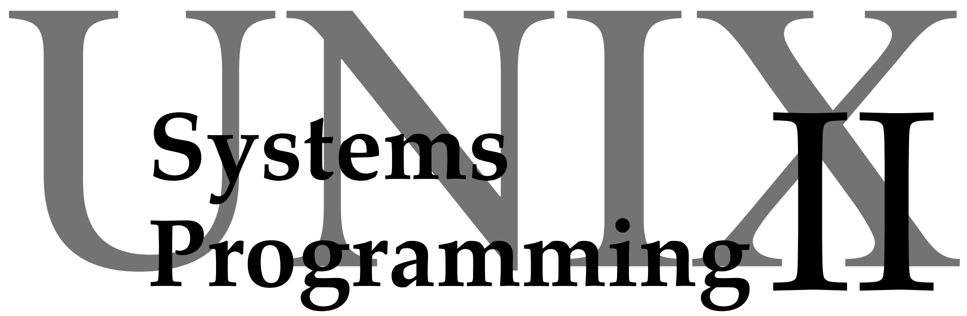
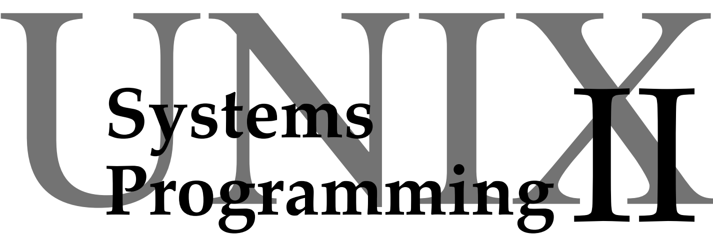
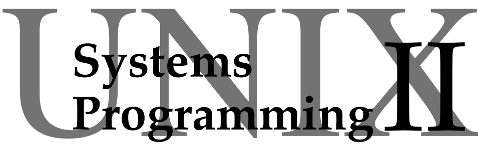






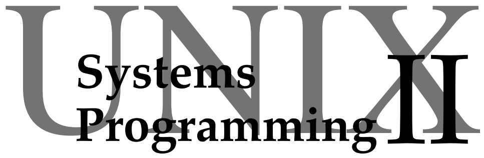
UNIX Systems Programming II

Short Course Notes

Alan Dix © 1996

http://www.hcibook.com/alan/

UNIX Systems Programming II
UNIX Systems Programming II

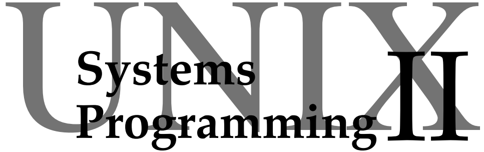


Course Outline

Alan Dix

http://www.hcibook.com/alan/

Session 1	files and devices	inodes, stat, /dev files, ioctl, reading directories, file descriptor sharing and dup2, locking and network caching
Session 2	process handling	UNIX processes, fork, exec, process death: SIGCHLD and wait, kill and I/O issues for fork
Session 3	inter-process communication	pipes: at the shell , in C code and use with exec, pseudo-terminals, sockets and deadlock avoidance
Session 4	non-blocking I/O and select	UNIX events: signals, times and I/O; setting timers, polling, select, interaction with signals and an example Internet server



Reading

- The Unix V Environment,
  Stephen R. Bourne,
  Wiley, 1987, ISBN 0 201 18484 2

  The author of the Borne Shell! A 'classic' which deals with system calls, the shell and other aspects of UNIX.

- Unix For Programmers and Users,
  Graham Glass,
  Prentice-Hall, 1993, ISBN 0 13 061771 7

  Slightly more recent book also covering shell and C programming.

☠ BEWARE – UNIX systems differ in details,
check on-line documentation

- UNIX manual pages:

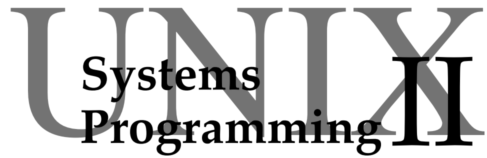
  `man creat` *etc.*

  Most of the system calls and functions are in section 2 and 3 of the manual. The pages are useful once you get used to reading them!

- The include files themselves

  `/usr/include/time.h` *etc.*

  Takes even more getting used to, but the ultimate reference to structures and definitions on <u>your</u> system.



Session 1

files and devices

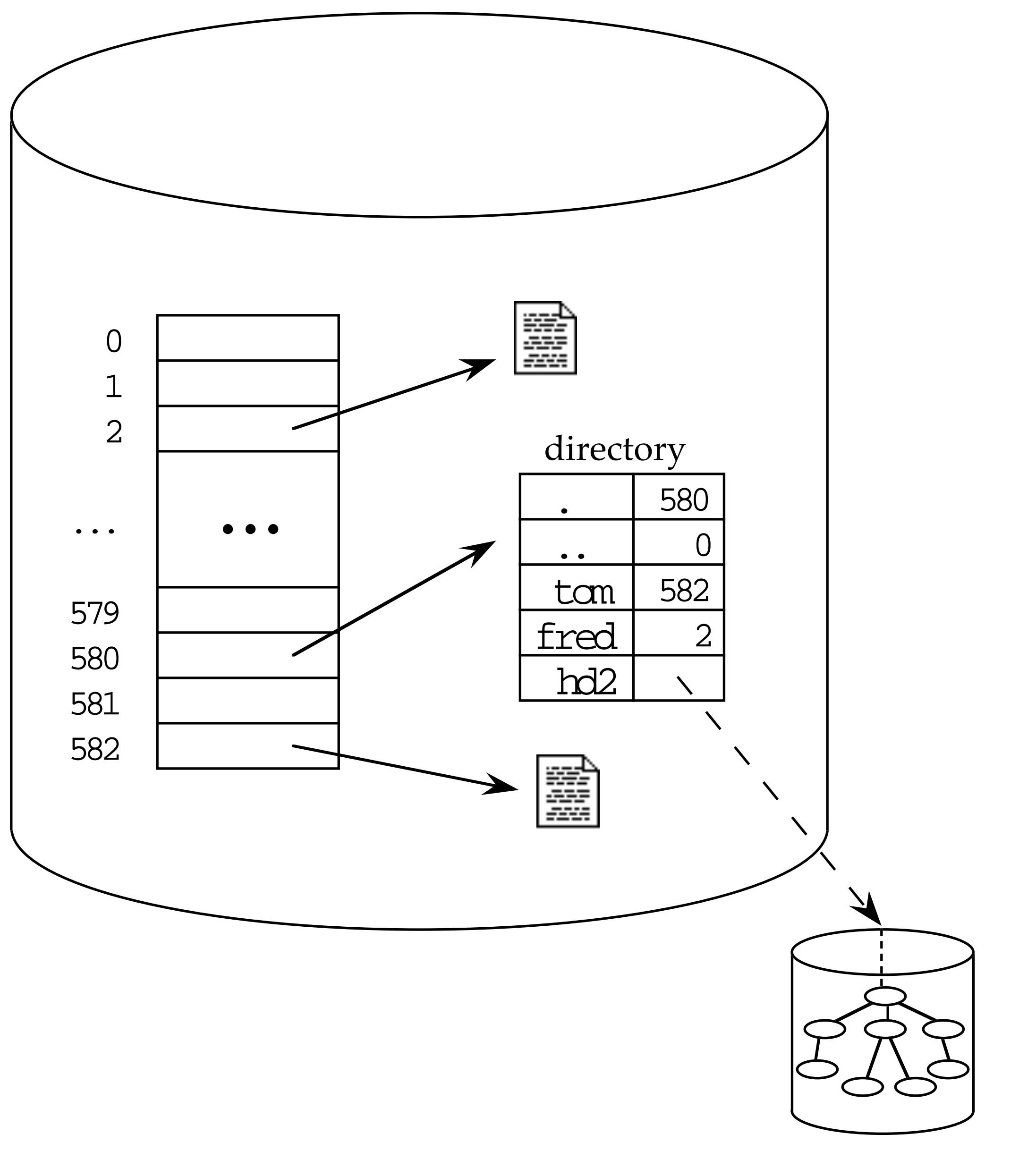
- inodes
- stat
- /dev
- special devices
- ioctl, fnctl
- directories
- file descriptors and dup2
- locking and network problems

