CS631 - Advanced Programming in the UNIX Environment
–
Advanced I/O / HTTP

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@stevens.edu
http://www.cs.stevens.edu/~jschauma/631/

Nonblocking I/O

Recall from our lecture on signals that certain system calls can block forever:

- `read(2)` from a particular file, if data isn't present (pipes, terminals, network devices)
- `write(2)` to the same kind of file
- `open(2)` of a particular file until a specific condition occurs
- `read(2)` and `write(2)` of files that have mandatory locking enabled
- certain `ioctls(2)`
- some IPC functions (such as `sendto(2)` or `recv(2)`)

See `eintr.c` from that lecture.

Nonblocking I/O

Recall from our lecture on signals that certain system calls can block forever:

- `read(2)` from a particular file, if data isn't present (pipes, terminals, network devices)
- `write(2)` to the same kind of file
- `open(2)` of a particular file until a specific condition occurs
- `read(2)` and `write(2)` of files that have mandatory locking enabled
- certain `ioctls(2)`
- some IPC functions (such as `sendto(2)` or `recv(2)`)

Nonblocking I/O lets us issue an I/O operation and not have it block forever. If the operation cannot be completed, return is made immediately with an error noting that the operating would have blocked (`EWOULDBLOCK` or `EAGAIN`).

Nonblocking I/O

Ways to specify nonblocking mode:

- pass `O_NONBLOCK` to `open(2)`:

```
open(path, O_RDRW|O_NONBLOCK);
```

- set `O_NONBLOCK` via `fcntl(2)`:

```
flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags|O_NONBLOCK);
```

Nonblocking I/O

```
$ cc -Wall nonblock.c -o block
$ cc -DNONBLOCK -Wall nonblock.c -o nonblock
$ ./nonblock >/dev/null
wrote 100000 bytes
[...]
$ ./block | ( sleep 3; cat >/dev/null )
[...]
$ ./nonblock | ( sleep 3; cat >/dev/null )
[...]
$ ( ./nonblock | cat >/dev/null ) 2>&1 | more
[...]
$ nc -l 8080 >/dev/null &

$ ./nonblock | nc hostname 8080
[...]
```

Resource Locking

Ways we have learned so far to ensure only one process has exclusive access to a resource:

- open file using `O_CREAT|O_EXCL`, then immediately `unlink(2)` it
- create a “lockfile” – if file exists, somebody else is using the resource
- use of a semaphore

What are some problems with each of these?

Advisory Locking

```
#include <fcntl.h>

int flock(int fd,int operation);
```

Returns: 0 if OK, -1 otherwise

- applies or removes an advisory lock on the file associated with the file descriptor fd
- *operation* can be `LOCK_NB` and any one of:
  - `LOCK_SH`
  - `LOCK_EX`
  - `LOCK_UN`
- locks entire file

Advisory Locking

```
$ cc -Wall flock.c
1$ ./a.out
Shared lock established - sleeping for 10 seconds.
[...]
Giving up all locks.

2$ ./a.out
Shared lock established - sleeping for 10 seconds.
Now trying to get an exclusive lock.
Unable to get an exclusive lock.
[...]
Exclusive lock established.

1$ ./a.out
[blocks until the other process terminates]
```

Advisory “Record” Locking

Record locking is done using `fcntl(2)`, using one of `F_GETLK`, `F_SETLK` or `F_SETLKW` and passing a

```
struct flock {
    short l_type;   /* F_RDLCK, F_WRLCK, or F_UNLCK */
    off_t l_start;  /* offset in bytes from l_whence */
    short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t l_len;    /* length, in bytes; 0 means "lock to EOF" */
    pid_t l_pid;    /* returned by F_GETLK */
}
```

Lock types are:

- `F_RDLCK` – Non-exclusive (read) lock; fails if write lock exists.
- `F_WRLCK` – Exclusive (write) lock; fails if any lock exists.
- `F_UNLCK` – Releases our lock on specified range.

