Advanced Programming in the UNIX Environment — *Advanced I/O*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

Table of Contents

Introduction

- This chapter covers numerous topics and functions that we lump under the term **advanced I/O**:
  - nonblocking I/O,
  - record locking,
  - System V STREAMS,
  - I/O multiplexing (the `select` and `poll` functions),
  - the `readv` and `writev` functions,
  - and memory-mapped I/O (`mmap`).

Nonblocking I/O I

- In previous chapter, we said that the system calls are divided into two categories: the "slow" ones and all the others. The slow system calls are those that can block forever. They include
  - Reads that can block the caller forever if data isn't present with certain file types (pipes, terminal devices, and network devices)
  - Writes that can block the caller forever if the data can't be accepted immediately by these same file types (no room in the pipe, network flow control, etc.)
  - Opens that block until some condition occurs on certain file types
  - Reads and writes of files that have mandatory record locking enabled
  - Certain `ioctl` operations
  - Some of the IPC functions

Nonblocking I/O II

- We also said that system calls related to disk I/O are not considered slow, even though the read or write of a disk file can block the caller temporarily.
- Nonblocking I/O lets us issue an I/O operation and not have it block forever.
- There are two ways to specify nonblocking I/O for a given descriptor:
  1. Call `open` with the `O_NONBLOCK` flag (Section 3.3).
  2. For a descriptor that is already open, we call `fcntl` to turn on the `O_NONBLOCK` file status flag (Section 3.14).
- Example 12.1

Record Locking

- Omitted.

STREAMS I

- The **STREAMS** mechanism is provided by System V as a general way to interface communication drivers into the kernel.
- We need to discuss STREAMS to understand the terminal interface in System V, the use of the `poll` function for I/O multiplexing, and the implementation of STREAMS-based pipes and named pipes.
- A stream provides a full-duplex path between a user process and a device driver. There is no need for a stream to talk to a hardware device.

STREAMS II

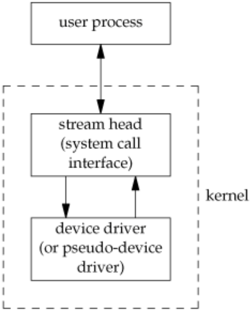
user process
stream head (system call interface)
device driver (or pseudo-device driver)
kernel

Figure: A simple stream

- Beneath the stream head, we can push processing modules onto the stream. This is done using an `ioctl` command.

STREAMS III

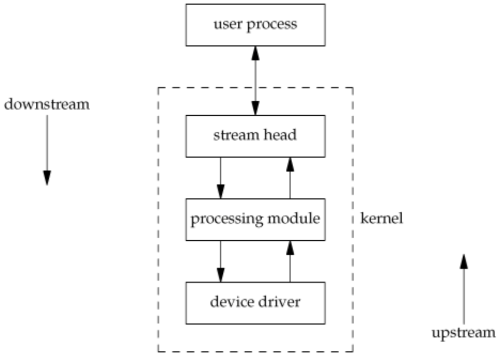

Figure: A simple stream

STREAMS IV

- Any number of processing modules can be *pushed* onto a stream.
- STREAMS modules are similar to device drivers in that they execute as part of the kernel, and they are normally link edited into the kernel when the kernel is built.
- We access a stream with the functions from Chapter 3: `open`, `close`, `read`, `write`, and `ioctl`. Additionally, three new functions were added to the SVR3 kernel to support STREAMS (`getmsg`, `putmsg`, and `poll`), and another two (`getpmsg` and `putpmsg`) were added with SVR4 to handle messages with different priority bands within a stream.

STREAMS Messages I

- All input and output under STREAMS is based on messages.
- The stream head and the user process exchange messages using `read`, `write`, `ioctl`, `getmsg`, `getpmsg`, `putmsg`, and `putpmsg`.
- Messages are also passed up and down a stream between the stream head, the processing modules, and the device driver.
- Between the user process and the stream head, a message consists of a message type, optional control information, and optional data.
- The control information and data are specified by `strbuf` structures:

STREAMS Messages II

```
struct strbuf{
  int maxlen;  /* size of buffer */
  int len;     /* number of bytes currently in buffer */
  char *buf;   /* pointer to buffer */
};
```