☞ a simple `ls`

UNIX filesystem

UNIX filesystem:

- ❍ disk (partition) – inode table (file contents)
- ❍ directories – maps names to files
- ❍ mount points – links between disks
- ❍ special files – e.g. /dev, /n



inodes

- each disk has an inode table
  - ❍ one inode for each file on the disk
- inode contains
  - ❍ permissions: read/write/execute
  - ❍ other flags: directory, symbolic link, setuid
  - ❍ times: creation, modification, access
  - ❍ pointers to disk blocks containing file
- directories
  - ❍ of the form:
    filename → inode number
  - ❍ no other information – all in the inode
- hard link:
  - ❍ two names → same inode number
  - ❍ when safe to delete?

inode reference counts

- inode has count of number of links

- unlink system call does two things:
  ① remove directory entry
  ② subtract 1 from link count

- when count is zero safe to delete file
  . . . or is it?

✘ what about open files?
  ❍ file disappears
  ❍ program does read/write
  ❍ disaster

✓ UNIX also keeps in-memory reference count
  ❍ only deleted when both are zero
  ❍ last close to the file → file finally disappears

stat system call

```
#include <sys/types.h>
#include <sys/stat.h>
struct stat buf

    int res = stat(filename,&buf );
```

- gets information for the file pointed to by `filename`

- all the inode information

- in addition device information
  - device number (which disk)
  - inode number

- inode information includes:
  - mode – `buf.st_mode`
  - size – `buf.st_size`
  - owner – `buf.st_uid`
  - block size – `buf.st_blksize`
  - number of links – `buf.st_nlink`
  - update time – `buf.st_mtime`

- can also get/set mode using `chmod`

stat – 2

- variants of `stat`:

  `lstat(filename,&buf)`
  - same except for symbolic links
  - gives info for link rather than target

  `fstat(fd,&buf)`
  - gives info for open file descriptor
  - useful for standard input/output

- mode flags – `buf.st_mode`
  - permissions
  - file type

- can test mode flags using bit operations

```
if (buf.st_mode & S_IWUSR ) …
```

  – user has write permission?

- also macros for file types, including:

`S_ISDIR(buf.st_mode)`	–	directory
`S_ISREG(buf.st_mode)`	–	regular file
`S_ISLNK(buf.st_mode)`	–	symbolic link

/dev

- UNIX makes everything look like a file!

- many physical devices appear in /dev
  - /dev/hd?? – hard disk
  - /dev/fd?? – floppy disk
  - /dev/s?? – serial port

- in addition, many logical devices
  - /dev/tty – the terminal for this process
  - /dev/win?? – windows
  - /dev/pty??
/dev/tty?? – pseudo-terminals
  - /dev/kmem – kernel memory
  - /dev/null – empty source, infinite sink

- logical devices may be system wide
e.g. pseudo-terminals

- or different for each process
e.g. /dev/tty

special files/devices

- logical and physical devices of two kinds:
  - ❍ character special
  - ❍ block special

- character special files
  - ❍ when it is reasonable to think of the device as a sequence of bytes
    e.g. serial port

- block special files
  - ❍ when it is reasonable to think of the device as a sequence of blocks of bytes
    e.g. formatted disk
  - ❍ get the block size right!

- some physical devices have both
  - ❍ hard disk
    high level – block special
    low level – character special
  - ❍ different semantics

ioctl system call

- all devices appear as files ...

  ... equal but different !

- `ioctl` system call allows device specific control

```
int res = ioctl(fd, cmd, arg );
```

`int fd`	–	file descriptor to control
`int cmd`	–	command/request to perform
`caddr_t arg;`	–	data structure to use/fill

- nature of requests depend on device
  - ❍ see section 4 of manual for device specific requests

filio	–	general 'file' requests
termio	-	terminal requests
sockio	-	socket requests (e.g. TCP/IP)

- type of argument depends on request
  - ❍ read description of request in manual page
- argument may supply data and/or be used for result
- some device specific wrapper functions:

`stty/gtty`	–	terminal drivers
`fcntl`	–	'files' (not just disk)

ioctl – examples

- close on exec

```
#include <sys/filio.h>
ioctl(fd,FIOCLEX,NULL);
```

- don’t block on `read`/`write`

```
#include <sys/filio.h>
int flag = 1;
ioctl(fd,FIONBIO,&flag);
```

- get terminal window size
  e.g. under X when windows may resize

```
#include <sys/termios.h>
struct winsize wsz;
ioctl(fd,TIOCGWINSZ,&wsz);
```

fcntl system call

```
#include <sys/filio.h>
#include <unistd.h>
#include <fcntl.h>
```

```
int res = fcntl(fd, cmd, arg );
```

`int fd`	–	file descriptor to control
`int cmd`	–	command/request to perform
`int arg;`	–	argument purports to be an `int` but is often a pointer!

- performs operations on open file descriptors
- similar to `ioctl`, with some overlap
- some requests cause `ioctl` to be called
- others cannot be performed with `ioctl` (e.g. locks)
- N.B. argument purports to be an `int` . . .
  . . . but is often a pointer!

directories

- stored as ordinary file
    - ❍ sequence of bytes

- inode entry says it is a directory

- can be accessed using ordinary read

✘ only if you know the format!!

⇒ special library functions
    - ❍ read the directory
    - ❍ put it in a data structure

reading directories

```
#include <dirent.h>
    opendir, readdir, closdir,
    seekdir, telldir, rewinddir
```

- the `directory` functions in man(3)
  - like a `stdio` for directories
  - functions to open, read, and close
  - but not write - that is `creat`!
  - also 'seek' and 'tell' style functions

- data structures
  - `DIR` structure takes the place of `FILE*`
  - read returns a pointer to a `struct dirent`
  - only important field is `d_name` – the file name

```
#include <dirent.h>
struct dirent *dp;
DIR *dirp = opendir("testdir");
dp = readdir(dirp);
while ( dp != NULL ) {
    printf("%s\n",dp->d_name);
    dp = readdir(dirp);
    }
closedir(dirp);
```

shared file descriptors

- file descriptors
  - ❍ point to shared structures
- shared:
  - ① within a process
  - ② between processes
- arises from:
  - ① `dup2`/`dup2` system calls
  - ② `fork` system call
- shared means
  - ❍ same file pointer

    writes – sequenced
    reads – first come / first served
  - ❍ last close matters

    files – may be deleted
    network – connection broken

dup2 system call

```
int res = dup2(old_fd, new_fd);
```

- makes `new_fd` point to same file/stream as `old_fd`

- `new_fd` is closed if already open

- most often used with standard I/O descriptors:

  ```
  dup2(fd,0);
  ```

  – standard input reads from `fd`

- can close the old descriptor
  . . . but new descriptor still works

  ```
  dup2(fd,0);
  close(fd):
  n = read(0,buff,buff_len);
  ```

- negative return on failure

locks

- lots of processes accessing the same file
  ⇒ trouble!

