Networking API

I.G.Batten@bham.ac.uk

Socket History

- Originally there were a variety of ad-hoc mechanisms which provided access directly to TCP (or whatever) from the OS trap/system-call layer
- Subsequently there have been better (for some value of "better") interfaces, notably in Plan 9 and similar
  - `open ("/net/tcp/192.168.1.1/10")` to get a stream
- But Sam Leffler's "socket" interface from 4.1c bsd, finalised in roughly its current form in 4.2bsd, is almost universal
  - Alternatives are usually different names and calling conventions for same concepts

Buffered I/O

- Userspace programs abstract I/O behind language-specific, hopefully portable, interfaces
  - Buffered Streams of various types in Java
  - FILE * (“stdio”, “standard I/O”) in C
  - IO::Stream in Perl
  - Similar facilities in C++, C#, Python, etc

Abstract I/O

- These facilities try to prevent kernel (or equivalent) routines being called every time you read or write a single character, handling buffering for you
  - stdio block buffers disk, line buffers stdout, character buffers stderr, plus “fflush”
- They also abstract things like permissions
- Common Lisp abstracts pathnames, but that facility is less common (OSX’s case insensitivity and handling of colons can be fun for Unix programs, ditto “what happens if I put a slash in a filename” on *nix)

Underlying Routines

- `fd = open(path, mode);` // return an int (**an fd, or file descriptor**) that indexes into a table of open files
- `bytes = read (fd, buffer, size);`
- `bytes = write (fd, buffer, length);`
- `err = close (fd);` // don't cast this to void, because NFS
- `err = ioctl (fd, operation, argptr);` // do stuff to underlying physical device
- `err = fcntl (fd, operation, argptr);` // do stuff to file descriptor generically

Works for devices

- `ioctl()` is used to do things like setting whether `read()` returns on \r or on each character as it is typed
- `fcntl()` is used to do things like set non-blocking I/O
  - Distinction less clear than it was, because hardware multiplexors for RS232 aren't a thing any more

“Everything is a file”

- New thing in Unix, compared to influences like Multics and TOPS-20, was that there was a file, with a name, you could `open()` for everything (“`/dev/console`”, “`/dev/kmem`”, “`/dev/drum`”, “`/dev/rsd3c`”)
- Files just a stream of bytes: anything else is up to user-space (records, for example). Multics did this for files, but not for devices
  - Not true for “block devices”, but not relevant here
- `/dev` contains “special files” that have a major device number (“what is this?”) and a minor device (“which one of them is this”) which can be `open()`d and then `read()` and `write()` used for I/O, with `ioctl()` doing any required magic

Problem for /net

- Creating a fully populated /net impossible even for IPv4 (2^32 addresses * 2^16 ports is 256TB even at one byte per entry, several Petabytes in real filesystems)
- Tricks by which files spring into existence when `open()`d and disappear when `close`()d are possible, but violate established Unix practice (“`ls`” doesn’t work).
  - Plan 9 “from outer space” was explicitly Unix-influenced, but not afraid to break Unix where necessary.
  - Linux `/proc` offers facility for files you cannot ls, but rarely used

“clone devices”

- There was some precedent with Pseudo-TTYs
- These are the things that are created by ssh to act as a fake teletype (yes!) so that programs that need to mess around with line disciplines, such as emacs and vi, see what looks like a tty rather than a raw network stream
- Large numbers (100s) were needed on time-share machines of the 80s and 90s, so a mechanism arose to allow you to open a generic device and have a specific device created on your behalf (“`/dev/ptmx`” - look it up on a Solaris machine with “`man pts`”, if you can find one)
- Quite slow, doesn’t scale to massive numbers (lots of problems on Solaris as Internet grew became a thing) - created new real devices on the fly

Sockets

- Compromise: we accept that we cannot make connections with `open()`, so don't need an object in the filesystem (or, more strictly, the filesystem namespace) corresponding to a network endpoint
- New system calls that create connections, but in a form that can be used (with care) with `read()` and `write()` (so that most things that work over a tty will work over a network, cf. uucp)
- "**sockets**"
- Sidenote: Sam Leffler went on to be a co-founder of Pixar, showing there is money in OS development

Socket

- `socket()` system call creates a single end of a network connection
  - Again breaks with Unix tradition, as it creates fds which are not ready for `read()` or `write()`
- `bind()` associates the local end, the socket itself, with some addressing information
- `connect()` actively links a socket to another endpoint
- `accept()` is a passive version of `connect()`, following `listen()`
- `send(), sendto(), recv(), recvfrom()` are `write()` and `read()` analogues, although you can use `read()` and `write()` too if you don't need the extra functionality
- `shutdown()` is a richer version of `close()` (allows half-close)

socket()