- There are about 25 different types of messages, but only a few of these are used between the user process and the stream head. The rest are passed up and down a stream within the kernel.
- We'll encounter only three of these message types with the functions we use:
  1. M_DATA (user data for I/O)
  2. M_PROTO (protocol control information)
  3. M_PCPROTO (high-priority protocol control information)

STREAMS Messages III

- Every message on a stream has a queueing priority:
  - High-priority messages (highest priority)
  - Priority band messages
  - Ordinary messages (lowest priority)
- Each STREAMS module has two input queues. One receives messages from the module above, and one receives messages from the module below. The messages on an input queue are arranged by priority.

putmsg and putpmsg Functions I

- A STREAMS message (control information or data, or both) is written to a stream using either putmsg or putpmsg. The difference between these two functions is that the latter allows us to specify a priority band for the message.

```
#include <stropts.h>
int putmsg(int filedes, const struct strbuf *ctlptr,
           const struct strbuf *dataptr, int flag);
int putpmsg(int filedes, const struct strbuf *ctlptr,
            const struct strbuf *dataptr, int band,
            int flag);
```

- We can also write to a stream, which is equivalent to a putmsg without any control information and with a *flag* of 0.
- These two functions can generate the three different priorities of messages: ordinary, priority band, and high priority.

STREAMS `ioctl` Operations

- Between Linux and Solaris, there are almost 40 different operations that can be performed on a stream using `ioctl`. Most of these operations are documented in the `streamio`(7) manual page.
- Example: `isastream` function.
- Example 12.10: List the names of the modules on a stream

getmsg and getpmsg Functions

- STREAMS messages are read from a stream head using read, getmsg, or getpmsg.

```
#include <stropts.h>
int getmsg(int filedes, struct strbuf *restrict ctlptr,
           struct strbuf *restrict dataptr,
           int *restrict flagptr);
int getpmsg(int filedes, struct strbuf *restrict ctlptr,
            struct strbuf *restrict dataptr,
            int *restrict bandptr,
            int *restrict flagptr);
```

- Example 14.19. Copy standard input to standard output using getmsg.

I/O Multiplexing I

- When we read from one descriptor and write to another, we can use blocking I/O in a loop.
- What if we have to read from two descriptors? In this case, we can't do a blocking `read` on either descriptor, as data may appear on one descriptor while we're blocked in a `read` on the other. A different technique is required to handle this case.
- Let's look at the structure of the `telnet`(1) command.

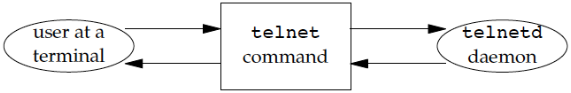

Figure: Overview of `telnet` program

I/O Multiplexing II

- The `telnet` process has two inputs and two outputs. We can't do a blocking `read` on either of the inputs, as we never know which input will have data for us.
- One way to handle this particular problem is to divide the process in two pieces (using `fork`), with each half handling one direction of data.

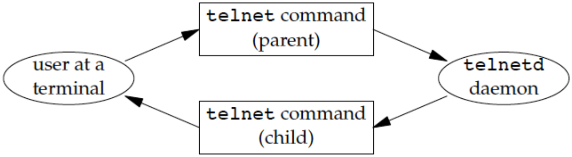

Figure: The `telnet` program using two processes

I/O Multiplexing III

- But this leads to a problem when the operation terminates: it does complicate the program somewhat.
- Instead of two processes, we could use two threads in a single process. This avoids the termination complexity, but requires that we deal with synchronization between the threads, which could add more complexity than it saves.
- We could use nonblocking I/O in a single process by setting both descriptors nonblocking and adopting poll mechanism. The problem is that it wastes CPU time.
- Although it works on any system that supports nonblocking I/O, polling should be avoided on a multitasking system.

