once major exception. An ``open'' system call or library function should be used to physically open up a channel to the pipe. With half-duplex pipes, this is unnecessary, since the pipe resides in the kernel and not on a physical filesystem. In our examples, we will treat the pipe as a stream, opening it up with fopen(), and closing it with fclose().

Consider a simple server process:

```
/*****************************************************************************
*
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett

*****************************************************************************
 MODULE: fifoserver.c

*****************************************************************************
/

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

```
#include <sys/stat.h>
#include <unistd.h>

#include <linux/stat.h>

#define FIFO_FILE       "MYFIFO"

int main(void)
{
        FILE *fp;
        char readbuf[80];

        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1)
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(readbuf, 80, fp);
                printf("Received string: %s\n", readbuf);
                fclose(fp);
        }

        return(0);
}
```

Since a FIFO blocks by default, run the server in the background after you compile it:

```
        $ fifoserver&
```

We will discuss a FIFO's blocking action in a moment. First, consider the following simple client frontend to our server:

```
/*****************************************************************************
*
 Excerpt from "Linux Programmer's Guide - Chapter 6"
 (C)opyright 1994-1995, Scott Burkett

*****************************************************************************
 MODULE: fifoclient.c

*****************************************************************************
/

#include <stdio.h>
#include <stdlib.h>

#define FIFO_FILE       "MYFIFO"

int main(int argc, char *argv[])
{
        FILE *fp;

        if ( argc != 2 ) {
                printf("USAGE: fifoclient [string]\n");
                exit(1);
        }

        if((fp = fopen(FIFO_FILE, "w")) == NULL) {
                perror("fopen");
                exit(1);
        }

        fputs(argv[1], fp);

        fclose(fp);
        return(0);
}
```

6.3.4 Blocking Actions on a FIFO

Normally, blocking occurs on a FIFO. In other words, if the FIFO is opened for reading, the process will "block" until some other process opens it for writing. This action works vice-versa as well. If this behavior is undesirable, the O_NONBLOCK flag can be used in an open() call to disable the default blocking action.

In the case with our simple server, we just shoved it into the background, and let it do its blocking there. The alternative would be to jump to another virtual console and run the client end, switching back and forth to see the resulting action.

6.3.5 The Infamous SIGPIPE Signal

On a last note, pipes must have a reader and a writer. If a process tries to write to a pipe that has no reader, it will be sent the SIGPIPE signal from the kernel. This is imperative when more than two processes are involved in a pipeline.

SEND(2) **Linux Programmer's Manual** **SEND(2)**

NAME top

send, sendto, sendmsg - send a message on a socket

SYNOPSIS top

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

DESCRIPTION top

The system calls **send**(), **sendto**(), and **sendmsg**() are used to transmit a message to another socket.

The **send**() call may be used only when the socket is in a *connected* state (so that the intended recipient is known). The only difference between **send**() and write(2) is the presence of *flags*. With a zero *flags* argument, **send**() is equivalent to write(2). Also, the following call

```
send(sockfd, buf, len, flags);
```

is equivalent to

```
sendto(sockfd, buf, len, flags, NULL, 0);
```

The argument *sockfd* is the file descriptor of the sending socket.

If **sendto**() is used on a connection-mode (**SOCK_STREAM**, **SOCK_SEQPACKET**) socket, the arguments *dest_addr* and *addrlen* are ignored (and the error **EISCONN** may be returned when they are not NULL and 0), and the error **ENOTCONN** is returned when the socket was not actually connected. Otherwise, the address of the target is given by *dest_addr* with *addrlen* specifying its size. For **sendmsg**(), the address of the target is given by *msg.msg_name*, with *msg.msg_namelen* specifying its size.

For **send**() and **sendto**(), the message is found in *buf* and has length *len*. For **sendmsg**(), the message is pointed to by the elements of the array *msg.msg_iov*. The **sendmsg**() call also allows sending ancillary data (also known as control information).

If the message is too long to pass atomically through the underlying protocol, the error **EMSGSIZE** is returned, and the message is not transmitted.

No indication of failure to deliver is implicit in a **send**(). Locally detected errors are indicated by a return value of -1.

When the message does not fit into the send buffer of the socket, **send**() normally blocks, unless the socket has been placed in nonblocking I/O mode. In nonblocking mode it would fail with the error **EAGAIN** or **EWOULDBLOCK** in this case. The select(2) call may be used to determine when it is possible to send more data.

The flags argument

The *flags* argument is the bitwise OR of zero or more of the following flags.

MSG_CONFIRM (since Linux 2.3.15)
Tell the link layer that forward progress happened: you got a successful reply from the other side. If the link layer doesn't get this it will regularly reprobe the neighbor (e.g., via a unicast ARP). Only valid on **SOCK_DGRAM** and **SOCK_RAW** sockets and currently implemented only for IPv4 and IPv6. See arp(7) for details.

MSG_DONTROUTE
Don't use a gateway to send out the packet, send to hosts only on directly connected networks. This is usually used only by diagnostic or routing programs. This is defined only for protocol families that route; packet sockets don't.

MSG_DONTWAIT (since Linux 2.2)
Enables nonblocking operation; if the operation would block, **EAGAIN** or **EWOULDBLOCK** is returned. This provides similar behavior to setting the **O_NONBLOCK** flag (via the fcntl(2) **F_SETFL** operation), but differs in that **MSG_DONTWAIT** is a per-call option, whereas **O_NONBLOCK** is a setting on the open file description (see open(2)), which will affect all threads in the calling process and as well as other processes that hold file descriptors referring to the same open file description.

MSG_EOR (since Linux 2.2)
Terminates a record (when this notion is supported, as for sockets of type **SOCK_SEQPACKET**).

MSG_MORE (since Linux 2.4.4)
The caller has more data to send. This flag is used with TCP sockets to obtain the same effect as the **TCP_CORK** socket

option (see tcp(7)), with the difference that this flag can be set on a per-call basis.

Since Linux 2.6, this flag is also supported for UDP sockets, and informs the kernel to package all of the data sent in calls with this flag set into a single datagram which is transmitted only when a call is performed that does not specify this flag. (See also the **UDP_CORK** socket option described in udp(7).)

MSG_NOSIGNAL (since Linux 2.2)
Don't generate a **SIGPIPE** signal if the peer on a stream-oriented socket has closed the connection. The **EPIPE** error is still returned. This provides similar behavior to using sigaction(2) to ignore **SIGPIPE**, but, whereas **MSG_NOSIGNAL** is a per-call feature, ignoring **SIGPIPE** sets a process attribute that affects all threads in the process.

MSG_OOB
Sends *out-of-band* data on sockets that support this notion (e.g., of type **SOCK_STREAM**); the underlying protocol must also support *out-of-band* data.

sendmsg()

The definition of the *msghdr* structure employed by **sendmsg**() is as follows:

```
struct msghdr {
    void         *msg_name;       /* optional address */
    socklen_t     msg_namelen;    /* size of address */
    struct iovec *msg_iov;        /* scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* ancillary data, see below */
    size_t        msg_controllen; /* ancillary data buffer len */
    int           msg_flags;      /* flags (unused) */
};
```

The *msg_name* field is used on an unconnected socket to specify the target address for a datagram. It points to a buffer containing the address; the *msg_namelen* field should be set to the size of the address. For a connected socket, these fields should be specified as NULL and 0, respectively.

The *msg_iov* and *msg_iovlen* fields specify scatter-gather locations, as for writev(2).

You may send control information using the *msg_control* and *msg_controllen* members. The maximum control buffer length the kernel can process is limited per socket by the value in */proc/sys/net/core/optmem_max*; see socket(7).

The *msg_flags* field is ignored.

RETURN VALUE top

On success, these calls return the number of bytes sent. On error, -1 is returned, and *errno* is set appropriately.

ERRORS top

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their respective manual pages.

EACCES (For UNIX domain sockets, which are identified by pathname) Write permission is denied on the destination socket file, or search permission is denied for one of the directories the path prefix. (See path_resolution(7).)

(For UDP sockets) An attempt was made to send to a network/broadcast address as though it was a unicast address.

EAGAIN or **EWOULDBLOCK**
The socket is marked nonblocking and the requested operation would block. POSIX.1-2001 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EAGAIN (Internet domain datagram sockets) The socket referred to by *sockfd* had not previously been bound to an address and, upon attempting to bind it to an ephemeral port, it was determined that all port numbers in the ephemeral port range are currently in use. See the discussion of */proc/sys/net/ipv4/ip_local_port_range* in ip(7).

EBADF *sockfd* is not a valid open file descriptor.

ECONNRESET
Connection reset by peer.

EDESTADDRREQ
The socket is not connection-mode, and no peer address is set.

EFAULT An invalid user space address was specified for an argument.

EINTR A signal occurred before any data was transmitted; see signal(7).

EINVAL Invalid argument passed.

EISCONN
The connection-mode socket was connected already but a recipient was specified. (Now either this error is returned, or the recipient specification is ignored.)

EMSGSIZE
The socket type requires that message be sent atomically, and the size of the message to be sent made this impossible.

ENOBUFS
The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion. (Normally, this does not occur in Linux. Packets are just silently dropped when a device queue overflows.)

ENOMEM No memory available.

ENOTCONN
The socket is not connected, and no target has been given.

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

EOPNOTSUPP
Some bit in the *flags* argument is inappropriate for the socket type.

EPIPE The local end has been shut down on a connection oriented socket. In this case, the process will also receive a **SIGPIPE** unless **MSG_NOSIGNAL** is set.

CONFORMING TO top

4.4BSD, SVr4, POSIX.1-2001. These interfaces first appeared in 4.2BSD.

POSIX.1-2001 describes only the **MSG_OOB** and **MSG_EOR** flags. POSIX.1-2008 adds a specification of **MSG_NOSIGNAL**. The **MSG_CONFIRM** flag is a Linux extension.

NOTES top

According to POSIX.1-2001, the *msg_controllen* field of the *msghdr* structure should be typed as *socklen_t*, but glibc currently types it as *size_t*.

See sendmmsg(2) for information about a Linux-specific system call that can be used to transmit multiple datagrams in a single call.

BUGS top

Linux may return **EPIPE** instead of **ENOTCONN**.

EXAMPLE top

An example of the use of **sendto**() is shown in getaddrinfo(3).

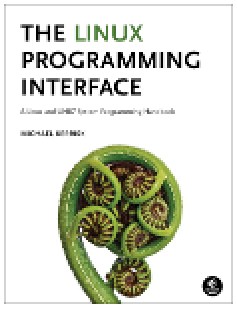
SEE ALSO top

fcntl(2), getsockopt(2), recv(2), select(2), sendfile(2), sendmmsg(2), shutdown(2), socket(2), write(2), cmsg(3), ip(7), socket(7), tcp(7), udp(7)

COLOPHON top

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Linux **2016-03-15** **SEND(2)**



RECV(2) **Linux Programmer's Manual** **RECV(2)**

NAME top

recv, recvfrom, recvmsg - receive a message from a socket

SYNOPSIS top

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

DESCRIPTION top

The **recv**(), **recvfrom**(), and **recvmsg**() calls are used to receive messages from a socket. They may be used to receive data on both connectionless and connection-oriented sockets. This page first describes common features of all three system calls, and then describes the differences between the calls.

The only difference between **recv**() and read(2) is the presence of *flags*. With a zero *flags* argument, **recv**() is generally equivalent to read(2) (but see NOTES). Also, the following call