Advisory "Record" locking

```
#include <unistd.h>

int lockf(int fd, int value, off_t size);
```

Returns: 0 on success, -1 on error

value can be:

- `F_ULOCK` – unlock locked sections
- `F_LOCK` – lock a section for exclusive use
- `F_TLOCK` – test and lock a section for exclusive use
- `F_TEST` – test a section for locks by other processes

		Request for	
		read lock	write lock
Region currently has	no locks	OK	OK
	one or more read locks	OK	denied
	one write lock	denied	denied

Advisory “Record” locking

Locks are:

- released if a process terminates
- released if a filedescriptor is closed (!)
- not inherited across `fork(2)`
- inherited across `exec(2)`
- released upon `exec(2)` if close-on-exec is set

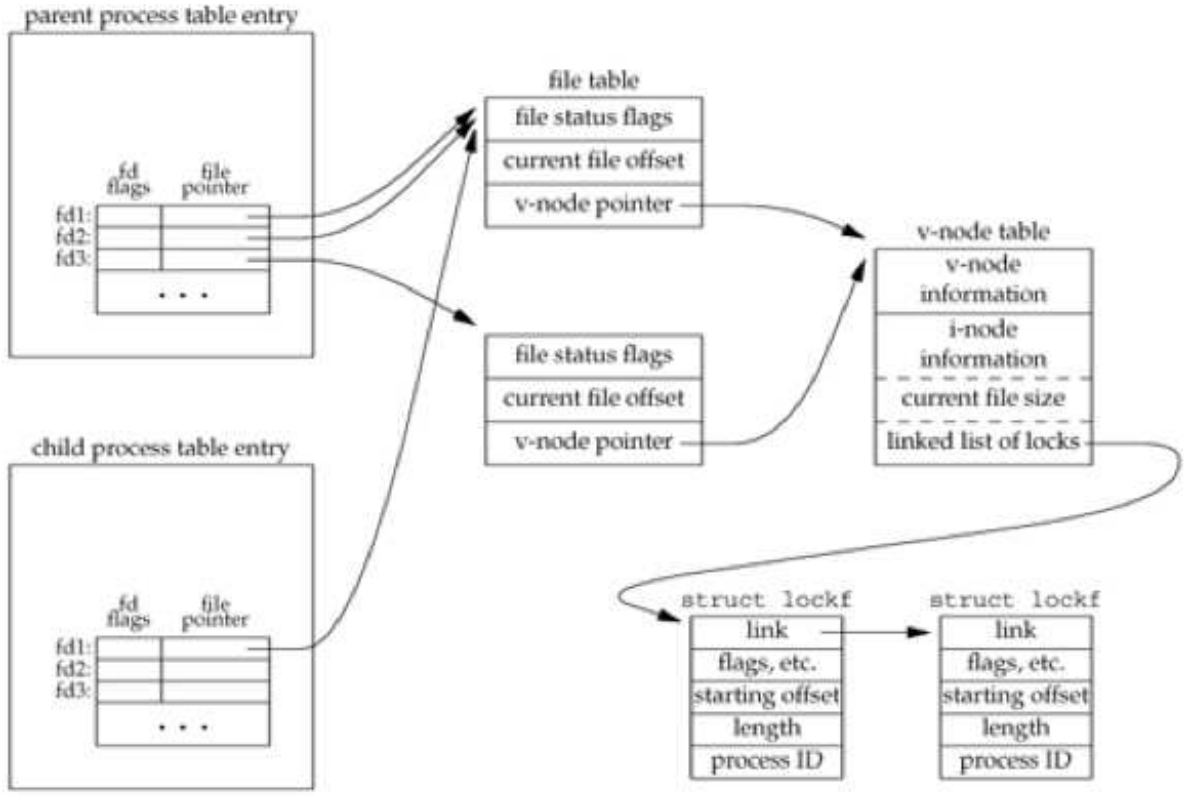
Advisory “Record” locking

Locks are associated with a *file and process pair*, not with a *filedescriptor*!

```
fd1 = open(pathname, ...);
write_lock(fd1...);          /* lock byte 0 */
if ((pid = fork()) > 0) {
        fd2 = dup(fd1);
        fd3 = open(pathname, ...);
} else if (pid == 0) {
        read_lock(fd1...); /* lock byte 1 */
}
```

Advisory “Record” locking

Locks are associated with a *file and process pair*, not with a *filedescriptor*!