- `int socket(int domain, int type, int protocol);`
  - domain = { PF_UNIX, PF_INET, PF_INET6, … }
    - Plus other stuff as appropriate for X25, ISDN…
  - type = { SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, … }
    - TCP, UDP, ability to make raw packets, other stuff on a per-OS basis
  - protocol is usually zero, but allows you to force use of a particular non-standard protocol (PF_INET + SOCK_DGRAM otherwise always UDP, etc)

bind

- `int bind(int s, const struct sockaddr *name, socklen_t namelen);`
  - struct sockaddr is a generic for struct sockaddr_in, sockaddr_in6, etc.
  - You put protocol-specific information in here

```c
int
get_listen_socket (const uint16_t port) {

  struct sockaddr_in6 my_address;

  memset (&my_address, '\0', sizeof (my_address));
  my_address.sin6_family = AF_INET6;   /* this is an ipv6 address */
  my_address.sin6_addr = in6addr_any;  /* bind to all interfaces */
  my_address.sin6_port = htons (port); /* network order */


  int s = socket (PF_INET6, SOCK_STREAM, 0);

  if (s < 0) {
     /* error handling */
  }

  if (bind (s, (struct sockaddr *) &my_address,
            sizeof (my_address)) != 0) {
     /* error handling */
   }
```

Byte Ordering

- IBM heritage, 68k, SPARC: “big endian” - **most significant byte** of a multi-byte quantity has **lowest** address
- Intel heritage, VAX heritage, ARM (usually) heritage: “little endian” - **most significant byte** of a multi-byte quantity has **highest** address
  - Means you can do multibyte addition, including the carry, while moving up the address space
  - “On the VAX bytes are handled backwards from most everyone else in the world. This is not expected to be fixed in the near future.”

Concretely

Represent the number 123456, 1E240 hex, in memory

Address	Big Endian	Little Endian
1000	**0**	**40**
1001	**1**	**E2**
1002	**E2**	**1**
1003	**40**	**0**

There are others!

- Less commonly encountered, but as well as 1234 (big endian) and 4321 (little endian) there is/was also, for example, 2143 (Pyramid and some pdp11s).
- For extra entertainment, ARM processors and modern SPARCs can operate with either ordering, although individual OSes tend to force it one way or the other
- Vital to use the OS-supplied macros and never try to roll your own

But on the wire?

- Network ordering is the natural big-endian order, so the most significant bit is transmitted first, the least significant bit is transmitted last
- Great for m68k and SPARC, not so good for everyone else
- Requires care for code-portability (and in any situation where you write ints to disk directly, in passing)
- More generally, of course, serialising more complex data types requires great care

Use the macros!

- `htonl()`, `ntohs()`, etc
- Convert between **h**ost order and **n**etwork order, for **s**horts and **l**ongs
- Anything coming from the network must be processed with `ntohl()` (usually), anything going to the network must be processed with `htonl()`.
- If you get it wrong on x86 your code almost certainly won't work (this is why it is important to test against independent implementations)
- More dangerously, if you get these wrong on SPARC your code will work (all the macros are no-ops), but it will not port to x86. Caused much fun over the years
  - allegedly this is why SPARC and PowerPC are big-endian, as both were m68k replacements and their sponsors were worried about their code-base not being fully compliant
  - See also debate about whether `*((char *) 0) == '\0'`

Voodoo Programming

- Many books, and some of my code, show use of bzero() or memset() to zero the contents of a sockaddr_in before use
- Shouldn't be necessary and correctly assigning values more portable
  - Are `int a = 0;` and `int a; bzero (&a, sizeof (a));` actually the same? Some standards say no (although 99.999% likely to work)
  - You should explicitly set all the elements you are using, at least, and not rely on things you need to be zero being zero'd by the above.

Connecting

Create address of other end, and then…

```
if( connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
       printf("\n Error : Connect Failed \n");
       return 1;
    }
```

If you succeed, the socket is connected

Where do addresses come from?

- Historically, `gethostbyname ()`
- Today all new code should use `getaddrinfo()`
- Consult documentation for the gory details
  - Abstract naming services, so look in a variety of places
  - /etc/hosts, DNS, OS-specific directories, AD…

What about servers?