- locks prevent multiple access
  ❍ atomic – cannot have half a lock
  ❍ mutual exclusion
  – at most one process has the lock

- traditional UNIX lock file:
  ❍ use `creat` to make an unreadable file
  `creat("lockfile",0);`
  ❍ subsequent calls to `creat` will fail
  (because `creat` on an existing file acts as an open)
  ❍ when done use `creat` to delete lock file

✓ uses ordinary UNIX file handling
– no special locking mechanism needed

✘ fails with network file systems

network files – NFS

- files stored on one or more servers
- remote files accessed by sending network messages
- simply sendUNIX `open`/`read`/`write` requests?
  - ✘ too much network traffic
  - ✘ too much server state
- NFS:
  - ✔ client workstations <u>cache</u> files
  - ✔ server is stateless (⇒ no open)
- some files don't last long
  - ⇒ don't tell server about every `creat`/`write`
    periodic synchronisation
- some files don't last long
  - ⇒ don't tell server about every `creat`/`write`
    periodic synchronisation

✘ `creat` only mutually exclusive on each machine

✘ odd anomalies with `read`/`write`

flock and the lock daemon

```
#include <sys/file.h>
    int res = flock(fd,op);
```

`int fd` – an open file descriptor
`int op` – a locking operation

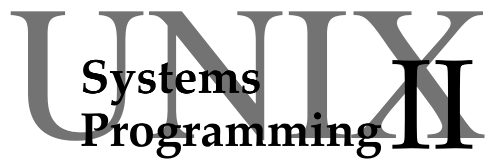
- locking operation one of:
  - `LOCK_SH` – shared lock (mainly for read)
  - `LOCK_EX` – exclusive lock (mainly for write)
  - `LOCK_UN` – release lock (unlock)
- if file is already locked
  - ❍ normally `flock` blocks until it is free
  - ❍ can or operation with `LOCK_NB`
    - ⇒ never blocks – error return instead
- how does it work
  - ❍ background process `lockd` on each machine
  - ❍ they handle the shared state

✔ only have shared state when necessary

✘ still insecure – locks are advisory
  - ❍ process can ignore it and open a locked file

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

- ☞ copy the code fragment on the directory slide
  to write a ‘`mini-ls`’ program
- ☞ it should list the file pointed to by its program first argument (`argv[1]`)
- ☞ compile and run it
- ☞ now modify it to use stat on each file
- ☞ get it to add a slash ‘`/`’ to the end of directories
- ☞ use your imagination to add other status info!
- ☞ compile and run again
- ☞ if you have time, try adding a ‘`-L`’ option
  when the ‘`-L`’ option is given, your program
  should give the details of symbolic links themselves
  as the ‘`-L`’ option does for the real `ls`



Session 2

process handling

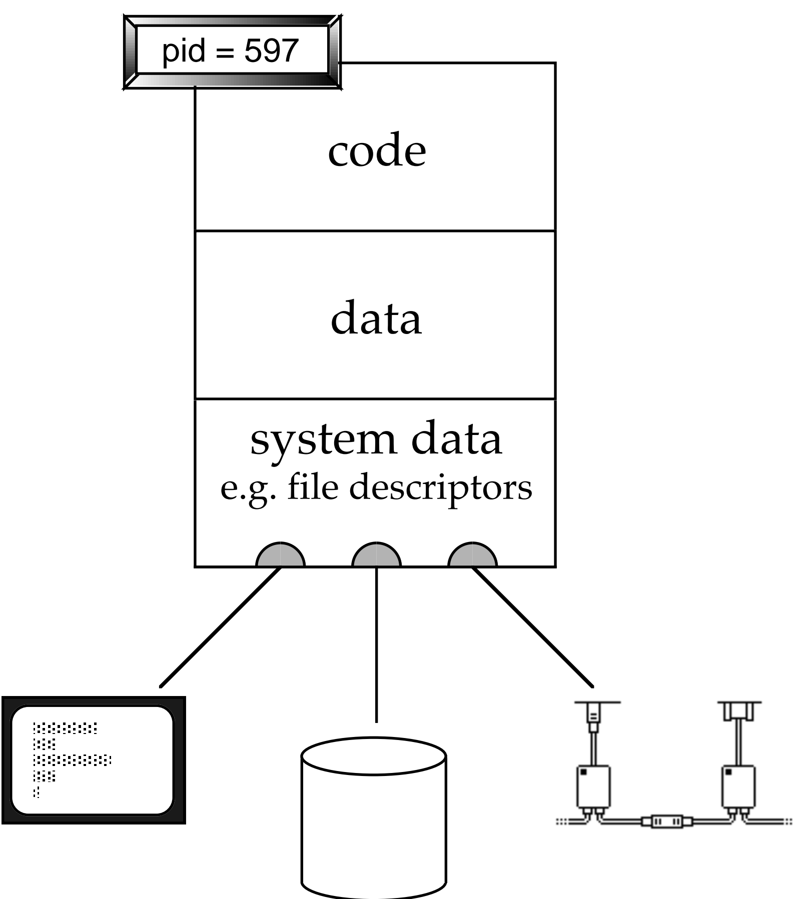
- UNIX processes and forking
- fork system call
- exec system call
- death of process
- kill
- fork and I/O

☞ using it

A UNIX process

UNIX process:

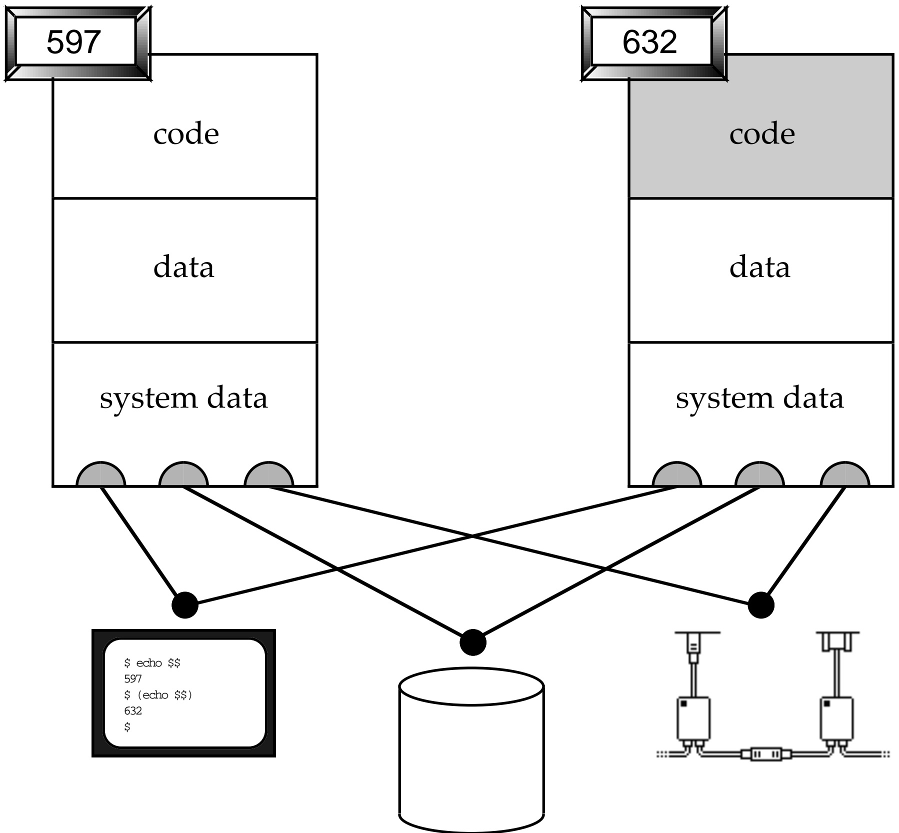
- identified by process id (pid)
- process includes:
  - ❍ program code
  - ❍ application data
  - ❍ system data
    - ❋ including file descriptors