```
recv(sockfd, buf, len, flags);
```

is equivalent to

```
recvfrom(sockfd, buf, len, flags, NULL, NULL);
```

All three calls return the length of the message on successful completion. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from.

If no messages are available at the socket, the receive calls wait for a message to arrive, unless the socket is nonblocking (see fcntl(2)), in which case the value -1 is returned and the external variable *errno* is set to **EAGAIN** or **EWOULDBLOCK**. The receive calls normally return any data available, up to the requested amount, rather than waiting for receipt of the full amount requested.

An application can use select(2), poll(2), or epoll(7) to determine when more data arrives on a socket.

The flags argument

The *flags* argument is formed by ORing one or more of the following values:

MSG_CMSG_CLOEXEC (**recvmsg**() only; since Linux 2.6.23)
Set the close-on-exec flag for the file descriptor received via a UNIX domain file descriptor using the **SCM_RIGHTS** operation (described in unix(7)). This flag is useful for the same reasons as the **O_CLOEXEC** flag of open(2).

MSG_DONTWAIT (since Linux 2.2)
Enables nonblocking operation; if the operation would block, the call fails with the error **EAGAIN** or **EWOULDBLOCK**. This provides similar behavior to setting the **O_NONBLOCK** flag (via the fcntl(2) **F_SETFL** operation), but differs in that **MSG_DONTWAIT** is a per-call option, whereas **O_NONBLOCK** is a setting on the open file description (see open(2)), which will affect all threads in the calling process and as well as other processes that hold file descriptors referring to the same open file description.

MSG_ERRQUEUE (since Linux 2.2)
This flag specifies that queued errors should be received from the socket error queue. The error is passed in an ancillary message with a type dependent on the protocol (for IPv4 **IP_RECVERR**). The user should supply a buffer of sufficient size. See cmsg(3) and ip(7) for more information. The payload of the original packet that caused the error is passed as normal data via *msg_iovec*. The original destination address of the datagram that caused the error is supplied via *msg_name*.

For local errors, no address is passed (this can be checked with the *cmsg_len* member of the *cmsghdr*). For error receives, the **MSG_ERRQUEUE** is set in the *msghdr*. After an error has been passed, the pending socket error is regenerated based on the next queued error and will be passed on the next socket operation.

The error is supplied in a *sock_extended_err* structure:

```
#define SO_EE_ORIGIN_NONE    0
#define SO_EE_ORIGIN_LOCAL   1
#define SO_EE_ORIGIN_ICMP    2
#define SO_EE_ORIGIN_ICMP6   3

struct sock_extended_err
{
    uint32_t ee_errno;   /* error number */
    uint8_t  ee_origin;  /* where the error originated */
```

```
                   uint8_t  ee_type;    /* type */
                   uint8_t  ee_code;    /* code */
                   uint8_t  ee_pad;     /* padding */
                   uint32_t ee_info;    /* additional information */
                   uint32_t ee_data;    /* other data */
                   /* More data may follow */
               };

               struct sockaddr *SO_EE_OFFENDER(struct sock_extended_err *);
```

ee_errno contains the *errno* number of the queued error. *ee_origin* is the origin code of where the error originated. The other fields are protocol-specific. The macro **SOCK_EE_OFFENDER** returns a pointer to the address of the network object where the error originated from given a pointer to the ancillary message. If this address is not known, the *sa_family* member of the *sockaddr* contains **AF_UNSPEC** and the other fields of the *sockaddr* are undefined. The payload of the packet that caused the error is passed as normal data.

For local errors, no address is passed (this can be checked with the *cmsg_len* member of the *cmsghdr*). For error receives, the **MSG_ERRQUEUE** is set in the *msghdr*. After an error has been passed, the pending socket error is regenerated based on the next queued error and will be passed on the next socket operation.

MSG_OOB
This flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols.

MSG_PEEK
This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data.

MSG_TRUNC (since Linux 2.2)
For raw (**AF_PACKET**), Internet datagram (since Linux 2.4.27/2.6.8), netlink (since Linux 2.6.22), and UNIX datagram (since Linux 3.4) sockets: return the real length of the packet or datagram, even when it was longer than the passed buffer.

For use with Internet stream sockets, see tcp(7).

MSG_WAITALL (since Linux 2.2)
This flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned. This flag has no effect for datagram sockets.

recvfrom()

recvfrom() places the received message into the buffer *buf*. The caller must specify the size of the buffer in *len*.

If *src_addr* is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by *src_addr*. In this case, *addrlen* is a value-result argument. Before the call, it should be initialized to the size of the buffer associated with *src_addr*. Upon return, *addrlen* is updated to contain the actual size of the source address. The returned address is truncated if the buffer provided is too small; in this case, *addrlen* will return a value greater than was supplied to the call.

If the caller is not interested in the source address, *src_addr* and *addrlen* should be specified as NULL.

recv()

The **recv**() call is normally used only on a *connected* socket (see connect(2)). It is equivalent to the call:

```
    recvfrom(fd, buf, len, flags, NULL, 0);
```

recvmsg()

The **recvmsg**() call uses a *msghdr* structure to minimize the number of directly supplied arguments. This structure is defined as follows in *<sys/socket.h>*:

```
    struct iovec {                    /* Scatter/gather array items */
        void  *iov_base;              /* Starting address */
        size_t iov_len;               /* Number of bytes to transfer */
    };

    struct msghdr {
        void         *msg_name;       /* optional address */
        socklen_t     msg_namelen;    /* size of address */
        struct iovec *msg_iov;        /* scatter/gather array */
        size_t        msg_iovlen;     /* # elements in msg_iov */
        void         *msg_control;    /* ancillary data, see below */
        size_t        msg_controllen; /* ancillary data buffer len */
        int           msg_flags;      /* flags on received message */
    };
```

The *msg_name* field points to a caller-allocated buffer that is used to return the source address if the socket is unconnected. The caller should set *msg_namelen* to the size of this buffer before this call; upon return from a successful call, *msg_namelen* will contain the length of the returned address. If the application does not need to know the source address, *msg_name* can be specified as NULL.

The fields *msg_iov* and *msg_iovlen* describe scatter-gather locations, as discussed in readv(2).

The field *msg_control*, which has length *msg_controllen*, points to a buffer for other protocol control-related messages or miscellaneous ancillary data. When **recvmsg**() is called, *msg_controllen* should contain the length of the available buffer in *msg_control*; upon return from a successful call it will contain the length of the control message sequence.

The messages are of the form:

```
    struct cmsghdr {
        size_t cmsg_len;    /* Data byte count, including header
                               (type is socklen_t in POSIX) */
        int    cmsg_level;  /* Originating protocol */
        int    cmsg_type;   /* Protocol-specific type */
    /* followed by
        unsigned char cmsg_data[]; */
    };
```

Ancillary data should be accessed only by the macros defined in cmsg(3).

As an example, Linux uses this ancillary data mechanism to pass extended errors, IP options, or file descriptors over UNIX domain sockets.

The *msg_flags* field in the *msghdr* is set on return of **recvmsg**(). It can contain several flags:

MSG_EOR
indicates end-of-record; the data returned completed a record (generally used with sockets of type **SOCK_SEQPACKET**).

MSG_TRUNC
indicates that the trailing portion of a datagram was discarded because the datagram was larger than the buffer supplied.

MSG_CTRUNC
indicates that some control data were discarded due to lack of space in the buffer for ancillary data.

MSG_OOB
is returned to indicate that expedited or out-of-band data were received.

MSG_ERRQUEUE
indicates that no data was received but an extended error from the socket error queue.

RETURN VALUE top

These calls return the number of bytes received, or -1 if an error occurred. In the event of an error, *errno* is set to indicate the error.

When a stream socket peer has performed an orderly shutdown, the return value will be 0 (the traditional "end-of-file" return).

Datagram sockets in various domains (e.g., the UNIX and Internet domains) permit zero-length datagrams. When such a datagram is received, the return value is 0.

The value 0 may also be returned if the requested number of bytes to receive from a stream socket was 0.

ERRORS top

These are some standard errors generated by the socket layer. Additional errors may be generated and returned from the underlying protocol modules; see their manual pages.

EAGAIN or **EWOULDBLOCK**
The socket is marked nonblocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received. POSIX.1 allows either error to be returned for this case, and does not require these constants to have the same value, so a portable application should check for both possibilities.

EBADF The argument *sockfd* is an invalid file descriptor.

ECONNREFUSED
A remote host refused to allow the network connection (typically because it is not running the requested service).

EFAULT The receive buffer pointer(s) point outside the process's address space.

EINTR The receive was interrupted by delivery of a signal before any data were available; see signal(7).

EINVAL Invalid argument passed.

ENOMEM Could not allocate memory for **recvmsg**().

ENOTCONN
The socket is associated with a connection-oriented protocol and has not been connected (see connect(2) and accept(2)).

ENOTSOCK
The file descriptor *sockfd* does not refer to a socket.

CONFORMING TO top

POSIX.1-2001, POSIX.1-2008, 4.4BSD (these interfaces first appeared in 4.2BSD).

POSIX.1 describes only the **MSG_OOB**, **MSG_PEEK**, and **MSG_WAITALL** flags.

NOTES top

If a zero-length datagram is pending, read(2) and **recv**() with a *flags* argument of zero provide different behavior. In this circumstance, read(2) has no effect (the datagram remains pending), while **recv**() consumes the pending datagram.

The *socklen_t* type was invented by POSIX. See also accept(2).

According to POSIX.1, the *msg_controllen* field of the *msghdr* structure should be typed as *socklen_t*, but glibc currently types it as *size_t*.

See recvmmsg(2) for information about a Linux-specific system call that can be used to receive multiple datagrams in a single call.

EXAMPLE top

An example of the use of **recvfrom**() is shown in getaddrinfo(3).

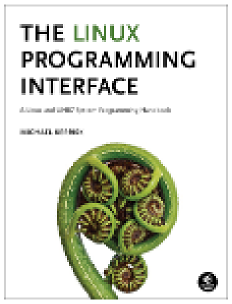
SEE ALSO top

fcntl(2), getsockopt(2), read(2), recvmmsg(2), select(2), shutdown(2), socket(2), cmsg(3), sockatmark(3), socket(7)

COLOPHON top

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Linux **2016-07-17** **RECV(2)**



MMAP(2) Linux Programmer's Manual MMAP(2)

NAME top

mmap, munmap - map or unmap files or devices into memory

SYNOPSIS top

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

See NOTES for information on feature test macro requirements.

DESCRIPTION top

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in *addr*. The *length* argument specifies the length of the mapping.

If *addr* is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If *addr* is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

The contents of a file mapping (as opposed to an anonymous mapping; see **MAP_ANONYMOUS** below), are initialized using *length* bytes starting at offset *offset* in the file (or other object) referred to by the file descriptor *fd*. *offset* must be a multiple of the page size as returned by *sysconf(_SC_PAGE_SIZE)*.

The *prot* argument describes the desired memory protection of the mapping (and must not conflict with the open mode of the file). It is either **PROT_NONE** or the bitwise OR of one or more of the following flags:

PROT_EXEC Pages may be executed.

PROT_READ Pages may be read.

PROT_WRITE Pages may be written.

PROT_NONE Pages may not be accessed.

The *flags* argument determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file. This behavior is determined by including exactly one of the following values in *flags*:

MAP_SHARED
Share this mapping. Updates to the mapping are visible to other processes mapping the same region, and (in the case of file-backed mappings) are carried through to the underlying file. (To precisely control when updates are carried through to the underlying file requires the use of msync(2).)

MAP_PRIVATE
Create a private copy-on-write mapping. Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. It is unspecified whether changes made to the file after the **mmap**() call are visible in the mapped region.

Both of these flags are described in POSIX.1-2001 and POSIX.1-2008.

In addition, zero or more of the following values can be ORed in *flags*:

MAP_32BIT (since Linux 2.4.20, 2.6)
Put the mapping into the first 2 Gigabytes of the process address space. This flag is supported only on x86-64, for 64-bit programs. It was added to allow thread stacks to be allocated somewhere in the first 2GB of memory, so as to improve context-switch performance on some early 64-bit processors. Modern x86-64 processors no longer have this performance problem, so use of this flag is not required on those systems. The **MAP_32BIT** flag is ignored when **MAP_FIXED** is set.

MAP_ANON
Synonym for **MAP_ANONYMOUS**. Deprecated.

MAP_ANONYMOUS
The mapping is not backed by any file; its contents are initialized to zero. The *fd* argument is ignored; however, some implementations require *fd* to be -1 if **MAP_ANONYMOUS** (or **MAP_ANON**) is specified, and portable applications should ensure this. The *offset* argument should be zero. The use of **MAP_ANONYMOUS** in conjunction with **MAP_SHARED** is supported on Linux only since kernel 2.4.

MAP_DENYWRITE
This flag is ignored. (Long ago, it signaled that attempts to write to the underlying file should fail with **ETXTBUSY**. But

this was a source of denial-of-service attacks.)

MAP_EXECUTABLE
This flag is ignored.

MAP_FILE
Compatibility flag. Ignored.

MAP_FIXED
Don't interpret *addr* as a hint: place the mapping at exactly that address. *addr* must be a multiple of the page size. If the memory region specified by *addr* and *len* overlaps pages of any existing mapping(s), then the overlapped part of the existing mapping(s) will be discarded. If the specified address cannot be used, **mmap**() will fail. Because requiring a fixed address for a mapping is less portable, the use of this option is discouraged.

MAP_GROWSDOWN
Used for stacks. Indicates to the kernel virtual memory system that the mapping should extend downward in memory.

MAP_HUGETLB (since Linux 2.6.32)
Allocate the mapping using "huge pages." See the Linux kernel source file *Documentation/vm/hugetlbpage.txt* for further information, as well as NOTES, below.

MAP_HUGE_2MB, **MAP_HUGE_1GB** (since Linux 3.8)
Used in conjunction with **MAP_HUGETLB** to select alternative hugetlb page sizes (respectively, 2 MB and 1 GB) on systems that support multiple hugetlb page sizes.

More generally, the desired huge page size can be configured by encoding the base-2 logarithm of the desired page size in the six bits at the offset **MAP_HUGE_SHIFT**. (A value of zero in this bit field provides the default huge page size; the default huge page size can be discovered vie the *Hugepagesize* field exposed by */proc/meminfo*.) Thus, the above two constants are defined as:

```
#define MAP_HUGE_2MB    (21 << MAP_HUGE_SHIFT)
#define MAP_HUGE_1GB    (30 << MAP_HUGE_SHIFT)
```

The range of huge page sizes that are supported by the system can be discovered by listing the subdirectories in */sys/kernel/mm/hugepages*.

MAP_LOCKED (since Linux 2.5.37)
Mark the mmaped region to be locked in the same way as mlock(2). This implementation will try to populate (prefault) the whole range but the mmap call doesn't fail with **ENOMEM** if this fails. Therefore major faults might happen later on. So the semantic is not as strong as mlock(2). One should use **mmap**() plus mlock(2) when major faults are not acceptable

after the initialization of the mapping. The **MAP_LOCKED** flag is ignored in older kernels.

MAP_NONBLOCK (since Linux 2.5.46)
Only meaningful in conjunction with **MAP_POPULATE**. Don't perform read-ahead: create page tables entries only for pages that are already present in RAM. Since Linux 2.6.23, this flag causes **MAP_POPULATE** to do nothing. One day, the combination of **MAP_POPULATE** and **MAP_NONBLOCK** may be reimplemented.

MAP_NORESERVE
Do not reserve swap space for this mapping. When swap space is reserved, one has the guarantee that it is possible to modify the mapping. When swap space is not reserved one might get **SIGSEGV** upon a write if no physical memory is available. See also the discussion of the file */proc/sys/vm/overcommit_memory* in proc(5). In kernels before 2.6, this flag had effect only for private writable mappings.

MAP_POPULATE (since Linux 2.5.46)
Populate (prefault) page tables for a mapping. For a file mapping, this causes read-ahead on the file. This will help to reduce blocking on page faults later. **MAP_POPULATE** is supported for private mappings only since Linux 2.6.23.

MAP_STACK (since Linux 2.6.27)
Allocate the mapping at an address suitable for a process or thread stack. This flag is currently a no-op, but is used in the glibc threading implementation so that if some architectures require special treatment for stack allocations, support can later be transparently implemented for glibc.

MAP_UNINITIALIZED (since Linux 2.6.33)
Don't clear anonymous pages. This flag is intended to improve performance on embedded devices. This flag is honored only if the kernel was configured with the **CONFIG_MMAP_ALLOW_UNINITIALIZED** option. Because of the security implications, that option is normally enabled only on embedded devices (i.e., devices where one has complete control of the contents of user memory).

Of the above flags, only **MAP_FIXED** is specified in POSIX.1-2001 and POSIX.1-2008. However, most systems also support **MAP_ANONYMOUS** (or its synonym **MAP_ANON**).

Some systems document the additional flags **MAP_AUTOGROW**, **MAP_AUTORESRV**, **MAP_COPY**, and **MAP_LOCAL**.

Memory mapped by **mmap**() is preserved across fork(2), with the same attributes.

A file is mapped in multiples of the page size. For a file that is not a multiple of the page size, the remaining memory is zeroed when mapped, and writes to that region are not written out to the file. The effect of changing the size of the underlying file of a mapping on the pages that correspond to added or removed regions of the file is unspecified.

munmap()

The **munmap**() system call deletes the mappings for the specified address range, and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

The address *addr* must be a multiple of the page size (but *length* need not be). All pages containing a part of the indicated range are unmapped, and subsequent references to these pages will generate **SIGSEGV**. It is not an error if the indicated range does not contain any mapped pages.

RETURN VALUE top

On success, **mmap**() returns a pointer to the mapped area. On error, the value **MAP_FAILED** (that is, *(void *) -1*) is returned, and *errno* is set to indicate the cause of the error.

On success, **munmap**() returns 0. On failure, it returns -1, and *errno* is set to indicate the cause of the error (probably to **EINVAL**).

ERRORS top

EACCES A file descriptor refers to a non-regular file. Or a file mapping was requested, but *fd* is not open for reading. Or **MAP_SHARED** was requested and **PROT_WRITE** is set, but *fd* is not open in read/write (**O_RDWR**) mode. Or **PROT_WRITE** is set, but the file is append-only.

EAGAIN The file has been locked, or too much memory has been locked (see setrlimit(2)).

EBADF *fd* is not a valid file descriptor (and **MAP_ANONYMOUS** was not set).

EINVAL We don't like *addr*, *length*, or *offset* (e.g., they are too large, or not aligned on a page boundary).

EINVAL (since Linux 2.6.12) *length* was 0.

EINVAL *flags* contained neither **MAP_PRIVATE** or **MAP_SHARED**, or contained both of these values.

ENFILE The system-wide limit on the total number of open files has been reached.

ENODEV The underlying filesystem of the specified file does not support memory mapping.

ENOMEM No memory is available.

ENOMEM The process's maximum number of mappings would have been exceeded. This error can also occur for munmap(2), when unmapping a region in the middle of an existing mapping, since this results in two smaller mappings on either side of the region being unmapped.

EOVERFLOW
On 32-bit architecture together with the large file extension (i.e., using 64-bit *off_t*): the number of pages used for *length* plus number of pages used for *offset* would overflow *unsigned long* (32 bits).

EPERM The *prot* argument asks for **PROT_EXEC** but the mapped area belongs to a file on a filesystem that was mounted no-exec.

EPERM The operation was prevented by a file seal; see fcntl(2).

ETXTBSY
MAP_DENYWRITE was set but the object specified by *fd* is open for writing.

Use of a mapped region can result in these signals:

SIGSEGV
Attempted write into a region mapped as read-only.

SIGBUS Attempted access to a portion of the buffer that does not correspond to the file (for example, beyond the end of the file, including the case where another process has truncated the file).

ATTRIBUTES top

For an explanation of the terms used in this section, see attributes(7).

Interface	**Attribute**	**Value**
mmap(), **munmap**()	Thread safety	MT-Safe

CONFORMING TO top

POSIX.1-2001, POSIX.1-2008, SVr4, 4.4BSD.

AVAILABILITY top

On POSIX systems on which **mmap**(), msync(2), and **munmap**() are available, **_POSIX_MAPPED_FILES** is defined in *<unistd.h>* to a value greater than 0. (See also sysconf(3).)

NOTES top

On some hardware architectures (e.g., i386), **PROT_WRITE** implies **PROT_READ**. It is architecture dependent whether **PROT_READ** implies **PROT_EXEC** or not. Portable programs should always set **PROT_EXEC** if they intend to execute code in the new mapping.

The portable way to create a mapping is to specify *addr* as 0 (NULL), and omit **MAP_FIXED** from *flags*. In this case, the system chooses the address for the mapping; the address is chosen so as not to conflict with any existing mapping, and will not be 0. If the **MAP_FIXED** flag is specified, and *addr* is 0 (NULL), then the mapped address will be 0 (NULL).

Certain *flags* constants are defined only if suitable feature test macros are defined (possibly by default): **_DEFAULT_SOURCE** with glibc 2.19 or later; or **_BSD_SOURCE** or **_SVID_SOURCE** in glibc 2.19 and earlier. (Requiring **_GNU_SOURCE** also suffices, and requiring that macro specifically would have been more logical, since these flags are all Linux-specific.) The relevant flags are: **MAP_32BIT**, **MAP_ANONYMOUS** (and the synonym **MAP_ANON**), **MAP_DENYWRITE**, **MAP_EXECUTABLE**, **MAP_FILE**, **MAP_GROWSDOWN**, **MAP_HUGETLB**, **MAP_LOCKED**, **MAP_NONBLOCK**, **MAP_NORESERVE**, **MAP_POPULATE**, and **MAP_STACK**.

Timestamps changes for file-backed mappings

For file-backed mappings, the *st_atime* field for the mapped file may be updated at any time between the **mmap**() and the corresponding unmapping; the first reference to a mapped page will update the field if it has not been already.

The *st_ctime* and *st_mtime* field for a file mapped with **PROT_WRITE** and **MAP_SHARED** will be updated after a write to the mapped region, and before a subsequent msync(2) with the **MS_SYNC** or **MS_ASYNC** flag, if one occurs.

Huge page (Huge TLB) mappings

For mappings that employ huge pages, the requirements for the arguments of **mmap**() and **munmap**() differ somewhat from the requirements for mappings that use the native system page size.

For **mmap**(), *offset* must be a multiple of the underlying huge page size. The system automatically aligns *length* to be a multiple of the underlying huge page size.

For **munmap**(), *addr* and *length* must both be a multiple of the underlying huge page size.

C library/kernel differences

This page describes the interface provided by the glibc **mmap**() wrapper function. Originally, this function invoked a system call of the same name. Since kernel 2.4, that system call has been superseded by mmap2(2), and nowadays the glibc **mmap**() wrapper function invokes mmap2(2) with a suitably adjusted value for *offset*.

BUGS top

On Linux, there are no guarantees like those suggested above under **MAP_NORESERVE**. By default, any process can be killed at any moment when the system runs out of memory.

In kernels before 2.6.7, the **MAP_POPULATE** flag has effect only if *prot* is specified as **PROT_NONE**.

SUSv3 specifies that **mmap**() should fail if *length* is 0. However, in kernels before 2.6.12, **mmap**() succeeded in this case: no mapping was created and the call returned *addr*. Since kernel 2.6.12, **mmap**() fails with the error **EINVAL** for this case.

POSIX specifies that the system shall always zero fill any partial page at the end of the object and that system will never write any modification of the object beyond its end. On Linux, when you write data to such partial page after the end of the object, the data stays in the page cache even after the file is closed and unmapped and even though the data is never written to the file itself, subsequent mappings may see the modified content. In some cases, this could be fixed by calling msync(2) before the unmap takes place; however, this doesn't work on tmpfs (for example, when using POSIX shared memory interface documented in shm_overview(7)).

EXAMPLE top

The following program prints part of the file specified in its first command-line argument to standard output. The range of bytes to be printed is specified via offset and length values in the second and third command-line arguments. The program creates a memory mapping of the required pages of the file and then uses write(2) to output the desired bytes.

Program source

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

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)
```

```
       int
       main(int argc, char *argv[])
       {
           char *addr;
           int fd;
           struct stat sb;
           off_t offset, pa_offset;
           size_t length;
           ssize_t s;

           if (argc < 3 || argc > 4) {
               fprintf(stderr, "%s file offset [length]\n", argv[0]);
               exit(EXIT_FAILURE);
           }

           fd = open(argv[1], O_RDONLY);
           if (fd == -1)
               handle_error("open");

           if (fstat(fd, &sb) == -1)           /* To obtain file size */
               handle_error("fstat");

           offset = atoi(argv[2]);
           pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);
               /* offset for mmap() must be page aligned */

           if (offset >= sb.st_size) {
               fprintf(stderr, "offset is past end of file\n");
               exit(EXIT_FAILURE);
           }

           if (argc == 4) {
               length = atoi(argv[3]);
               if (offset + length > sb.st_size)
                   length = sb.st_size - offset;
                       /* Can't display bytes past end of file */

           } else {    /* No length arg ==> display to end of file */
               length = sb.st_size - offset;
           }

           addr = mmap(NULL, length + offset - pa_offset, PROT_READ,
                       MAP_PRIVATE, fd, pa_offset);
           if (addr == MAP_FAILED)
               handle_error("mmap");

           s = write(STDOUT_FILENO, addr + offset - pa_offset, length);
           if (s != length) {
               if (s == -1)
                   handle_error("write");

               fprintf(stderr, "partial write");
               exit(EXIT_FAILURE);
           }

           munmap(addr, length + offset - pa_offset);
           close(fd);

           exit(EXIT_SUCCESS);
       }
```