I/O Multiplexing IV

- Another technique is called **asynchronous I/O**. To do this, we tell the kernel to notify us with a signal when a descriptor is ready for I/O. There are two problems with this. First, not all systems support this feature. Second, there is only one of these signals per process.
- A better technique is to use **I/O multiplexing**. To do this, we build a list of the descriptors that we are interested in and call a function that doesn't return until one of the descriptors is ready for I/O. On return from the function, we are told which descriptors are ready for I/O.
- Three functions `poll`, `pselect`, and `select` allow us to perform I/O multiplexing.

select and pselect Functions I

- The select function lets us do I/O multiplexing under all POSIX-compatible platforms. The arguments we pass to select tell the kernel
  - Which descriptors we're interested in.
  - What conditions we're interested in for each descriptor.
  - How long we want to wait.
- On the return from select, the kernel tells us
  - The total count of the number of descriptors that are ready
  - Which descriptors are ready for each of the three conditions
- With this return information, we can call the appropriate I/O function and know that the function won't block.

select and pselect Functions II

```c
#include <sys/select.h>
int select(int maxfdp1, fd_set *restrict readfds,
           fd_set *restrict writefds,
           fd_set *restrict exceptfds,
           struct timeval *restrict tvptr);
struct timeval{
  long tv_sec;   /* seconds */
  long tv_usec;  /* and microseconds */
};
```

- First parameter *maxfdp1* is *maximum fd plus 1*, indicates the number of fd we want check.
- The middle three arguments *readfds*, *writefds*, and *exceptfds* are pointers to **descriptor sets**.

select and pselect Functions III

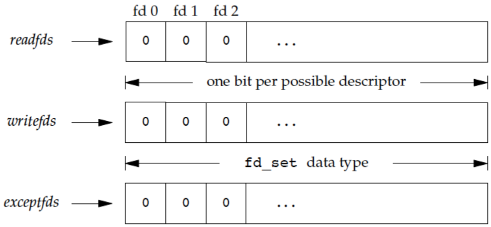

Figure: Specifying the read, write, and exception descriptors for `select`

- The only thing we can do with the `fd_set` data type is allocate a variable of this type, assign a variable of this type to another variable of the same type, or use one of the following four functions on a variable of this type.

select and pselect Functions IV

```c
#include <sys/select.h>
int FD_ISSET(int fd, fd_set *fdset);
void FD_CLR(int fd, fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
void FD_ZERO(fd_set *fdset);
```

- These interfaces can be implemented as either macros or functions.
- There are three possible return values from `select`.
  1. A return value of -1 means that an error occurred. In this case, none of the descriptor sets will be modified.

select and pselect Functions V

2. A return value of 0 means that no descriptors are ready. This happens if the time limit expires before any of the descriptors are ready. When this happens, all the descriptor sets will be zeroed out.
3. A positive return value specifies the number of descriptors that are ready. This value is the sum of the descriptors ready in all three sets, so if the same descriptor is ready to be read and written, it will be counted twice in the return value. The only bits left on in the three descriptor sets are the bits corresponding to the descriptors that are ready.

readv and writev Functions I

- The readv and writev functions let us read into and write from multiple noncontiguous buffers in a single function call. These operations are called **scatter read** and **gather write**.

```
#include <sys/uio.h>
ssize_t  readv(int filedes, const struct iovec *iov,
               int iovcnt);
ssize_t writev(int filedes, const struct iovec *iov,
               int iovcnt);
```

- The second argument to both functions is a pointer to an array of iovec structures:

```
struct iovec{
  void    *iov_base;  /* starting address of buffer */
  size_t  iov_len;    /* size of buffer */
};
```

readv and writev Functions II

- The number of elements in the *iov* array is specified by *iovcnt*. It is limited to IOV_MAX.

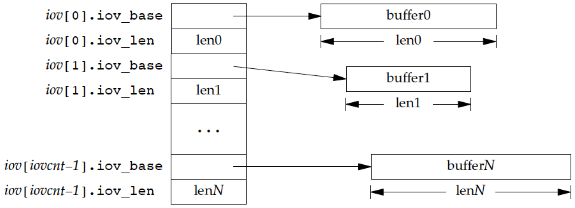

Figure: The iovec structure for readv and writev

readv and writev Functions III

- The writev function gathers the output data from the buffers in order: *iov*[0], *iov*[1], through *iov*[*iovcnt*1]; writev returns the total number of bytes output, which should normally equal the sum of all the buffer lengths.
- The readv function scatters the data into the buffers in order, always filling one buffer before proceeding to the next. readv returns the total number of bytes that were read. A count of 0 is returned if there is no more data and the end of file is encountered.
- Example. As we expect, the system time increases when we call write twice, compared to calling either write or writev once.