Mandatory locking

- not implemented on all UNIX flavors
  - `chmod g+s,g-x file`
- possible to be circumvented:

```
$ mandatory-lock /tmp/file &
$ echo foo > /tmp/file2
$ rm /tmp/file
$ mv /tmp/file2 /tmp/file
```

I/O Multiplexing

Standard I/O loop:

```
while ((n = read(fd1, buf, BUFFSIZE)) > 0) {
        if (write(fd2, buf, n) != n) {
                fprintf(stderr, "write error\n");
                exit(1);
         }
}
```

Suppose you want to read from multiple file descriptors - now what?

I/O Multiplexing

When handling I/O on multiple file descriptors, we have the following options:

- blocking mode: open one fd, block, wait (possibly forever), then test the next fd
- fork and use one process for each, communicate using signals or other IPC
- non-blocking mode: open one fd, immediately get results, open next fd, immediately get results, sleep for some time
- asynchronous I/O: get notified by the kernel when either fd is ready for I/O

I/O Multiplexing

Instead of blocking forever (undesirable), using *non-blocking* mode (busy-polling is inefficient) or using *asynchronous I/O* (somewhat limited), we can:

- build a set of file descriptors we're interested in
- call a function that will return if any of the file descriptors are ready for I/O (or a timeout has elapsed)

I/O Multiplexing

```
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>

int select(int maxfdp1, fd_set *readfds, fd_set *writefds
           fd_set *exceptfds, struct timeval *tvptr);
```

Returns: count of ready descriptors, 0 on timeout, -1 otherwise

Arguments passed:

- which descriptors we're interested in
- what conditions we're interested in
- how long we want to wait
  - `tvptr == NULL` means wait forever
  - `tvptr->tv_sec == tvptr->tv_usec == 0` means don't wait at all
  - wait for specified amount of time

`select(2)` tells us both the total count of descriptors that are ready as well as which ones are ready.

I/O Multiplexing

- filedescriptor sets are manipulated using the `FD_*` functions/macros
- read/write sets indicate readiness for read/write; *except* indicates an exception condition (for example OOB data, certain terminal events)
- EOF means ready for read - `read(2)` will just return 0 (as usual)
- `pselect(2)` provides finer-grained timeout control; allows you to specify a signal mask (original signal mask is restored upon return)
- `poll(2)` provides a conceptually similar interface

See also:

- last week's `strchkread.c`
- `http://daniel.haxx.se/docs/poll-vs-select.html`

Asynchronous I/O

- System V derived async I/O
  - limited to STREAMS
  - enabled via `ioctl(2)`
  - uses `SIGPOLL`
- BSD derived async I/O
  - limited to terminals and networks
  - enabled via `fcntl(2)` (`O_ASYNC`, `F_SETOWN`)
  - uses `SIGIO` and `SIGURG`

Mentioned here for completeness's sake only.
See `aio(7)` for an example of POSIX AIO.

Memory Mapped I/O

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

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

Returns: pointer to mapped region if OK

Protection specified for a region:

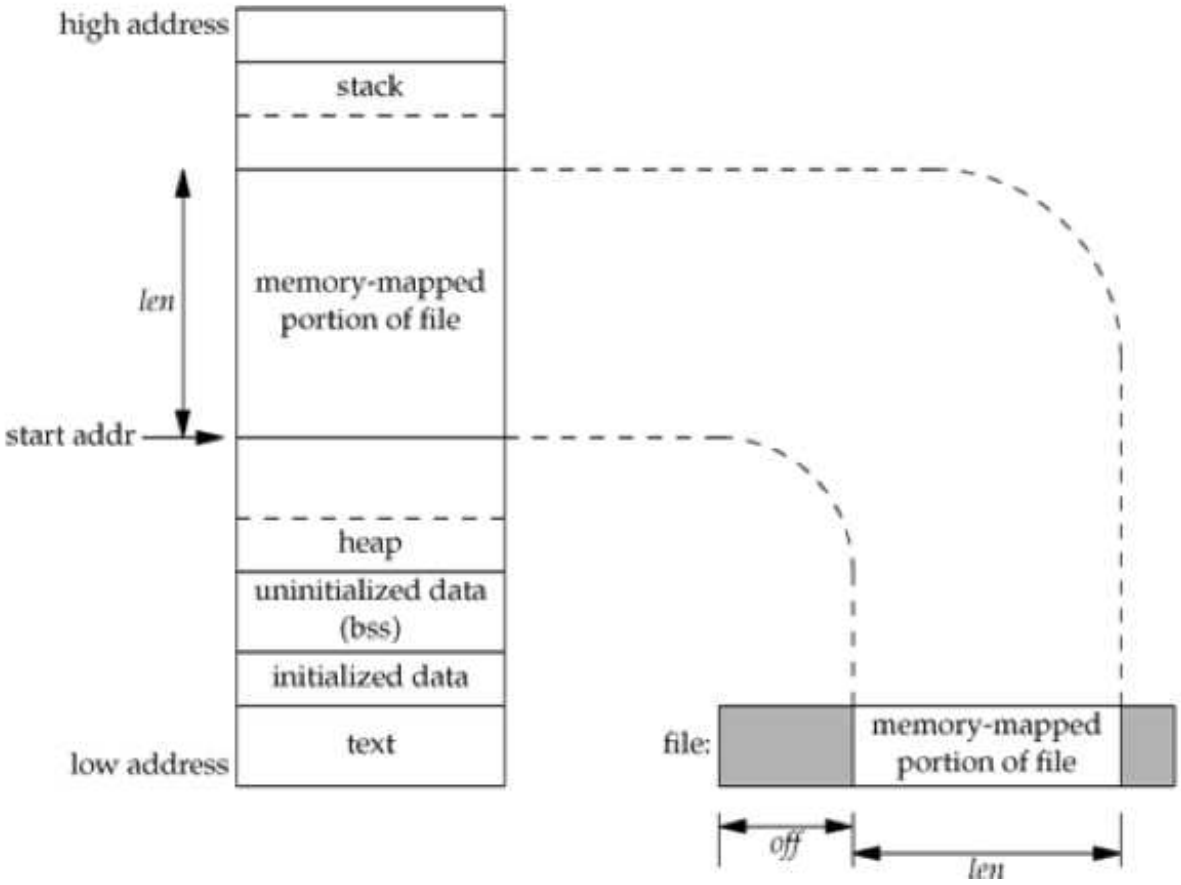
- `PROT_READ` – region can be read
- `PROT_WRITE` – region can be written
- `PROT_EXEC` – region can be executed
- `PROT_NONE` – region can not be accessed

flag needs to be one of

- `MAP_SHARED`
- `MAP_PRIVATE`
- `MAP_COPY`

which may be OR'd with other flags (see `mmap(2)` for details).

Memory Mapped I/O



Memory Mapped I/O

Operation	Linux 2.4.22 (Intel x86)			Solaris 9 (SPARC)		
	User	System	Clock	User	System	Clock
`read/write`	0.04	1.02	39.76	0.18	9.70	41.66
`mmap/memcpy`	0.64	1.31	24.26	1.68	7.94	28.53

Exercise: write a program that benchmarks this performance and run it on the systems you have access to.

Memory Mapped I/O

`http://cvsweb.netbsd.org/bsdweb.cgi/src/bin/cp/utils.c?rev=HEAD`

Final Project

Final project: write a simple web server.

`http://www.cs.stevens.edu/~jschauma/631/f15-final-project.html`

HTTP

Hypertext Transfer Protocol

RFC2616

HTTP

HTTP is a request/response protocol.