SEE ALSO top

getpagesize(2), memfd_create(2), mincore(2), mlock(2), mmap2(2), mprotect(2), mremap(2), msync(2), remap_file_pages(2), setrlimit(2), shmat(2), shm_open(3), shm_overview(7)

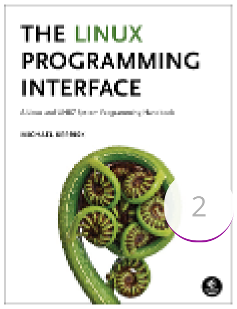
The descriptions of the following files in proc(5): */proc/[pid]/maps*, */proc/[pid]/map_files*, and */proc/[pid]/smaps*.

B.O. Gallmeister, POSIX.4, O'Reilly, pp. 128-129 and 389-391.

COLOPHON top

This page is part of release 4.08 of the Linux *man-pages* project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.

Linux **2016-07-17** **MMAP(2)**



Part II

Concurrency

25

A Dialogue on Concurrency

Professor: *And thus we reach the second of our three pillars of operating systems:* ***concurrency****.*

Student: *I thought there were four pillars...?*

Professor: *Nope, that was in an older version of the book.*

Student: *Umm... OK. So what is concurrency, oh wonderful professor?*

Professor: *Well, imagine we have a peach —*

Student: *(interrupting) Peaches again! What is it with you and peaches?*

Professor: *Ever read T.S. Eliot? The Love Song of J. Alfred Prufrock, "Do I dare to eat a peach", and all that fun stuff?*

Student: *Oh yes! In English class in high school. Great stuff! I really liked the part where —*

Professor: *(interrupting) This has nothing to do with that — I just like peaches. Anyhow, imagine there are a lot of peaches on a table, and a lot of people who wish to eat them. Let's say we did it this way: each eater first identifies a peach visually, and then tries to grab it and eat it. What is wrong with this approach?*

Student: *Hmmm... seems like you might see a peach that somebody else also sees. If they get there first, when you reach out, no peach for you!*

Professor: *Exactly! So what should we do about it?*

Student: *Well, probably develop a better way of going about this. Maybe form a line, and when you get to the front, grab a peach and get on with it.*

Professor: *Good! But what's wrong with your approach?*

Student: *Sheesh, do I have to do all the work?*

Professor: *Yes.*

Student: *OK, let me think. Well, we used to have many people grabbing for peaches all at once, which is faster. But in my way, we just go one at a time, which is correct, but quite a bit slower. The best kind of approach would be fast and correct, probably.*

Professor: *You are really starting to impress. In fact, you just told us everything we need to know about concurrency! Well done.*

Student: *I did? I thought we were just talking about peaches. Remember, this is usually a part where you make it about computers again.*

Professor: *Indeed. My apologies! One must never forget the concrete. Well, as it turns out, there are certain types of programs that we call* ***multi-threaded*** *applications; each* ***thread*** *is kind of like an independent agent running around in this program, doing things on the program's behalf. But these threads access memory, and for them, each spot of memory is kind of like one of those peaches. If we don't coordinate access to memory between threads, the program won't work as expected. Make sense?*

Student: *Kind of. But why do we talk about this in an OS class? Isn't that just application programming?*

Professor: *Good question! A few reasons, actually. First, the OS must support multi-threaded applications with primitives such as* ***locks*** *and* ***condition variables****, which we'll talk about soon. Second, the OS itself was the first concurrent program — it must access its own memory very carefully or many strange and terrible things will happen. Really, it can get quite grisly.*

Student: *I see. Sounds interesting. There are more details, I imagine?*

Professor: *Indeed there are...*

26 Concurrency: An Introduction

Thus far, we have seen the development of the basic abstractions that the OS performs. We have seen how to take a single physical CPU and turn it into multiple **virtual CPUs**, thus enabling the illusion of multiple programs running at the same time. We have also seen how to create the illusion of a large, private **virtual memory** for each process; this abstraction of the **address space** enables each program to behave as if it has its own memory when indeed the OS is secretly multiplexing address spaces across physical memory (and sometimes, disk).

In this note, we introduce a new abstraction for a single running process: that of a **thread**. Instead of our classic view of a single point of execution within a program (i.e., a single PC where instructions are being fetched from and executed), a **multi-threaded** program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from). Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they *share* the same address space and thus can access the same data.

The state of a single thread is thus very similar to that of a process. It has a program counter (PC) that tracks where the program is fetching instructions from. Each thread has its own private set of registers it uses for computation; thus, if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch** must take place. The context switch between threads is quite similar to the context switch between processes, as the register state of T1 must be saved and the register state of T2 restored before running T2. With processes, we saved state to a **process control block (PCB)**; now, we'll need one or more **thread control blocks (TCBs)** to store the state of each thread of a process. There is one major difference, though, in the context switch we perform between threads as compared to processes: the address space remains the same (i.e., there is no need to switch which page table we are using).

One other major difference between threads and processes concerns the stack. In our simple model of the address space of a classic process (which we can now call a **single-threaded** process), there is a single stack, usually residing at the bottom of the address space (Figure 26.1, left).

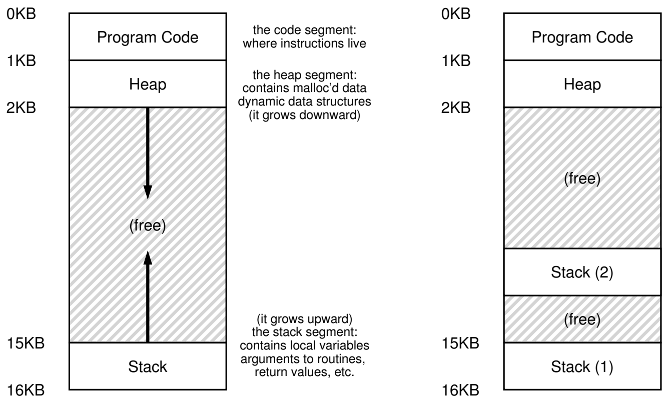

0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)

Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

However, in a multi-threaded process, each thread runs independently and of course may call into various routines to do whatever work it is doing. Instead of a single stack in the address space, there will be one per thread. Let's say we have a multi-threaded process that has two threads in it; the resulting address space looks different (Figure 26.1, right).

In this figure, you can see two stacks spread throughout the address space of the process. Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called **thread-local** storage, i.e., the stack of the relevant thread.