Forking

UNIX 'fork' duplicates process:

- copies complete process state:
  - program data + system data
  - including file descriptors
- code immutable – shared



Forking – 2

- old process called the parent
- new process called the child
- process ids
  - ❍ allocated sequentially
  - ❍ so effectively unique
    (but do wrap after a very long time)
- finding process ids
  - ❍ at the shell prompt:
    use '**`ps`**'
  - ❍ in a C program:
    use '**`int p = getpid();`**'
  - ❍ in a shell script:
    use '**`$$`**'

  N.B. useful for naming temporary files:

```
tmpfile = "/tmp/myfile$$"
```

Fork system call

```
pid_t p = fork();
```

(`pid_t` ≈ `int`)

- if successful
  - ❍ process
  - ❍ successful fork returns:

0	–	to child process
child pid	–	to parent process

  ⇒ parent and child are different!

- negative result on failure

Execution – 1

- parent forks

597

```
    int i = 3, c_pid = -1;
➡   c_pid = fork();
    if ( c_pid == 0 )
        printf("child\n");
    else if ( c_pid > 0 )
        printf("parent\n");
    else
        printf("failed\n");
```

DATA
```
i = 3
c_pid = -1
```

- after fork parent and child identical

597

```
    int i = 3, c_pid = -1;
    c_pid = fork();
➡   if ( c_pid == 0 )
        printf("child\n");
    else if ( c_pid > 0 )
        printf("parent\n");
    else
        printf("failed\n");
```

DATA
```
i = 3
c_pid = 632
```

632

```
    int i = 3, c_pid = -1;
    c_pid = fork();
➡   if ( c_pid == 0 )
        printf("child\n");
    else if ( c_pid > 0 )
        printf("parent\n");
    else
        printf("failed\n");
```

DATA
```
i = 3
c_pid = 0
```

- except for the return value of fork

Execution – 2

- because data are different

597

```
    int i = 3, c_pid = -1;
    c_pid = fork();
➡   if ( c_pid == 0 )
        printf("child\n");
    else if ( c_pid > 0 )
        printf("parent\n");
    else
        printf("failed\n");
```

DATA

```
i = 3
c_pid = 632
```

632

```
    int i = 3, c_pid = -1;
    c_pid = fork();
➡   if ( c_pid == 0 )
        printf("child\n");
    else if ( c_pid > 0 )
        printf("parent\n");
    else
        printf("failed\n");
```

DATA

```
i = 3
c_pid = 0
```

- program execution differs

597

```
    int i = 3, c_pid = -1;
    c_pid = fork();
    if ( c_pid == 0 )
        printf("child\n");
    else if ( c_pid > 0 )
➡       printf("parent\n");
    else
        printf("failed\n");
```

DATA

```
i = 3
c_pid = 632
```

632

```
    int i = 3, c_pid = -1;
    c_pid = fork();
    if ( c_pid == 0 )
➡       printf("child\n");
    else if ( c_pid > 0 )
        printf("parent\n");
    else
        printf("failed\n");
```

DATA

```
i = 3
c_pid = 0
```

- so parent and child behaviour diverge

exec system call

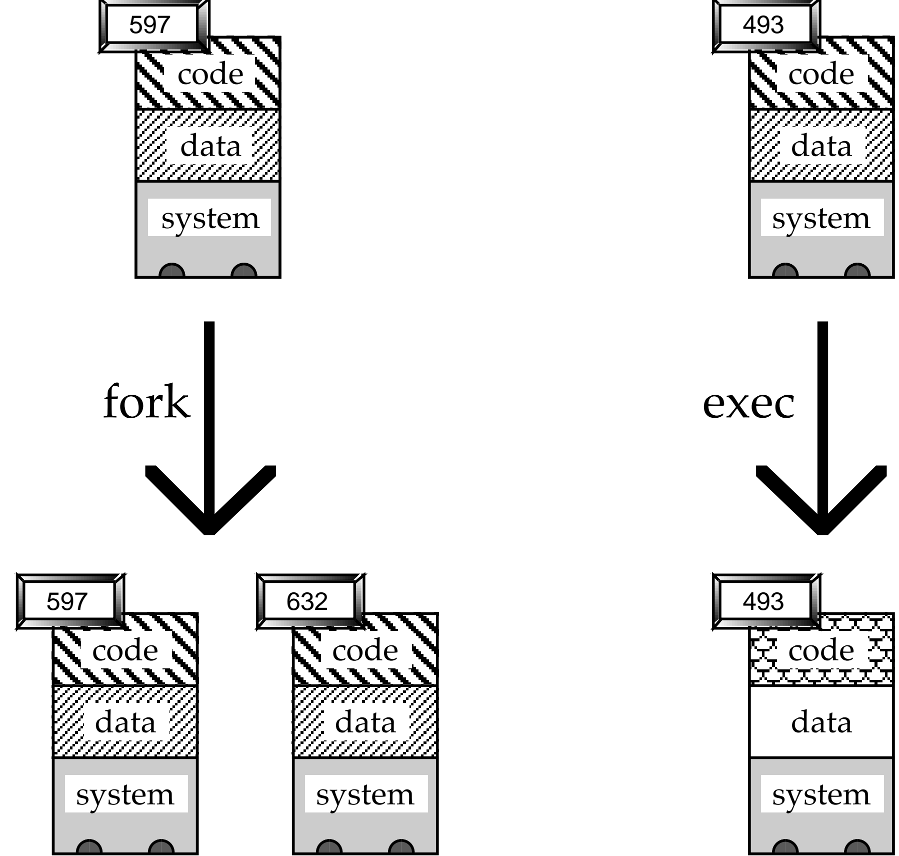
```
execv(char *prog, char **argv);
```

- replaces the current process with `prog`
- never returns except on failure
- `argv` is passed to the 'main' of `prog`
  N.B. needs at least `argv[0]` set to program name
- new process:
  - ❍ code – replaced by `prog`
  - ❍ data – reinitialised
  - ❍ system data – partly retained

  ❋ file descriptors still open
- several variants (`execl`, `execvp`, . . .)
- often used after `fork` to spawn a fresh program

exec vs. fork

- fork <u>duplicates</u> process
- exec <u>replaces</u> process



- fork child shares open file descriptors
- exec-ed process retains open fds

death of a forked process

- when parent dies
  - children become orphans !
  - system init process 'adopts' them

- when child dies
  - parent (or init) informed by signal
    (SIGCHLD)
  - child process partly destroyed
  - rump retained until parent 'reaps'
    – using `wait` or `wait3` system call
  - until then child is 'zombie'
    – `ps` says `<exiting>` or `<defunct>`

N.B. zombie state necessary so parent can discover <u>which</u> child died

wait

- if parent does not reap children
  . . . they stay zombies forever
  . . . system resources may run out

- reap using variants of `wait`

```
union wait status;
int pid = wait(status);
```