The Hypertext Transfer Protocol

HTTP is a request/response protocol:

1. client sends a request to the server
2. server responds

The Hypertext Transfer Protocol

HTTP is a request/response protocol:

1. client sends a request to the server
   - request method
   - URI
   - protocol version
   - request modifiers
   - client information
2. server responds

HTTP: A client request

```
$ telnet www.google.com 80
Trying 2607:f8b0:400c:c02::93...
Connected to www.google.com.
Escape character is '^]'.
GET / HTTP/1.0
```

The Hypertext Transfer Protocol

HTTP is a request/response protocol:

1. client sends a request to the server
   - request method
   - URI
   - protocol version
   - request modifiers
   - client information
2. server responds
   - status line (including success or error code)
   - server information
   - entity metainformation
   - content

HTTP: a server response

```
HTTP/1.0 200 OK
Date: Mon, 22 Oct 2012 03:08:18 GMT
Content-Type: text/html; charset=ISO-8859-1
Server: gws

<!doctype html><html itemscope="itemscope"
itemtype="http://schema.org/WebPage"><head><meta content="Search the
world's information, including webpages, images, videos and more. Google
has many special features to help you find exactly what you're looking
for." name="description"><meta content="noodp" name="robots"><meta
itemprop="image"
content="/images/google_favicon_128.png"><title>Google</title><script>
window.google={kEI:"oriEUNmMGMX50gH6kYGwBw",getEI:function(a){var
b;while(a&&!(a.getAttribute&&(b=a.getAttribute("eid"))))a=a.parentNode;return
b||google.kEI},https:function(){return
window.location.protocol=="https:"},kEXPI:"25657,30316,39523,39977,40362
```

The Hypertext Transfer Protocol

Server status codes:

- 1xx – Informational; Request received, continuing process
- 2xx – Success; The action was successfully received, understood, and accepted
- 3xx – Redirection; Further action must be taken in order to complete the request
- 4xx – Client Error; The request contains bad syntax or cannot be fulfilled
- 5xx – Server Error; The server failed to fulfill an apparently valid request

HTTP: A client request

```
$ telnet www.cs.stevens.edu 80
Trying 155.246.89.84...
Connected to www.cs.stevens-tech.edu.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 302 Found
Date: Sat, 31 Oct 2015 19:02:16 GMT
Server: Apache/2.2.22 (Debian)
Location: http://www.stevens.edu/ses/cs
Vary: Accept-Encoding
Content-Length: 297
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>302 Found</title>
</head><body>
<h1>Found</h1>
<p>The document has moved <a href="http://www.stevens.edu/ses/cs">here</a>.</p>
<hr>
<address>Apache/2.2.22 (Debian) Server at www.cs.stevens.edu Port 80</address>
</body></html>
```

HTTP - more than just text

HTTP is a *Transfer Protocol* – serving *data*, not any specific text format.

- `Accept-Encoding` client header can specify different formats such as *gzip*, *Shared Dictionary Compression over HTTP (SDCH)* etc.
- corresponding server headers: `Content-Type` and `Content-Encoding`

HTTP - more than just static data

HTTP is a *Transfer Protocol* – what is transferred need not be static; resources may generate different data to return based on many variables.

- CGI – resource is *executed*, needs to generate appropriate response headers
- server-side scripting (ASP, PHP, Perl, ...)
- client-side scripting (JavaScript/ECMAScript/JScript,...)
- applications based on HTTP, using:
  - AJAX
  - RESTful services
  - JSON, XML, YAML to represent state and abstract information

Writing a *simple* HTTP server

- parse command-line options, initialize world, ...
- open socket
- run as a dæmon, loop forever
  - accept connection
  - fork child to handle request
- upon `SIGHUP` re-read configuration, restart

Writing a *simple* HTTP server

Processing requests consists of:

- reading request from socket
- parsing request
  - valid syntax?
  - type of request (GET, HEAD, POST)?
  - determine pathname
    - $\sim$ translation
    - translate relative into absolute pathname
- generate server status response
- handle request

Writing a *simple* HTTP server

Processing requests consists of:

- handling regular file request
  - `stat(2)` file
  - `open(2)` file
  - `read(2)` file
  - `write(2)` to socket
  - `close(2)` file
  - terminate connection
  - exit child handler
- handling CGI execution
  - setup environment
  - setup filedescriptors (stdin/stdout)
  - fork-exec executable

Homework

Be able to explain how the shell executes the following statement:

```
( ./a.out | cat >/dev/null ) 2>&1 | more
```

HW#3: webserver framework

`http://www.cs.stevens.edu/~jschauma/631/f15-hw3.html`

Final project: write a simple web server.

`http://www.cs.stevens.edu/~jschauma/631/f15-final-project.html`