You might also notice how this ruins our beautiful address space layout. Before, the stack and heap could grow independently and trouble only arose when you ran out of room in the address space. Here, we no longer have such a nice situation. Fortunately, this is usually OK, as stacks do not generally have to be very large (the exception being in programs that make heavy use of recursion).

26.1 Why Use Threads?

Before getting into the details of threads and some of the problems you might have in writing multi-threaded programs, let's first answer a more simple question. Why should you use threads at all?

As it turns out, there are at least two major reasons you should use threads. The first is simple: **parallelism**. Imagine you are writing a program that performs operations on very large arrays, for example, adding two large arrays together, or incrementing the value of each element in the array by some amount. If you are running on just a single processor, the task is straightforward: just perform each operation and be done. However, if you are executing the program on a system with multiple processors, you have the potential of speeding up this process considerably by using the processors to each perform a portion of the work. The task of transforming your standard **single-threaded** program into a program that does this sort of work on multiple CPUs is called **parallelization**, and using a thread per CPU to do this work is a natural and typical way to make programs run faster on modern hardware.

The second reason is a bit more subtle: to avoid blocking program progress due to slow I/O. Imagine that you are writing a program that performs different types of I/O: either waiting to send or receive a message, for an explicit disk I/O to complete, or even (implicitly) for a page fault to finish. Instead of waiting, your program may wish to do something else, including utilizing the CPU to perform computation, or even issuing further I/O requests. Using threads is a natural way to avoid getting stuck; while one thread in your program waits (i.e., is blocked waiting for I/O), the CPU scheduler can switch to other threads, which are ready to run and do something useful. Threading enables **overlap** of I/O with other activities *within* a single program, much like **multiprogramming** did for processes *across* programs; as a result, many modern server-based applications (web servers, database management systems, and the like) make use of threads in their implementations.

Of course, in either of the cases mentioned above, you could use multiple *processes* instead of threads. However, threads share an address space and thus make it easy to share data, and hence are a natural choice when constructing these types of programs. Processes are a more sound choice for logically separate tasks where little sharing of data structures in memory is needed.

26.2 An Example: Thread Creation

Let's get into some of the details. Say we wanted to run a program that creates two threads, each of which does some independent work, in this case printing "A" or "B". The code is shown in Figure 26.2 (page 4).

The main program creates two threads, each of which will run the function `mythread()`, though with different arguments (the string `A` or `B`). Once a thread is created, it may start running right away (depending on the whims of the scheduler); alternately, it may be put in a "ready" but not "running" state and thus not run yet. Of course, on a multiprocessor, the threads could even be running at the same time, but let's not worry about this possibility quite yet.

After creating the two threads (let's call them T1 and T2), the main thread calls `pthread_join()`, which waits for a particular thread to complete. It does so twice, thus ensuring T1 and T2 will run and complete before finally allowing the main thread to run again; when it does, it will print "main: end" and exit. Overall, three threads were employed during this run: the main thread, T1, and T2.

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

Figure 26.2: **Simple Thread Creation Code (t0.c)**

Let us examine the possible execution ordering of this little program. In the execution diagram (Figure 26.3, page 5), time increases in the downwards direction, and each column shows when a different thread (the main one, or Thread 1, or Thread 2) is running.

Note, however, that this ordering is not the only possible ordering. In fact, given a sequence of instructions, there are quite a few, depending on which thread the scheduler decides to run at a given point. For example, once a thread is created, it may run immediately, which would lead to the execution shown in Figure 26.4 (page 5).

We also could even see "B" printed before "A", if, say, the scheduler decided to run Thread 2 first even though Thread 1 was created earlier; there is no reason to assume that a thread that is created first will run first. Figure 26.5 (page 5) shows this final execution ordering, with Thread 2 getting to strut its stuff before Thread 1.

As you might be able to see, one way to think about thread creation is that it is a bit like making a function call; however, instead of first executing the function and then returning to the caller, the system instead creates a new thread of execution for the routine that is being called, and it runs independently of the caller, perhaps before returning from the create, but perhaps much later. What runs next is determined by the OS **scheduler**, and although the scheduler likely implements some sensible algorithm, it is hard to know what will run at any given moment in time.

As you also might be able to tell from this example, threads make life complicated: it is already hard to tell what will run when! Computers are hard enough to understand without concurrency. Unfortunately, with concurrency, it simply gets worse. Much worse.

main	**Thread 1**	**Thread2**
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
		runs
		prints “B”
		returns
prints “main: end”		

Figure 26.3: **Thread Trace (1)**

main	**Thread 1**	**Thread2**
starts running		
prints “main: begin”		
creates Thread 1		
	runs	
	prints “A”	
	returns	
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
returns immediately; T1 is done		
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Figure 26.4: **Thread Trace (2)**

main	**Thread 1**	**Thread2**
starts running		
prints “main: begin”		
creates Thread 1		
creates Thread 2		
		runs
		prints “B”
		returns
waits for T1		
	runs	
	prints “A”	
	returns	
waits for T2		
returns immediately; T2 is done		
prints “main: end”		

Figure 26.5: **Thread Trace (3)**

```
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

Figure 26.6: **Sharing Data: Uh Oh (t1.c)**

26.3 Why It Gets Worse: Shared Data

The simple thread example we showed above was useful in showing how threads are created and how they can run in different orders depending on how the scheduler decides to run them. What it doesn't show you, though, is how threads interact when they access shared data.

Let us imagine a simple example where two threads wish to update a global shared variable. The code we'll study is in Figure 26.6 (page 6).

Here are a few notes about the code. First, as Stevens suggests [SR05], we wrap the thread creation and join routines to simply exit on failure; for a program as simple as this one, we want to at least notice an error occurred (if it did), but not do anything very smart about it (e.g., just exit). Thus, `Pthread_create()` simply calls `pthread_create()` and makes sure the return code is 0; if it isn't, `Pthread_create()` just prints a message and exits.

Second, instead of using two separate function bodies for the worker threads, we just use a single piece of code, and pass the thread an argument (in this case, a string) so we can have each thread print a different letter before its messages.

Finally, and most importantly, we can now look at what each worker is trying to do: add a number to the shared variable `counter`, and do so 10 million times (1e7) in a loop. Thus, the desired final result is: 20,000,000.

We now compile and run the program, to see how it behaves. Sometimes, everything works how we might expect:

```
prompt> gcc -o main main.c -Wall -pthread
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Unfortunately, when we run this code, even on a single processor, we don't necessarily get the desired result. Sometimes, we get:

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19345221)
```

Let's try it one more time, just to see if we've gone crazy. After all, aren't computers supposed to produce **deterministic** results, as you have been taught?! Perhaps your professors have been lying to you? *(gasp)*

```
prompt> ./main
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 19221041)
```

Not only is each run wrong, but also yields a *different* result! A big question remains: why does this happen?

TIP: KNOW AND USE YOUR TOOLS

You should always learn new tools that help you write, debug, and understand computer systems. Here, we use a neat tool called a **disassembler.** When you run a disassembler on an executable, it shows you what assembly instructions make up the program. For example, if we wish to understand the low-level code to update a counter (as in our example), we run `objdump` (Linux) to see the assembly code:

```
prompt> objdump -d main
```

Doing so produces a long listing of all the instructions in the program, neatly labeled (particularly if you compiled with the `-g` flag), which includes symbol information in the program. The `objdump` program is just one of many tools you should learn how to use; a debugger like `gdb`, memory profilers like `valgrind` or `purify`, and of course the compiler itself are others that you should spend time to learn more about; the better you are at using your tools, the better systems you'll be able to build.

26.4 The Heart Of The Problem: Uncontrolled Scheduling

To understand why this happens, we must understand the code sequence that the compiler generates for the update to `counter`. In this case, we wish to simply add a number (1) to `counter`. Thus, the code sequence for doing so might look something like this (in x86);

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

This example assumes that the variable `counter` is located at address 0x8049a1c. In this three-instruction sequence, the x86 `mov` instruction is used first to get the memory value at the address and put it into register `eax`. Then, the add is performed, adding 1 (0x1) to the contents of the `eax` register, and finally, the contents of `eax` are stored back into memory at the same address.

Let us imagine one of our two threads (Thread 1) enters this region of code, and is thus about to increment `counter` by one. It loads the value of `counter` (let's say it's 50 to begin with) into its register `eax`. Thus, `eax=50` for Thread 1. Then it adds one to the register; thus `eax=51`. Now, something unfortunate happens: a timer interrupt goes off; thus, the OS saves the state of the currently running thread (its PC, its registers including `eax`, etc.) to the thread's TCB.

Now something worse happens: Thread 2 is chosen to run, and it enters this same piece of code. It also executes the first instruction, getting the value of `counter` and putting it into its `eax` (remember: each thread when running has its own private registers; the registers are **virtualized** by the context-switch code that saves and restores them). The value of

			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**counter**
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Figure 26.7: **The Problem: Up Close and Personal**

counter is still 50 at this point, and thus Thread 2 has eax=50. Let's then assume that Thread 2 executes the next two instructions, incrementing eax by 1 (thus eax=51), and then saving the contents of eax into counter (address 0x8049a1c). Thus, the global variable counter now has the value 51.

Finally, another context switch occurs, and Thread 1 resumes running. Recall that it had just executed the mov and add, and is now about to perform the final mov instruction. Recall also that eax=51. Thus, the final mov instruction executes, and saves the value to memory; the counter is set to 51 again.

Put simply, what has happened is this: the code to increment counter has been run twice, but counter, which started at 50, is now only equal to 51. A "correct" version of this program should have resulted in the variable counter equal to 52.

Let's look at a detailed execution trace to understand the problem better. Assume, for this example, that the above code is loaded at address 100 in memory, like the following sequence (note for those of you used to nice, RISC-like instruction sets: x86 has variable-length instructions; this mov instruction takes up 5 bytes of memory, and the add only 3):

```
100 mov    0x8049a1c, %eax
105 add    $0x1, %eax
108 mov    %eax, 0x8049a1c
```

With these assumptions, what happens is shown in Figure 26.7. Assume the counter starts at value 50, and trace through this example to make sure you understand what is going on.

What we have demonstrated here is called a **race condition**: the results depend on the timing execution of the code. With some bad luck (i.e., context switches that occur at untimely points in the execution), we get the wrong result. In fact, we may get a different result each time; thus, instead of a nice **deterministic** computation (which we are used to from computers), we call this result **indeterminate**, where it is not known what the output will be and it is indeed likely to be different across runs.

Because multiple threads executing this code can result in a race condition, we call this code a **critical section**. A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

What we really want for this code is what we call **mutual exclusion**. This property guarantees that if one thread is executing within the critical section, the others will be prevented from doing so.

Virtually all of these terms, by the way, were coined by Edsger Dijkstra, who was a pioneer in the field and indeed won the Turing Award because of this and other work; see his 1968 paper on "Cooperating Sequential Processes" [D68] for an amazingly clear description of the problem. We'll be hearing more about Dijkstra in this section of the book.

26.5 The Wish For Atomicity

One way to solve this problem would be to have more powerful instructions that, in a single step, did exactly whatever we needed done and thus removed the possibility of an untimely interrupt. For example, what if we had a super instruction that looked like this?