  ❍ block until a child has died
  return `pid` of deceased child

```
int sysv(char *prog,char **argv)
{
   union wait status;
   int pid = fork();
   if ( pid < 0 ) return -1;
   if ( pid == 0 ) execvp(prog,argv);
   if ( wait(&status) != pid )  return -1;
   return 0;
}
```

- `wait3` similar, but with more options

SIGCHLD & wait3

- wait blocks
  ✘ no good for concurrent execution
  ✔ SIGCHILD says when child has died

① first catch your signal

```
signal(SIGCHLD,my_reaper);
```

- function 'my_reaper' called when signal arrives

② then reap a child

```
int my_reaper()
{
    union wait status;
    while( wait3(&status,WNOHANG,NULL) >= 0 );
}
```

- use WNOHANG so that wait3 doesn't block
- loop to reap multiple children

kill system call

```
int kill(int pid, int sig);
```

- sends the signal `sig` to process `pid`
- target process – `pid`
  - ❍ must have same effective user id
    (usually launched by same user)
  - ❍ super user programs can kill anyone!
  - ❍ special case: `pid == -1`
    ⇒ broadcast (kill nearly everyone)
    don't worry – super user only
- kill?
  - ❍ only kills the process if it is not caught
    (using `signal` system call)
  - ❍ but some signals can never be caught
- self-destruct
  - ❍ a process can send signals to itself!

fork and I/O

low-level I/O

- open file descriptors shared so:
  - ❍ output is merged
  - ❍ input goes to first read
    - – accept similar
  - ❍ close down may be delayed until all processes close fd
  - ⇒ `close` all unwanted fds or use `ioctl` to set `close-on-exec`

high-level I/O

- C stdio is buffered:
  - ❍ duplicated at fork
  - ❍ may get flushed after fork
    - ⇒ duplicate writes
  - ✓ `stderr` OK – unbuffered
  - ⇒ careful with stdio use `stderr` or `setbuff(fd,NULL)`

☞ ☞ ☞ Hands on ✍ ✍ ✍

☞ look at the program `fork-test.c` in `prog2`
check you understand how it works

☞ copy it into your directory and compile and run it

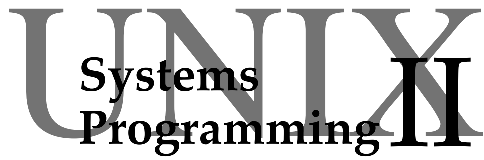
☞ write a test program for the `sysv` function on the `wait` slide

☞ get it to run `/bin/cat` on a test file
the `argv` structure you pass to `sysv` will look something like this:

```
static char *ex_argv[3] = {
        "cat",
        "tom",
        NULL };
```

☞ try redirecting the output by opening a file and then using `dup2`:

```
int fd = open( "dick", O_WRONLY | O_CREAT );
dup2(fd,1);
```



Session 3

inter-process
communication

- shell pipes
- building pipes
- psuedo-terminals
- sockets
- socket I/O and deadlock

☞ using it

IPC

- processes may need to communicate
  - ☞ inter-process communication (IPC)

- different circumstances:
  - ❍ local machine or over network?
  - ❍ forked from single process?
  - ❍ terminal handling required?

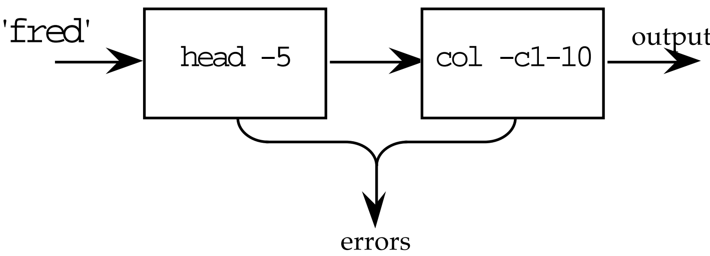
- different mechanisms:
  - ❍ pipes
    - – local, forked
      no terminal handler
  - ❍ pseudo-terminals
    - – local, not necessarily forked
      terminal handler
  - ❍ sockets
    - – remote (e.g. TCP/IP)
      no terminal handler

shell pipes

- UNIX shell pipes join
  the standard output of one command
  to the standard input of another

```
$ head -5 fred | cut -c1-10
```

  - commands run at the same time
  - standard error from both are mixed together (!)



- shell pipes are a special case of a general mechanism

- DOS has pipes too . . .
  . . . but just a shorthand for hidden temporary files!

pipe system call

```
int p[2];
int res = pipe(p);
```

- returns a pair of file descriptors
- connected: `p[1]` → `p[0]`
  - any data written to `p[1]` ...
    ... appears as input to `p[0]`
- buffered within operating system
- initially connects process to itself

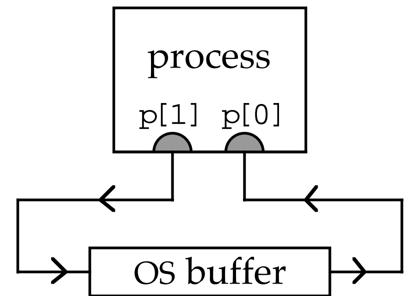
```
int p[2], res, n;
char buff[100]

res = pipe(p);
write(p[1],"hello world",11);
n = read(p[0],buff,100);
write(1,buff,n);
```

- not particularly useful!

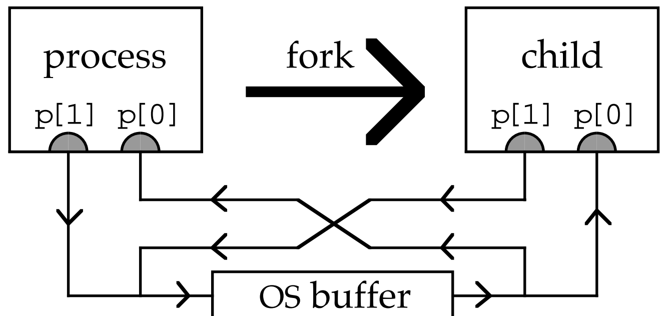
linking processes with pipes

- pipe cannot be used to link existing processes
- but can link process as they fork
- uses the fact that forked file descriptors are shared

① use pipe system call to link process to itself

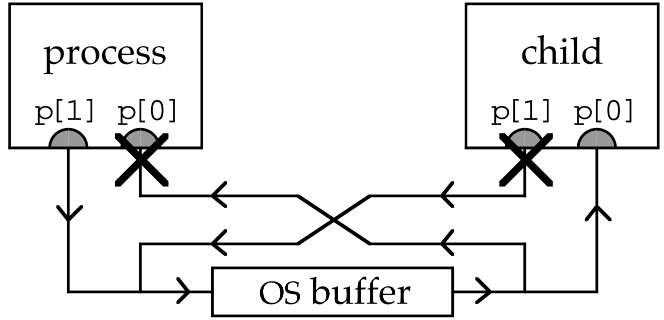


② use fork – file descriptors shared
⇒ both parent and child can:
read from p[0] and write to p[1]

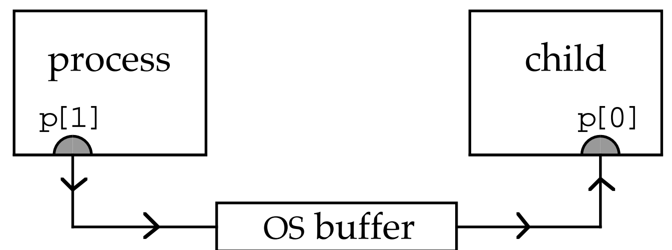


linking with pipes – 2

③ one side closes `p[0]` and the other closes `p[1]`



④ now the two processes can communicate



Note:

- buffer full
  ⇒ `write(p[1] …)` will block
- buffer empty
  ⇒ `read(p[0] …)` will block
- `p[1]` closed and buffer empty
  ⇒ `read(p[0] …)` returns 0
  (end of file)

piping to and from programs

- typical use of pipe:
  - ❍ pipe created between parent and child (stages ① to ④)
  - ❍ child uses dup2 to connect p[0] to stdin
  - ❍ child execs another program “X”
  - ❍ output of parent (through p[1]) → standard input of X

- child code:

```
close(p[1]);      /* output side     */
dup2(p[0],0);
close(p[0]);      /* still open as 0 */

exec("X");
```

- alternatively:
  - ❍ parent retains input side of pipe p[0]
    child connects output side to standard output
    ⇒ parent captures program output
  - ❍ open two pipes for both

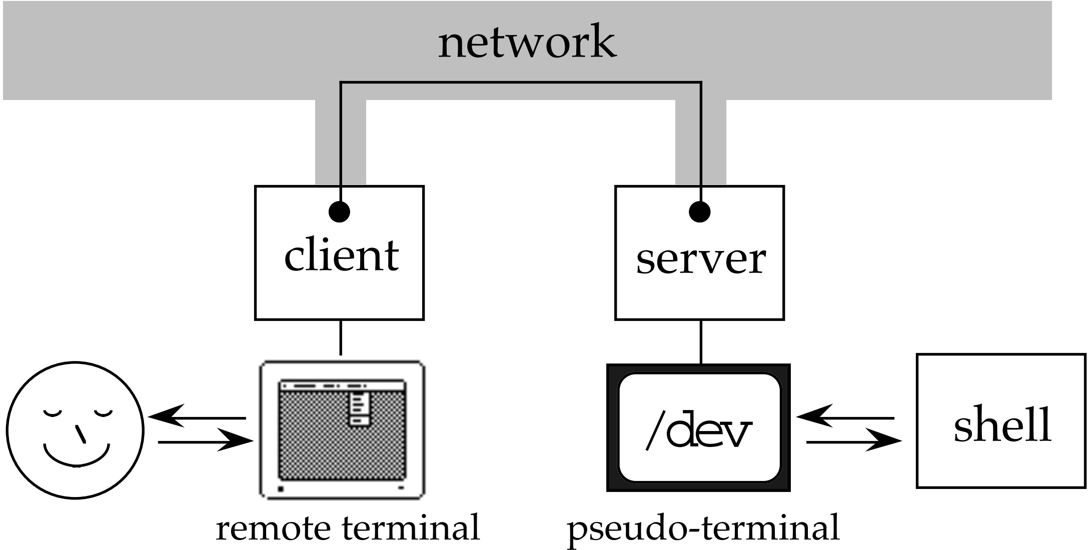
pseudo-terminals

- some programs need a terminal
  e.g. screen editors

  or behave differently with pipes
  e.g. `ls` on Berkeley based UNIX

- pseudo-terminals:
  - ❍ have terminal driver between end-points
  - ❍ allow remote processes to connect

- uses:
  - ❍ window managers
  - ❍ remote logins over network



pseudo-terminals – 2

- special 'virtual' devices in `/dev`

- in pairs
  - ❍ `/dev/ttyp[a-z][0-9]` – slave end
  - ❍ `/dev/ptyp[a-z][0-9]` – controller end

- connection
  - ❍ output to `/dev/ttyp??`
    → input of corresponding `/dev/ptyp??`
  - ❍ output to `/dev/ptyp??`
    → input of corresponding `/dev/ttyp??`
  - ❍ both liable to transformation by tty driver

- asymmetry
  - ❍ `/dev/ttyp??`
    – behaves like the computer end of a tty including `stty` options
  - ❍ `/dev/ptyp??`
    – behaves like the user end of a tty

opening pseudo-terminals

- use normal `open` system call
  - ❍ control end program

```
int fd = open("/dev/ptypb7",2);
```

  - ❍ slave end program

```
int tty_fd = open("/dev/ttypb7",2);
```

- full-duplex connection
  – can read or write to either end

- finding a pseudo-terminal
  - ❍ control end often 'fishes'
    – tries to open each `pty` in turn
  - ❍ how does the other process know which `tty`?
    – often forked after fishing (just like pipes!)
    – other form of IPC

sockets

- generic connection mechanism
  - ❍ networks: e.g. Internet (TCP/IP)
  - ❍ local: e.g. UNIX domain '`socketpair`'
- roots
  - ❍ original Berkeley TCP/IP implementation
- features
  - ❍ central abstraction - the socket - an end-point
    like an electrical connector
  - ❍ uses normal read/write system calls
  - ❍ sockets associated with UNIX file descriptors
    but some not for normal I/O
  - ❍ some extra system calls
    especially fot TCP/IP
- normally bi-directional
  - ❍ read and write to same file descriptor
  - ? close one direction
  - ✔ special socket call `shutdown(sock,dir)`

socketpair system call

- the simplest sockets have no network connection

```
int s[2];
int res = socketpair(s,AF_UNIX,SOCK_STREAM);
```

- returns a pair of sockets (file descriptors)

- like pipes, but both bidirectional: `s[1]` ↔ `s[0]`

```
int s[2], res, n;
char buff[100]

res = socketpair(s,AF_UNIX,SOCK_STREAM);
write(s[1],"one way",7);
n = read(s[0],buff,100);
write(1,buff,n);
write(s[0],"and back again",14);
n = read(s[1],buff,100);
write(1,buff,n);
```

- again use `fork` to establish connected processes

- in fact `pipe` now implemented using `socketpair`

read & write with sockets

- pipes and pseudo-terminals similar

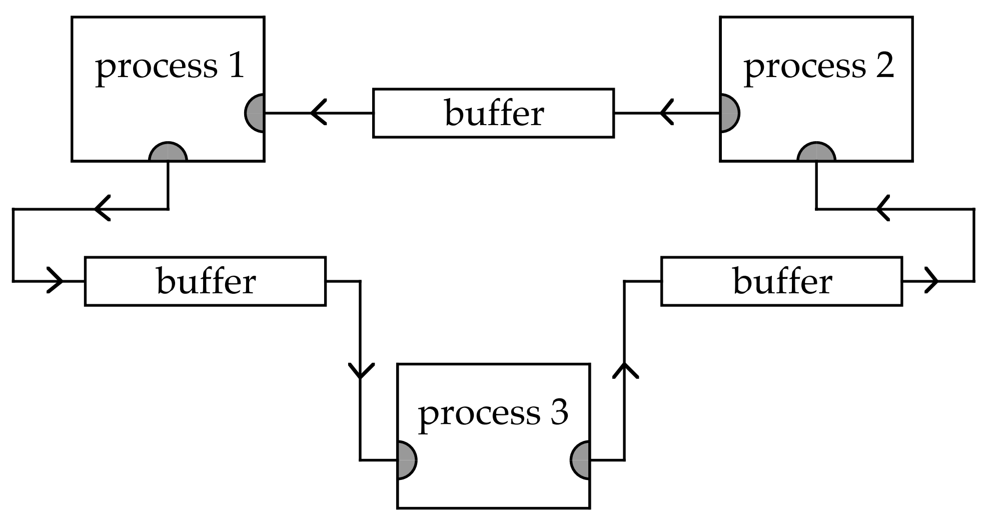
- reading may <u>block</u>
  - ❍ reading from a file either:
    (i) succeeds
    (ii) gets end of file (`ret = 0`)
  - ❍ reading from a socket waits until
    (i) (network) data received (`ret > 0`)
    (ii) connection closed (`ret = 0`)
    (iii) network error (`ret < 0`)