- Create a socket, as before (bound to INADDR_ANY or in6addr_any, note the difference in case)
- Call `listen(s, backlog)`
  - No-one really knows what backlog means, but 5 is the traditional number - it might mean a backlog of 5, it might be non-linear, it might be ignored: only way to know is to read the kernel source
- Then call `accept()`

```c
while (1) {
    struct sockaddr_in6 their_addr;
    socklen_t size = sizeof(their_addr);

    int newsock = accept(sock, (struct sockaddr *)&their_addr, &size);

    if (newsock == -1) {
        perror("accept");
    }
    else {
        printf("Got a connection from %s on port %d\n",
                /* decode sockaddr_in6 here */
        handle(newsock);
    }
}
```

- NB1: you get back a new socket, with the existing socket left to listen for more
- NB2: if you pass in non-zero arguments, the address of the caller is filled in for you
- NB3: note that **&** on the size: you pass in a pointer to how much space there is, so you can be told how big the result is

Old-School

- `fork()` is a routine which makes a copy of a running process and leaves both of them running
- Only difference is that `fork()` returns a process id into the parent process and 0 into the child process
- Traditional servers fork on incoming connections and have a process per connection

fork() server

```
pid = fork();
 if (pid == 0) {
     /* In child process */
     close(sock); /* child will not accept() again */
     handle(newsock); /* your code goes here */
     return 0;
 }
 else {
     /* Parent process */
     if (pid == -1) {
         perror("fork");
         return -1;
     }
     else {
         close(newsock); /* this is important */
     }
 }
```

New-School

- Create a new thread on each successful call to `accept()`
- No need to close the `listen()`ing socket, as everything in one process
- *pthreads* makes this portable-ish, although high-load performance is something of a lottery

Use thread control blocks

```
typedef struct thread_control_block {
  int client;
  struct sockaddr_in6 their_address;
  socklen_t their_address_size;
} thread_control_block_t;

    thread_control_block_t *tcb_p = malloc (sizeof (*tcb_p));

    if (tcb_p == 0) {
      perror ("malloc");
      exit (1);
    }

    tcb_p->their_address_size = sizeof (tcb_p->their_address);

    /* we call accept as before, except we now put the data into the
       thread control block, rather than onto our own stack,
       because...[1] */
    if ((tcb_p->client = accept (s, (struct sockaddr *) &(tcb_p->their_address),
            &(tcb_p->their_address_size))) < 0) {
```

And create a thread

```
pthread_t thread;

if (pthread_create (&thread, 0, &client_thread,
                    (void *) tcb_p) != 0) {
   /* error handling */
}
```

Key points with threads

- **Can** access:
  - global variables (might need a mutex)
  - variables on their **own** stack (automatics in the thread start routine and anything it calls)
  - malloc’d space which is created by another thread (again, might need a mutex)
- **Must not access**:
  - variables on other threads’ stacks, including the “main” thread.

Pros and Cons

- `fork()` is expensive (but not as bad as it used to be)
  - **NEVER USE `vfork()` EVEN IF OLD BOOKS TELL YOU TO**
  - 4.2BSD, 1984, last updated for OSX 1993: “Users should not depend on the memory sharing semantics of vfork as it will, in that case, be made synonymous to fork.
  - Linux, 2012: “Some consider the semantics of vfork() to be an architectural blemish, and the 4.2BSD man page stated: “This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork() as it will, in that case, be made synonymous to fork(2).””
- child processes are isolated, which is good for security but bad for co-operation
- child processes can drop privilege (ie, can determine correct user and then become that user irrevocably) while all threads run with same privilege: this is important for security.
- Unlikely you could write a production-quality server for a publicly available service with threads alone

I/O

- You can then `read()` and `write()` the new socket
  - or use `recv()` and `send()` for special purposes
  - `readv()` and `writev()` for the hardcore
- Then either `close()` it
  - or use `shutdown()` if you need to just shut down in one direction

Who called me?

- You are told on the `accept()`
- And can call `getpeername()` at any time (just returns what `accept()` would have told you had you asked)
- You can call `getsockname()` to find out your own name if you have forgotten

How do I write the server logic with fork?

- Often uses exec() to replace the child with a new process, and dup() or dup2() to put the socket onto filedescriptors 0 (stdin), 1 (stdout) and 2 (stderr).

```
pid = fork();
 if (pid == 0) {
     /* In child process */
     close(sock);
dup2(newsock, 0);
dup2(newsock, 1);
dup2(newsock, 2);
execl (“/some/binary”, “binary”, “argument”, 0);
    /* NOTREACHED */
exit(1);
 }
 else {
     /* Parent process */
     if (pid == -1) {
         perror("fork");
         return 1;
     }
     else {
         close(newsock);
     }
 }
```

You need fork() earlier

- Network services usually are “daemons”: processes which run unattended without user interaction, started when the machine boots but also available to start/restart from the command-line.
  - Alternative is “inetd”, but this has fallen from favour.
- Therefore, a process needs to start, put itself into the background and detach from its immediate environment so it will run indefinitely.