```
memory-add 0x8049a1c, $0x1
```

Assume this instruction adds a value to a memory location, and the hardware guarantees that it executes **atomically**; when the instruction executed, it would perform the update as desired. It could not be interrupted mid-instruction, because that is precisely the guarantee we receive from the hardware: when an interrupt occurs, either the instruction has not run at all, or it has run to completion; there is no in-between state. Hardware can be a beautiful thing, no?

Atomically, in this context, means "as a unit", which sometimes we take as "all or none." What we'd like is to execute the three instruction sequence atomically:

```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```

As we said, if we had a single instruction to do this, we could just issue that instruction and be done. But in the general case, we won't have such an instruction. Imagine we were building a concurrent B-tree, and wished to update it; would we really want the hardware to support an "atomic update of B-tree" instruction? Probably not, at least in a sane instruction set.

Thus, what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call **synchronization primitives**. By using these hardware synchronization primitives, in combination with some help from the operating system, we will be able to build multi-threaded code that accesses critical sections in a

TIP: USE ATOMIC OPERATIONS

Atomic operations are one of the most powerful underlying techniques in building computer systems, from the computer architecture, to concurrent code (what we are studying here), to file systems (which we'll study soon enough), database management systems, and even distributed systems [L+93].

The idea behind making a series of actions **atomic** is simply expressed with the phrase "all or nothing"; it should either appear as if all of the actions you wish to group together occurred, or that none of them occurred, with no in-between state visible. Sometimes, the grouping of many actions into a single atomic action is called a **transaction**, an idea developed in great detail in the world of databases and transaction processing [GR92].

In our theme of exploring concurrency, we'll be using synchronization primitives to turn short sequences of instructions into atomic blocks of execution, but the idea of atomicity is much bigger than that, as we will see. For example, file systems use techniques such as journaling or copy-on-write in order to atomically transition their on-disk state, critical for operating correctly in the face of system failures. If that doesn't make sense, don't worry — it will, in some future chapter.

synchronized and controlled manner, and thus reliably produces the correct result despite the challenging nature of concurrent execution. Pretty awesome, right?

This is the problem we will study in this section of the book. It is a wonderful and hard problem, and should make your mind hurt (a bit). If it doesn't, then you don't understand! Keep working until your head hurts; you then know you're headed in the right direction. At that point, take a break; we don't want your head hurting too much.

THE CRUX:
HOW TO PROVIDE SUPPORT FOR SYNCHRONIZATION

What support do we need from the hardware in order to build useful synchronization primitives? What support do we need from the OS? How can we build these primitives correctly and efficiently? How can programs use them to get the desired results?

26.6 One More Problem: Waiting For Another

This chapter has set up the problem of concurrency as if only one type of interaction occurs between threads, that of accessing shared variables and the need to support atomicity for critical sections. As it turns out, there is another common interaction that arises, where one thread must wait for another to complete some action before it continues. This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.

Thus, in the coming chapters, we'll be not only studying how to build support for synchronization primitives to support atomicity but also for mechanisms to support this type of sleeping/waking interaction that is common in multi-threaded programs. If this doesn't make sense right now, that is OK! It will soon enough, when you read the chapter on **condition variables**. If it doesn't by then, well, then it is less OK, and you should read that chapter again (and again) until it does make sense.

26.7 Summary: Why in OS Class?

Before wrapping up, one question that you might have is: why are we studying this in OS class? "History" is the one-word answer; the OS was the first concurrent program, and many techniques were created for use *within* the OS. Later, with multi-threaded processes, application programmers also had to consider such things.

For example, imagine the case where there are two processes running. Assume they both call `write()` to write to the file, and both wish to append the data to the file (i.e., add the data to the end of the file, thus increasing its length). To do so, both must allocate a new block, record in the inode of the file where this block lives, and change the size of the file to reflect the new larger size (among other things; we'll learn more about files in the third part of the book). Because an interrupt may occur at any time, the code that updates these shared structures (e.g., a bitmap for allocation, or the file's inode) are critical sections; thus, OS designers, from the very beginning of the introduction of the interrupt, had to worry about how the OS updates internal structures. An untimely interrupt causes all of the problems described above. Not surprisingly, page tables, process lists, file system structures, and virtually every kernel data structure has to be carefully accessed, with the proper synchronization primitives, to work correctly.

ASIDE: **KEY CONCURRENCY TERMS**
CRITICAL SECTION, RACE CONDITION,
INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

References

[D65] "Solution of a problem in concurrent programming control"
E. W. Dijkstra
Communications of the ACM, 8(9):569, September 1965
Pointed to as the first paper of Dijkstra's where he outlines the mutual exclusion problem and a solution. The solution, however, is not widely used; advanced hardware and OS support is needed, as we will see in the coming chapters.

[D68] "Cooperating sequential processes"
Edsger W. Dijkstra, 1968
Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF
Dijkstra has an amazing number of his old papers, notes, and thoughts recorded (for posterity) on this website at the last place he worked, the University of Texas. Much of his foundational work, however, was done years earlier while he was at the Technische Hochshule of Eindhoven (THE), including this famous paper on "cooperating sequential processes", which basically outlines all of the thinking that has to go into writing multi-threaded programs. Dijkstra discovered much of this while working on an operating system named after his school: the "THE" operating system (said "T", "H", "E", and not like the word "the").

[GR92] "Transaction Processing: Concepts and Techniques"
Jim Gray and Andreas Reuter
Morgan Kaufmann, September 1992
This book is the bible of transaction processing, written by one of the legends of the field, Jim Gray. It is, for this reason, also considered Jim Gray's "brain dump", in which he wrote down everything he knows about how database management systems work. Sadly, Gray passed away tragically a few years back, and many of us lost a friend and great mentor, including the co-authors of said book, who were lucky enough to interact with Gray during their graduate school years.

[L+93] "Atomic Transactions"
Nancy Lynch, Michael Merritt, William Weihl, Alan Fekete
Morgan Kaufmann, August 1993
A nice text on some of the theory and practice of atomic transactions for distributed systems. Perhaps a bit formal for some, but lots of good material is found herein.

[SR05] "Advanced Programming in the UNIX Environment"
W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
As we've said many times, buy this book, and read it, in little chunks, preferably before going to bed. This way, you will actually fall asleep more quickly; more importantly, you learn a little more about how to become a serious UNIX programmer.

Homework

This program, `x86.py`, allows you to see how different thread interleavings either cause or avoid race conditions. See the README for details on how the program works and its basic inputs, then answer the questions below.

Questions

1. To start, let's examine a simple program, "loop.s". First, just look at the program, and see if you can understand it: `cat loop.s`. Then, run it with these arguments:

   ```
   ./x86.py -p loop.s -t 1 -i 100 -R dx
   ```

   This specifies a single thread, an interrupt every 100 instructions, and tracing of register `%dx`. Can you figure out what the value of `%dx` will be during the run? Once you have, run the same above and use the `-c` flag to check your answers; note the answers, on the left, show the value of the register (or memory value) *after* the instruction on the right has run.

2. Now run the same code but with these flags:

   ```
   ./x86.py -p loop.s -t 2 -i 100 -a dx=3,dx=3 -R dx
   ```

   This specifies two threads, and initializes each %dx register to 3. What values will `%dx` see? Run with the `-c` flag to see the answers. Does the presence of multiple threads affect anything about your calculations? Is there a race condition in this code?

3. Now run the following:

   ```
   ./x86.py -p loop.s -t 2 -i 3 -r -a dx=3,dx=3 -R dx
   ```

   This makes the interrupt interval quite small and random; use different seeds with `-s` to see different interleavings. Does the frequency of interruption change anything about this program?

4. Next we'll examine a different program (`looping-race-nolock.s`). This program accesses a shared variable located at memory address 2000; we'll call this variable `x` for simplicity. Run it with a single thread and make sure you understand what it does, like this:

   ```
   ./x86.py -p looping-race-nolock.s -t 1 -M 2000
   ```

   What value is found in `x` (i.e., at memory address 2000) throughout the run? Use `-c` to check your answer.

5. Now run with multiple iterations and threads:

   ```
   ./x86.py -p looping-race-nolock.s -t 2 -a bx=3 -M 2000
   ```

   Do you understand why the code in each thread loops three times? What will the final value of `x` be?

6. Now run with random interrupt intervals:

   ```
   ./x86.py -p looping-race-nolock.s -t 2 -M 2000 -i 4 -r -s 0
   ```

   Then change the random seed, setting `-s 1`, then `-s 2`, etc. Can you tell, just by looking at the thread interleaving, what the final value of `x` will be? Does the exact location of the interrupt matter? Where can it safely occur? Where does an interrupt cause trouble? In other words, where is the critical section exactly?

7. Now use a fixed interrupt interval to explore the program further. Run:

   ```
   ./x86.py -p looping-race-nolock.s -a bx=1 -t 2 -M 2000 -i 1
   ```

   See if you can guess what the final value of the shared variable `x` will be. What about when you change `-i 2`, `-i 3`, etc.? For which interrupt intervals does the program give the "correct" final answer?

8. Now run the same code for more loops (e.g., set `-a bx=100`). What interrupt intervals, set with the `-i` flag, lead to a "correct" outcome? Which intervals lead to surprising results?

9. We'll examine one last program in this homework (`wait-for-me.s`). Run the code like this:

   ```
   ./x86.py -p wait-for-me.s -a ax=1,ax=0 -R ax -M 2000
   ```

   This sets the `%ax` register to 1 for thread 0, and 0 for thread 1, and watches the value of `%ax` and memory location 2000 throughout the run. How should the code behave? How is the value at location 2000 being used by the threads? What will its final value be?

10. Now switch the inputs:

    ```
    ./x86.py -p wait-for-me.s -a ax=0,ax=1 -R ax -M 2000
    ```

    How do the threads behave? What is thread 0 doing? How would changing the interrupt interval (e.g., `-i 1000`, or perhaps to use random intervals) change the trace outcome? Is the program efficiently using the CPU?

27

Interlude: Thread API

This chapter briefly covers the main portions of the thread API. Each part will be explained further in the subsequent chapters, as we show how to use the API. More details can be found in various books and online sources [B89, B97, B+96, K+96]. We should note that the subsequent chapters introduce the concepts of locks and condition variables more slowly, with many examples; this chapter is thus better used as a reference.

CRUX: HOW TO CREATE AND CONTROL THREADS

What interfaces should the OS present for thread creation and control? How should these interfaces be designed to enable ease of use as well as utility?