Memory-Mapped I/O I

- **Memory-mapped I/O** lets us map a file on disk into a buffer in memory so that, when we fetch bytes from the buffer, the corresponding bytes of the file are read. Similarly, when we store data in the buffer, the corresponding bytes are automatically written to the file. This lets us perform I/O without using `read` or `write`.
- To use this feature, we have to tell the kernel to map a given file to a region in memory. This is done by the `mmap` function.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot,
           int flag, int filedes, off_t off);
```

Memory-Mapped I/O II

- The return value of this function is the starting address of the mapped area.
- The *addr* argument lets us specify the address of where we want the mapped region to start. We normally set this to 0 to allow the system to choose the starting address.
- The *filedes* argument is the file descriptor specifying the file that is to be mapped. We have to open this file before we can map it into the address space.
- The *len* argument is the number of bytes to map, and *off* is the starting offset in the file of the bytes to map.
- The *prot* argument specifies the protection of the mapped region.

Memory-Mapped I/O III

- We can specify the protection as either `PROT_NONE` or the bitwise OR of any combination of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`.
- The protection specified for a region can't allow more access than the open mode of the file.

Memory-Mapped I/O IV

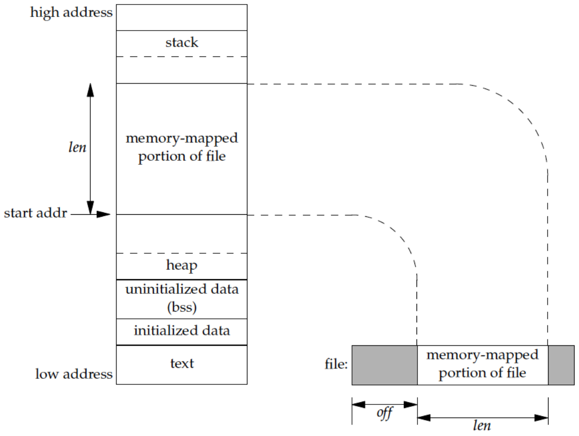

Figure: Example of a memory-mapped file

Memory-Mapped I/O V

- The *flag* argument affects various attributes of the mapped region.

`MAP_FIXED`	The return value must equal *addr*. Use of this flag is discouraged, as it hinders portability. If this flag is not specified and if *addr* is nonzero, then the kernel uses *addr* as a hint of where to place the mapped region, but there is no guarantee that the requested address will be used. Maximum portability is obtained by specifying *addr* as 0.

Memory-Mapped I/O VI

`MAP_SHARED` This flag describes the disposition of store operations into the mapped region by this process. This flag specifies that store operations modify the mapped file that is, a store operation is equivalent to a `write` to the file. Either this flag or the next (`MAP_PRIVATE`), but not both, must be specified.

`MAP_PRIVATE` This flag says that store operations into the mapped region cause a private copy of the mapped file to be created. All successive references to the mapped region then reference the copy.

Memory-Mapped I/O VII

- The value of *off* and the value of *addr* (if `MAP_FIXED` is specified) are required to be multiples of the system's virtual memory page size.
- We cannot append to a file with `mmap`. We must first grow the file.
- A memory-mapped region is inherited by a child across a `fork`, but is not inherited by the new program across an `exec`.
- Example.
- We can change the permissions on an existing mapping by calling `mprotect`.

```c
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
```

Memory-Mapped I/O VIII

- If the pages in a shared mapping have been modified, we can call msync to flush the changes to the file that backs the mapping. The msync function is similar to fsync, but works on memory-mapped regions.

```
#include <sys/mman.h>
int msync(void *addr, size_t len, int flags);
```

- A memory-mapped region is automatically unmapped when the process terminates or by calling munmap directly. Closing the file descriptor *filedes* does **NOT** unmap the region.

```
#include <sys/mman.h>
int munmap(caddr_t addr, size_t len);
```

Memory-Mapped I/O IX

- The call to `munmap` does not cause the contents of the mapped region to be written to the disk file.
- Memory-mapped I/O is faster when copying one regular file to another. There are limitations. We can't use it to copy between certain devices (such as a network device or a terminal device), and we have to be careful if the size of the underlying file could change after we map it.

The End

The End of Chapter 14.