Daemons

- Huge amounts of voodoo, but in essence you need to:
  - Detach from a “controlling tty” and make sure all output goes to files/syslog/null.
    - When ^C is hit, everything for which the tty is controlling gets `SIGINT`
  - Become a session group leader
  - Make sure there is no-one wait()ing for you

Method 1: call `daemon()`

- Unfortunately, not portable: not in POSIX, but is in most Linuxes and (surprisingly) recent Solaris and OSX.
    - I will be OK with people using this in submissions
- Various greybeard people will raise various objections, not all of them totally invalid, about what it leaves the child able to do.

Method 2: fork, fork

- `fork()` once, parent exits.
- `setsid()`
- [[Drop privilege with `setuid()`]]
- `fork()` again, parent exits
- `close()` 0, 1, 2, re-open as logfiles or /dev/null
-

Being root

- Traditionally, you needed to be root to `bind()` to ports below 1024. The reasons are irrelevant in 2017 but the problem persists.
    - Risk created by non-root `listen()`ing on <1024 far less than risk created by daemons running as root
- Possible to give that power to specified users or non-root processes with very modern Unixes, or indeed turn the restriction off, but sadly rarely used.
- Therefore, best practice is to `bind()` to <1024 while root, then `setuid()` to a less privileged user (httpd, webservd, etc).
- More complex if you then need to log real users in: architecture of `sshd` very tricky to minimise section run as root.

Fun with setuid()

- Root can become anyone
- No-one else should be able to do this without root’s help
- Utterly renouncing your power to be root ever again sadly error-prone and non-portable, because of real/effective uid split introduced in some Unixes.
- Probably strongest to setuid after first, before second `fork()`. Speak to an expert in your precise operating system.

Best Practice

- Run one daemon, isolated from everything else, in a Docker instance, Solaris Zone, *BSD jail, etc.
- If you can’t do that, investigate chroot().
- Daemons that run long-term as root are wrong, wrong, wrong and almost always represent a major security threat.
    - Any buffer overrun = r/w compromise of every file on machine.

More Voodoo

- Older texts may show complex code to avoid every ending up in the position where file descriptors 0, 1 and 2 in a process are all closed or, more generally, where a process has no open file-descriptors, even momentarily.
- Including opening /dev/null onto fd 0 “just in case” and avoiding close(0); close(1); close(2); dup2(s, 0)…
- There was a bug in 4.2bsd in the early 1980s, fixed within about six months. And yet still the superstitions remain.
    - There’s a good mini-project in trying to trace the origins of all the myths and legends around how you daemonise a process

select() or poll()

- What if you want to read and process two sockets at the same time?
  - `select()` or `poll()`, depending on your heritage
    - These days, select() is usually a wrapper around poll(), or sometimes vice versa, with same kernel code used. On older systems one was “better” than the other, but not today.
  - Allow you to specify one or more file descriptors, and returns telling you which are safe to read without their blocking
  - Almost always preferable to non-blocking I/O (tendency to burn 100% CPU) and signal-driven I/O (difficult to get right)
  - Again, today worth considering threads

Buffered I/O

- Using `read()` and `write()` and their networking analogues gets a system call (ie, a context switch) on every use
- This is exactly what stdio seeks to avoid
- You can if you want use `fdopen()` on Unix to get a buffered view of a socket. This is generally a good idea (scatter/gather I/O with `readv()` and `writev()` is only for the keen).
  - Similar facilities in other languages/OSes

Summary: TCP client

- `s = socket (PF_INET6, SOCK_STREAM, …)`
- `bind (s)`
- `connect (s)`
- `while (1) { read(s)` **or** `write(s) }`
- `close (s)` **or** `shutdown (s)`

Summary: TCP Server

- `s = socket(PF_INET6, SOCK_STREAM, …)`
- `bind(s)`
- `listen(s)`
- `while (1) { ns = accept (s); fork();`
  - /* child */ `close(s); while (1) { doio (ns) } close (ns);`
  - /* parent */ `close (ns);`
- Failure to close ns in parent is classic long-term leak-and-fall-over