27.1 Thread Creation

The first thing you have to be able to do to write a multi-threaded program is to create new threads, and thus some kind of thread creation interface must exist. In POSIX, it is easy:

```
#include <pthread.h>
int
pthread_create(      pthread_t *        thread,
               const pthread_attr_t *  attr,
                     void *             (*start_routine)(void*),
                     void *             arg);
```

This declaration might look a little complex (particularly if you haven't used function pointers in C), but actually it's not too bad. There are four arguments: `thread`, `attr`, `start_routine`, and `arg`. The first, `thread`, is a pointer to a structure of type `pthread_t`; we'll use this structure to interact with this thread, and thus we need to pass it to `pthread_create()` in order to initialize it.

The second argument, `attr`, is used to specify any attributes this thread might have. Some examples include setting the stack size or perhaps information about the scheduling priority of the thread. An attribute is initialized with a separate call to `pthread_attr_init()`; see the manual page for details. However, in most cases, the defaults will be fine; in this case, we will simply pass the value `NULL` in.

The third argument is the most complex, but is really just asking: which function should this thread start running in? In C, we call this a **function pointer**, and this one tells us the following is expected: a function name (`start_routine`), which is passed a single argument of type `void *` (as indicated in the parentheses after `start_routine`), and which returns a value of type `void *` (i.e., a **void pointer**).

If this routine instead required an integer argument, instead of a void pointer, the declaration would look like this:

```
int pthread_create(..., // first two args are the same
                   void *    (*start_routine)(int),
                   int       arg);
```

If instead the routine took a void pointer as an argument, but returned an integer, it would look like this:

```
int pthread_create(..., // first two args are the same
                   int       (*start_routine)(void *),
                   void *     arg);
```

Finally, the fourth argument, `arg`, is exactly the argument to be passed to the function where the thread begins execution. You might ask: why do we need these void pointers? Well, the answer is quite simple: having a void pointer as an argument to the function `start_routine` allows us to pass in *any* type of argument; having it as a return value allows the thread to return *any* type of result.

Let's look at an example in Figure 27.1. Here we just create a thread that is passed two arguments, packaged into a single type we define ourselves (`myarg_t`). The thread, once created, can simply cast its argument to the type it expects and thus unpack the arguments as desired.

And there it is! Once you create a thread, you really have another live executing entity, complete with its own call stack, running within the *same* address space as all the currently existing threads in the program. The fun thus begins!

27.2 Thread Completion

The example above shows how to create a thread. However, what happens if you want to wait for a thread to complete? You need to do something special in order to wait for completion; in particular, you must call the routine `pthread_join()`.

```
int pthread_join(pthread_t thread, void **value_ptr);
```

```
#include <pthread.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p;
    int rc;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Figure 27.1: **Creating a Thread**

This routine takes two arguments. The first is of type `pthread_t`, and is used to specify which thread to wait for. This variable is initialized by the thread creation routine (when you pass a pointer to it as an argument to `pthread_create()`); if you keep it around, you can use it to wait for that thread to terminate.

The second argument is a pointer to the return value you expect to get back. Because the routine can return anything, it is defined to return a pointer to void; because the `pthread_join()` routine *changes* the value of the passed in argument, you need to pass in a pointer to that value, not just the value itself.

Let's look at another example (Figure 27.2). In the code, a single thread is again created, and passed a couple of arguments via the `myarg_t` structure. To return values, the `myret_t` type is used. Once the thread is finished running, the main thread, which has been waiting inside of the `pthread_join()` routine[1], then returns, and we can access the values returned from the thread, namely whatever is in `myret_t`.

A few things to note about this example. First, often times we don't have to do all of this painful packing and unpacking of arguments. For example, if we just create a thread with no arguments, we can pass `NULL` in as an argument when the thread is created. Similarly, we can pass `NULL` into `pthread_join()` if we don't care about the return value.

Second, if we are just passing in a single value (e.g., an int), we don't

[1]Note we use wrapper functions here; specifically, we call Malloc(), Pthread_join(), and Pthread_create(), which just call their similarly-named lower-case versions and make sure the routines did not return anything unexpected.

```
#include <stdio.h>
#include <pthread.h>
#include <assert.h>
#include <stdlib.h>

typedef struct __myarg_t {
    int a;
    int b;
} myarg_t;

typedef struct __myret_t {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t *r = Malloc(sizeof(myret_t));
    r->x = 1;
    r->y = 2;
    return (void *) r;
}

int
main(int argc, char *argv[]) {
    int rc;
    pthread_t p;
    myret_t *m;

    myarg_t args;
    args.a = 10;
    args.b = 20;
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &m);
    printf("returned %d %d\n", m->x, m->y);
    return 0;
}
```

Figure 27.2: **Waiting for Thread Completion**

have to package it up as an argument. Figure 27.3 shows an example. In this case, life is a bit simpler, as we don't have to package arguments and return values inside of structures.

Third, we should note that one has to be extremely careful with how values are returned from a thread. In particular, never return a pointer which refers to something allocated on the thread's call stack. If you do, what do you think will happen? (think about it!) Here is an example of a dangerous piece of code, modified from the example in Figure 27.2.

```
void *mythread(void *arg) {
    myarg_t *m = (myarg_t *) arg;
    printf("%d %d\n", m->a, m->b);
    myret_t r; // ALLOCATED ON STACK: BAD!
    r.x = 1;
    r.y = 2;
    return (void *) &r;
}
```

```
void *mythread(void *arg) {
    int m = (int) arg;
    printf("%d\n", m);
    return (void *) (arg + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    int rc, m;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &m);
    printf("returned %d\n", m);
    return 0;
}
```

Figure 27.3: **Simpler Argument Passing to a Thread**

In this case, the variable `r` is allocated on the stack of `mythread`. However, when it returns, the value is automatically deallocated (that's why the stack is so easy to use, after all!), and thus, passing back a pointer to a now deallocated variable will lead to all sorts of bad results. Certainly, when you print out the values you think you returned, you'll probably (but not necessarily!) be surprised. Try it and find out for yourself[2]!

Finally, you might notice that the use of `pthread_create()` to create a thread, followed by an immediate call to `pthread_join()`, is a pretty strange way to create a thread. In fact, there is an easier way to accomplish this exact task; it's called a **procedure call**. Clearly, we'll usually be creating more than just one thread and waiting for it to complete, otherwise there is not much purpose to using threads at all.

We should note that not all code that is multi-threaded uses the join routine. For example, a multi-threaded web server might create a number of worker threads, and then use the main thread to accept requests and pass them to the workers, indefinitely. Such long-lived programs thus may not need to join. However, a parallel program that creates threads to execute a particular task (in parallel) will likely use join to make sure all such work completes before exiting or moving onto the next stage of computation.

27.3 Locks

Beyond thread creation and join, probably the next most useful set of functions provided by the POSIX threads library are those for providing mutual exclusion to a critical section via **locks**. The most basic pair of routines to use for this purpose is provided by this pair of routines:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

[2]Fortunately the compiler `gcc` will likely complain when you write code like this, which is yet another reason to pay attention to compiler warnings.

The routines should be easy to understand and use. When you have a region of code you realize is a **critical section**, and thus needs to be protected by locks in order to operate as desired. You can probably imagine what the code looks like:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

The intent of the code is as follows: if no other thread holds the lock when `pthread_mutex_lock()` is called, the thread will acquire the lock and enter the critical section. If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock (implying that the thread holding the lock has released it via the unlock call). Of course, many threads may be stuck waiting inside the lock acquisition function at a given time; only the thread with the lock acquired, however, should call unlock.

Unfortunately, this code is broken, in two important ways. The first problem is a **lack of proper initialization**. All locks must be properly initialized in order to guarantee that they have the correct values to begin with and thus work as desired when lock and unlock are called.

With POSIX threads, there are two ways to initialize locks. One way to do this is to use `PTHREAD_MUTEX_INITIALIZER`, as follows:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Doing so sets the lock to the default values and thus makes the lock usable. The dynamic way to do it (i.e., at run time) is to make a call to `pthread_mutex_init()`, as follows:

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

The first argument to this routine is the address of the lock itself, whereas the second is an optional set of attributes. Read more about the attributes yourself; passing `NULL` in simply uses the defaults. Either way works, but we usually use the dynamic (latter) method. Note that a corresponding call to `pthread_mutex_destroy()` should also be made, when you are done with the lock; see the manual page for all of details.

The second problem with the code above is that it fails to check error codes when calling lock and unlock. Just like virtually any library routine you call in a UNIX system, these routines can also fail! If your code doesn't properly check error codes, the failure will happen silently, which in this case could allow multiple threads into a critical section. Minimally, use wrappers, which assert that the routine succeeded (e.g., as in Figure 27.4); more sophisticated (non-toy) programs, which can't simply exit when something goes wrong, should check for failure and do something appropriate when the lock or unlock does not succeed.

```
// Use this to keep your code clean but check for failures
// Only use if exiting program is OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
  int rc = pthread_mutex_lock(mutex);
  assert(rc == 0);
}
```

Figure 27.4: **An Example Wrapper**

The lock and unlock routines are not the only routines within the pthreads library to interact with locks. In particular, here are two more routines which may be of interest:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

These two calls are used in lock acquisition. The `trylock` version returns failure if the lock is already held; the `timedlock` version of acquiring a lock returns after a timeout or after acquiring the lock, whichever happens first. Thus, the timedlock with a timeout of zero degenerates to the trylock case. Both of these versions should generally be avoided; however, there are a few cases where avoiding getting stuck (perhaps indefinitely) in a lock acquisition routine can be useful, as we'll see in future chapters (e.g., when we study deadlock).

27.4 Condition Variables

The other major component of any threads library, and certainly the case with POSIX threads, is the presence of a **condition variable**. Condition variables are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue. Two primary routines are used by programs wishing to interact in this way:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

To use a condition variable, one has to in addition have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.

The first routine, `pthread_cond_wait()`, puts the calling thread to sleep, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about. A typical usage looks like this:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

In this code, after initialization of the relevant lock and condition[3], a thread checks to see if the variable `ready` has yet been set to something other than zero. If not, the thread simply calls the wait routine in order to sleep until some other thread wakes it.

The code to wake a thread, which would run in some other thread, looks like this:

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

A few things to note about this code sequence. First, when signaling (as well as when modifying the global variable `ready`), we always make sure to have the lock held. This ensures that we don't accidentally introduce a race condition into our code.

Second, you might notice that the wait call takes a lock as its second parameter, whereas the signal call only takes a condition. The reason for this difference is that the wait call, in addition to putting the calling thread to sleep, *releases* the lock when putting said caller to sleep. Imagine if it did not: how could the other thread acquire the lock and signal it to wake up? However, *before* returning after being woken, the `pthread_cond_wait()` re-acquires the lock, thus ensuring that any time the waiting thread is running between the lock acquire at the beginning of the wait sequence, and the lock release at the end, it holds the lock.

One last oddity: the waiting thread re-checks the condition in a while loop, instead of a simple if statement. We'll discuss this issue in detail when we study condition variables in a future chapter, but in general, using a while loop is the simple and safe thing to do. Although it rechecks the condition (perhaps adding a little overhead), there are some pthread implementations that could spuriously wake up a waiting thread; in such a case, without rechecking, the waiting thread will continue thinking that the condition has changed even though it has not. It is safer thus to view waking up as a hint that something might have changed, rather than an absolute fact.

Note that sometimes it is tempting to use a simple flag to signal between two threads, instead of a condition variable and associated lock. For example, we could rewrite the waiting code above to look more like this in the waiting code:

```
while (ready == 0)
    ; // spin
```

The associated signaling code would look like this:

```
ready = 1;
```

[3]Note that one could use `pthread_cond_init()` (and corresponding the `pthread_cond_destroy()` call) instead of the static initializer `PTHREAD_COND_INITIALIZER`. Sound like more work? It is.

Don't ever do this, for the following reasons. First, it performs poorly in many cases (spinning for a long time just wastes CPU cycles). Second, it is error prone. As recent research shows [X+10], it is surprisingly easy to make mistakes when using flags (as above) to synchronize between threads; in that study, roughly half the uses of these *ad hoc* synchronizations were buggy! Don't be lazy; use condition variables even when you think you can get away without doing so.

If condition variables sound confusing, don't worry too much (yet) – we'll be covering them in great detail in a subsequent chapter. Until then, it should suffice to know that they exist and to have some idea how and why they are used.

27.5 Compiling and Running

All of the code examples in this chapter are relatively easy to get up and running. To compile them, you must include the header `pthread.h` in your code. On the link line, you must also explicitly link with the pthreads library, by adding the `-pthread` flag.

For example, to compile a simple multi-threaded program, all you have to do is the following:

```
prompt> gcc -o main main.c -Wall -pthread
```

As long as `main.c` includes the pthreads header, you have now successfully compiled a concurrent program. Whether it works or not, as usual, is a different matter entirely.

27.6 Summary

We have introduced the basics of the pthread library, including thread creation, building mutual exclusion via locks, and signaling and waiting via condition variables. You don't need much else to write robust and efficient multi-threaded code, except patience and a great deal of care!

We now end the chapter with a set of tips that might be useful to you when you write multi-threaded code (see the aside on the following page for details). There are other aspects of the API that are interesting; if you want more information, type `man -k pthread` on a Linux system to see over one hundred APIs that make up the entire interface. However, the basics discussed herein should enable you to build sophisticated (and hopefully, correct and performant) multi-threaded programs. The hard part with threads is not the APIs, but rather the tricky logic of how you build concurrent programs. Read on to learn more.

ASIDE: **THREAD API GUIDELINES**

There are a number of small but important things to remember when you use the POSIX thread library (or really, any thread library) to build a multi-threaded program. They are:

- **Keep it simple.** Above all else, any code to lock or signal between threads should be as simple as possible. Tricky thread interactions lead to bugs.
- **Minimize thread interactions.** Try to keep the number of ways in which threads interact to a minimum. Each interaction should be carefully thought out and constructed with tried and true approaches (many of which we will learn about in the coming chapters).
- **Initialize locks and condition variables.** Failure to do so will lead to code that sometimes works and sometimes fails in very strange ways.
- **Check your return codes.** Of course, in any C and UNIX programming you do, you should be checking each and every return code, and it's true here as well. Failure to do so will lead to bizarre and hard to understand behavior, making you likely to (a) scream, (b) pull some of your hair out, or (c) both.
- **Be careful with how you pass arguments to, and return values from, threads.** In particular, any time you are passing a reference to a variable allocated on the stack, you are probably doing something wrong.
- **Each thread has its own stack.** As related to the point above, please remember that each thread has its own stack. Thus, if you have a locally-allocated variable inside of some function a thread is executing, it is essentially *private* to that thread; no other thread can (easily) access it. To share data between threads, the values must be in the **heap** or otherwise some locale that is globally accessible.
- **Always use condition variables to signal between threads.** While it is often tempting to use a simple flag, don't do it.
- **Use the manual pages.** On Linux, in particular, the pthread man pages are highly informative and discuss much of the nuances presented here, often in even more detail. Read them carefully!

References

[B89] "An Introduction to Programming with Threads"
Andrew D. Birrell
DEC Technical Report,January, 1989
Available: https://birrell.org/andrew/papers/035-Threads.pdf
A classic but older introduction to threaded programming. Still a worthwhile read, and freely available.

[B97] "Programming with POSIX Threads"
David R. Butenhof
Addison-Wesley, May 1997
Another one of these books on threads.

[B+96] "PThreads Programming:
A POSIX Standard for Better Multiprocessing"
Dick Buttlar, Jacqueline Farrell, Bradford Nichols
O'Reilly, September 1996
A reasonable book from the excellent, practical publishing house O'Reilly. Our bookshelves certainly contain a great deal of books from this company, including some excellent offerings on Perl, Python, and Javascript (particularly Crockford's "Javascript: The Good Parts".)

[K+96] "Programming With Threads"
Steve Kleiman, Devang Shah, Bart Smaalders
Prentice Hall, January 1996
Probably one of the better books in this space. Get it at your local library. Or steal it from your mother. More seriously, just ask your mother for it – she'll let you borrow it, don't worry.

[X+10] "Ad Hoc Synchronization Considered Harmful"
Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, Zhiqiang Ma
OSDI 2010, Vancouver, Canada
This paper shows how seemingly simple synchronization code can lead to a surprising number of bugs. Use condition variables and do the signaling correctly!

Homework (Code)

In this section, we'll write some simple multi-threaded programs and use a specific tool, called **helgrind**, to find problems in these programs.

Read the README in the homework download for details on how to build the programs and run `helgrind`.

Questions

1. First build `main-race.c`. Examine the code so you can see the (hopefully obvious) data race in the code. Now run `helgrind` (by typing `valgrind --tool=helgrind main-race`) to see how it reports the race. Does it point to the right lines of code? What other information does it give to you?

2. What happens when you remove one of the offending lines of code? Now add a lock around one of the updates to the shared variable, and then around both. What does `helgrind` report in each of these cases?

3. Now let's look at `main-deadlock.c`. Examine the code. This code has a problem known as **deadlock** (which we discuss in much more depth in a forthcoming chapter). Can you see what problem it might have?

4. Now run `helgrind` on this code. What does `helgrind` report?

5. Now run `helgrind` on `main-deadlock-global.c`. Examine the code; does it have the same problem that `main-deadlock.c` has? Should `helgrind` be reporting the same error? What does this tell you about tools like `helgrind`?

6. Let's next look at `main-signal.c`. This code uses a variable (`done`) to signal that the child is done and that the parent can now continue. Why is this code inefficient? (what does the parent end up spending its time doing, particularly if the child thread takes a long time to complete?)

7. Now run `helgrind` on this program. What does it report? Is the code correct?

8. Now look at a slightly modified version of the code, which is found in `main-signal-cv.c`. This version uses a condition variable to do the signaling (and associated lock). Why is this code preferred to the previous version? Is it correctness, or performance, or both?

9. Once again run `helgrind` on `main-signal-cv`. Does it report any errors?

User-Mode Thread Implementation

Introduction

For a long time, processes were the only unit of parallel computation. There are two problems with this:

- processes are very expensive to create and dispatch, due to the fact that each has its own virtual address space and ownership of numerous system resources.
- each process operates in its own address space, and cannot share in-memory resources with parallel processes (this was before it was possible to map shared segments into a process' address space).

The answer to these needs was to create threads. A thread ...

- is an independently schedulable unit of execution.
- runs within the address space of a process.
- has access to all of the system resources owned by that process.
- has its own general registers.
- has its own stack (within the owning process' address space).

Threads were added to Unix/Linux as an after-thought. In Unix, a process could be scheduled for execution, and could create threads for additional parallelism. Windows NT is a newer operating system, and it was designed with threads from the start ... and so the abstractions are cleaner:

- a process is a container for an address space and resources.
- a thread is **the** unit of scheduled execution.

A Simple Threads Library

When threads were first added to Linux they were entirely implemented in a user-mode library, with no assistance from the operating system. This is not merely historical trivia, but interesting as an examination of what kinds of problems can and cannot be solved without operating system assistance.

The basic model is:

- Each time a new thread was created:
  - we allocate memory for a (fixed size) thread-private stack from the heap.
  - we create a new thread descriptor that contains identification information, scheduling information, and a pointer to the stack.
  - we add the new thread to a ready queue.
- When a thread calls *yield()* or *sleep()* we save its general registers (on its own stack), and then select the next thread on the ready queue.
- To dispatch a new thread, we simply restore its saved registers (including the stack pointer), and return from the call that caused it to *yield*.
- If a thread called *sleep()* we would remove it from the ready queue. When it was re-awakened, we would put it back onto the ready queue.
- When a thread exited, we would free its stack and thread descriptor.

Eventually people wanted preemptive scheduling to ensure good interactive response and prevent buggy threads from tieing up the application:

- Linux processes can schedule the delivery of (SIGALARM) timer signals and register a handler for them.

- Before dispatching a thread, the we can schedule a SIGALARM that will interrupt the thread if it runs too long.
- If a thread runs too long, the SIGALARM handler can *yield* on behalf of that thread, saving its state, moving on to the next thread in the ready queue.

But the addition of preemptive scheduling created new problems for critical sections that required before-or-after, all-or-none serialization. Fortunately Linux processes can temporarily block signals (much as it is possible to temporarily disable an interrupt) via the *sigprocmask(2)* system call.

Kernel implemented threads

There are two fundamental problems with implementing threads in a user mode library:

- what happens when a system call blocks

  If a user-mode thread issues a system call that blocks (e.g. *open* or *read*), the process is blocked until that operation completes. This means that when a thread blocks, all threads (within that process) stop executing. Since the threads were implemented in user-mode, the operating system has no knowledge of them, and cannot know that other threads (in that process) might still be runnable.

- exploiting multi-processors

  If the CPU has multiple execution cores, the operating system can schedule processes on each to run in parallel. But if the operating system is not aware that a process is comprised of multiple threads, those threads cannot execute in parallel on the available cores.

Both of these problems are solved if threads are implemented by the operating system rather than by a user-mode thread library.

Performance Implications

If non-preemptive scheduling can be used, user-mode threads operating in with a *sleep*/*yield* model are much more efficient than doing context switches through the operating system. There are, today, *light weight thread* implementations to reap these benefits.

If preemptive scheduling is to be used, the costs of setting alarms and servicing the signals may well be greater than the cost of simply allowing the operating system to do the scheduling.

If the threads can run in parallel on a multi-processor, the added throughput resulting from true parallel execution may be far greater than the efficiency losses associated with more expensive context switches through the operating system. Also, the operating system knows which threads are part of the same process, and may be able to schedule them to maximize cache-line sharing.

Like preemptive scheduling, the signal disabling and reenabling for a user-mode mutex or condition variable implementation may be more expensive than simply using the kernel-mode implementations. But it may be possible to get the best of both worlds with a user-mode implementation that uses an atomic instruction to attempt to get a lock, but calls the operating system if that allocation fails (somewhat like the *futex(7)* approach).

0