- writing may <u>block</u>
  - ❍ writing to a socket may
    (i) <u>send</u> to the network (`ret > 0`)
    (ii) find connection is closed (`ret = 0`)
    (iii) network error (`ret < 0`)
  - ❍ it may return instantly
  - ❍ but may block if buffers are full

✘ BEWARE – may work during testing
(sufficient buffer space)
then fail in use
(block and deadlock when buffers full)

deadlock

- cycles of pipes/sockets can deadlock



- OK so long as buffers don't fill up
- if everyone writes faster then they read everyone waits for everyone else!
- duplex channels similar

  $\Rightarrow$ don't use blocking I/O

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ write a test program that 'talks to itself' using a pipe, as in the example code on the pipe system call slide

☞ do a similar a test with a socket pair

☞ you are all logged on to the same machine, so should be able to communicate using pseudo-terminals

☞ try it from the shell; one of you types:

```
cat >/dev/ttypn
```

(for some suitably chosen n!)

and the other types:

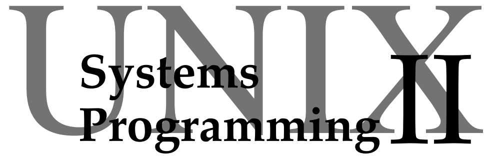
```
cat </dev/ptypn
```

have a nice chat!:

☞ what happens if you try doing the `cats` in the opposite order?

☞ if there is time, try using `fork`, `exec` and `pipe` to perform the equivalent of the shell command:

```
ls /dev | head -30
```



Session 4

non-blocking I/O
and select

- UNIX events
- setting timers
- polling
- select system call
- signals and select

☞ proxy server

UNIX Events

Computational programs:

- busy most of the time
- read / write when they are ready

Interactive programs:

- servers & clients
- idle most of the time
- respond to events

UNIX processes – 4 types of event

① signal (interrupt)

② time (alarm)

③ input ready
   read will not block

④ output can accept (more) data
   write will not block

Responding to events

Events:

① signal (interrupt)
② time (alarm)
③ input (read) ready
④ output (write) ready

Responding

- interrupt handler – ①&②
  - use `signal` system call
  - use `setitimer` to send `SIGALRM`
- turntaking – ②,③&④
  - call `read`/`write` when ready
  - use `sleep` for delays
- polling – ②,③&④
  - use non-blocking `read`/`write`
  - use `time` to do things at specific times
- wait for several events
  - use `select` system call
  - timeout or `SIGALRM`

setting timers

- processes in UNIX have ‘timers’
  - ❍ exactly 3 of them (why 3?!)
  - ❍ like private alarm clocks
  - ❍ precision of milliseconds . . .

    . . . but not accuracy!

- two datatypes

```
struct timeval {                       – single time
     long tv_sec;                      – in seconds
     long tv_usec;                     – and milliseconds
     }
struct itimerval {
     struct timeval it_interval;       – period of timer
     struct timeval it_value;          – next alarm
     }
```

N.B. `it_interval == 0` ⇒ only one alarm

- setting a timer

```
struct itimerval value, oldvalue;
int which;
int res = setitimer(which, &value, &oldvalue);
```

- `which` says which timer to use (an int)
- when timer expires, process sent `SIGALRM`
- read a timer with `getitimer(which, &value);`

polling in UNIX

```
#include <sys/filio.h>
int flag = 1;
            ioctl(fd,FIONBIO,&flag);
```

- call to `ioctl` tells system:
  don't block on `read`/`write`

- polling therefore possible

- structure of polling telnet-like client:

```
int flag = 1;
ioctl(tty_fd,FIONBIO,&flag);
ioctl(net_fd,FIONBIO,&flag);

for(;;) {
       /*  any terminal input?  */
    n = read(tty_fd,buff,buff_len);
    if ( n > 0 ) {  /* yes! do something */  }
       /*  any network input?   */
    n = read(net_fd,buff,buff_len);
    if ( n > 0 ) {  /* yes! do something */  }
}
```

polling pros and cons

✔ program is ‘in control’

✔ similar to normal programs
(i.e. non-interactive programs)

✘ busy polling consumes CPU time

✔ put a `sleep` in the loop

```
int flag = 1;
ioctl(tty_fd,FIONBIO,&flag);
ioctl(net_fd,FIONBIO,&flag);

for(;;) {
     n = read(tty_fd,buff,buff_len);
     if ( n > 0 ) {  /* do something */  }
     n = read(net_fd,buff,buff_len);
     if ( n > 0 ) {  /* do something */  }
     sleep(5);
}
```

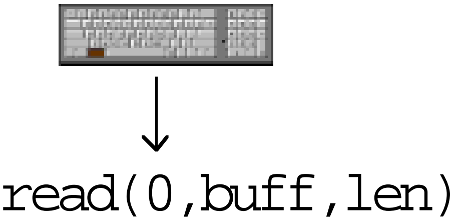
✘ kills interactive performance

✔ OK if fast response not critical
(e.g. no user interaction)

read & write

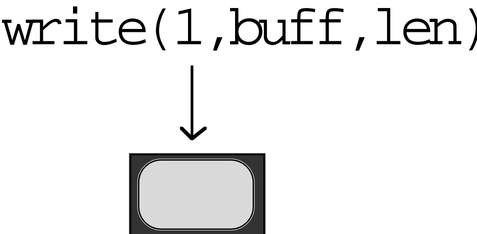
`read`:

- waits on one file descriptor
- returns when input data is ready
- and reads the data into a buffer



`write`:

- waits on one file descriptor
- returns when output is possible
- and writes the data from the buffer

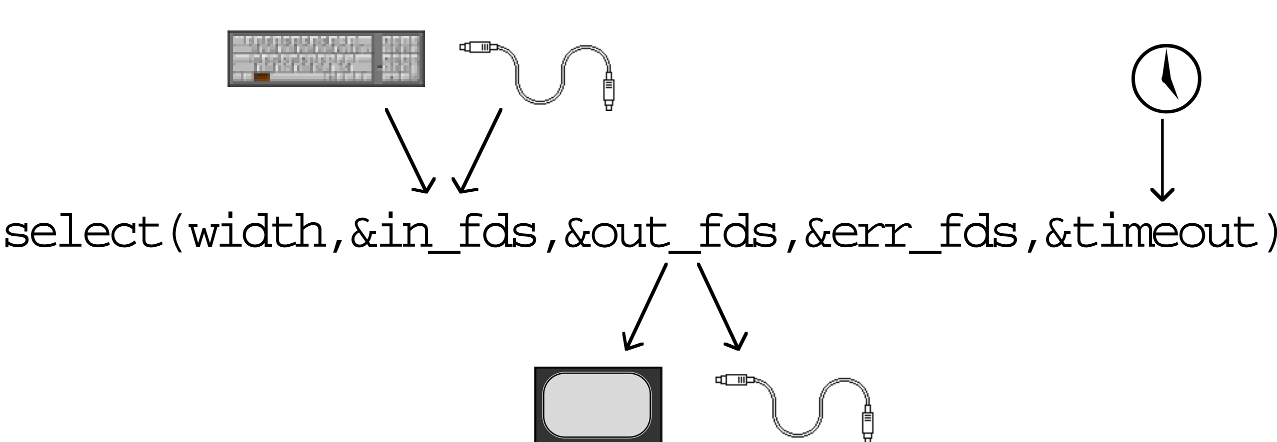


select

`select`:

- waits on many file descriptor
- returns when input or output ready
- but does no actual I/O

\+ also allows timeout



select system call – 2

```
int ret =
      select(size,&in_fds,&out_fds,&err_fds,&timeout);
```

- `in_fds`, `out_fds`:
  – bitmaps of file descriptors
  - `in_fds` – wait for input
    i.e. read will not block
  - `out_fds` – wait for output
    i.e. write will not block
- `size`: – size of `in_fds`, `out_fds`, `err_fds`
- `timeout`: – when to timeout
  in seconds and milliseconds

Returns when:

- input ready on one of `in_fds` (`ret > 0`)
- output ready on one of `out_fds` (`ret > 0`)
- error occurs on one of `err_fds` (`ret > 0`)
- timeout expires (`ret == 0`)
- signal has been caught (`ret < 0`)
- some other error occurs (`ret < 0`)

select and I/O

```
#include <sys/types.h>
      fd_set  in_fds, out_fds, err_fds
```

- modified by call:

call	–	bit set	=	wait for file desc
return	–	bit set	=	file desc ready
		return value from select		
			=	number ready

- long integer in early UNIX systems

```
in_fds = in_fds || ( 1<<fd );
```

⇒ limit of 32 file descriptors
. . . but some systems allow more

- now a special `fd_set` structure
  actually an array of integers!
  - setting:

```
FD_ZERO( &in_fds );
FD_SET( fd, &in_fds );
FD_CLR( fd, &in_fds );
```

  - testing:

```
if ( FD_ISSET(fd,&in_fds) )  ...
```

select and I/O – 2

- input
  - ❍ terminal / socket
    - – `read` will not block
  - ❍ passive socket
    - – `accept` will not block

- output
  - ❍ terminal / socket
    - – `write` 'ready'
  - ❍ `write` relies on system resources
  - ❍ change between `select` and `write`?
    - ⇒ `write` may block
  - ❄ use non-blocking `write`

- can 'get away' without `select` on `write`
  . . . but dangerous!

select and timeouts

```
#include <sys/time.h>
            struct timeval timeout;
```

- `timeout.tv_secs`
  `timeout.tv_ms`
  – maximum time to wait in seconds and ms

- if no I/O ready and no signals in time limit
  then select returns with zero result
  N.B. `in_fds`, `out_fds`, `err_fds` all zero also

- modified by call?
  - ❍ ideally should return time remaining
  - ❍ doesn't now . . .
    . . . but may do one day

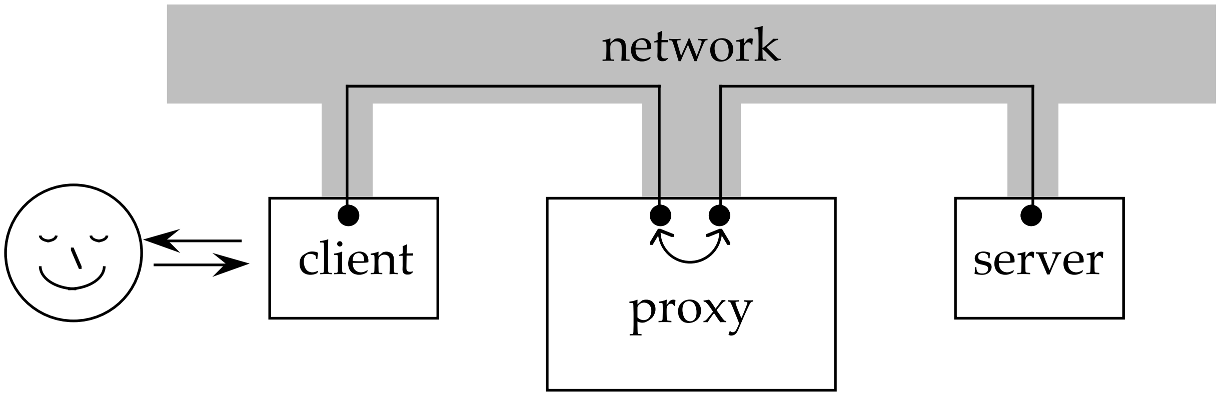
  ⇒ don't rely on `timeout` not being changed
  reset for each call to `select`

select and signals

- signal occurs during system call:
  `read`, `write`, or `select`
- signal not caught . . .
  . . . process aborts!
- signal caught . . .
  ① relevant handler called
  ② systems call returns with 'error'
- how do you know?
  ❍ negative return value
  ❍ `errno` set to `EINTR`
- negative return & `errno` ≠ `EINTR`
  ⇒ really an error!

example – proxy server

- proxy server
  - ❍ monitors Internet traffic to and from server



- structure of code

① wait for client connection

② connect to remote Internet server

③ loop forever
waiting for client or server input:
- ❍ when client data ready
  - read it
  - send to server
  - echo it to terminal
- ❍ when server data ready
  - read it
  - send to client
  - echo it to terminal

proxy code – 1

① Main loop

```
main(...) {
        /*  establish port                         */
    port_sk = tcp_passive_open(port);
        /*  wait for client to connect             */
    client_sk = tcp_accept(port_sk);

        /*  only want one client,                  */
        /*  so close port_sk                            */
    close(port_sk);

        /*  now connect to remote server           */
    serv_sk =  tcp_active_open(rem_host,rem_port);

    ret = do_proxy( client_sk, serv_sk );

    exit(0);
}
```

- basically sets up network connections
  and then calls `do_proxy`

proxy code – 2

② perform proxy loop

```
int do_proxy( int client_sk, int serv_sk )
{
```

- first declare and initialise fd bitmaps

```
    fd_set  read_fds, write_fds, ex_fds;
    FD_ZERO(&read_fds);  FD_ZERO(&write_fds);
    FD_ZERO(&ex_fds);
    FD_SET(client_sk,&read_fds);
    FD_SET(serv_sk ,&read_fds);
```

- then loop forever

```
    for(;;) {
        int      num, len;
```

- copy bitmaps because `select` modifies them

```
        fd_set   read_copy  =  read_fds;
        fd_set   write_copy =  write_fds;
        fd_set   ex_copy    =  ex_fds;
        static struct timeval timeout = {0,0};
```

- then call `select`

```
        num = select(MAX_FD, &read_copy, &write_copy,
                     &ex_copy, &timeout);
```

  ➥ check return – ③, ④ & ⑤ at this point

```
        }
    return 0;
}
```

proxy code – 3

③ check for signals, errors and timeout

- first check for signals:
  in this case, we are not expecting any so return
  in general, we may need to do some processing
  following the interrupt
  it is usually better for the interrupt to set some
  flag and let the main loop do most of the work
  this reduces the risk of stacked interrupts and
  mistakes in concurrent access to data structures

```
if (num < 0 && errno == EINTR ) {
                    /* stopped by signal */
    perror("EINTR");  return 1;
    }
```

- if there has been no signal `num < 0` is an error

```
if (num < 0  ) {        /* not stopped by signal */
    perror("select");  return 1;
    }
```

- if num is zero then a timeout has occurred
  again, in this case no processing
  but in general this is the opportunity for animation
  or other periodic activity

```
if ( num == 0 ) continue;    /*  timeout   */
```

proxy code – 4

④ check for client input
client ready if bit is set in `read_copy`

```
    if ( FD_ISSET(client_sk,&read_copy) ) {
       int len = read( client_sk, buff, buf_len );
```

- on end of file or error exit the loop

```
       if ( len <= 0 ) {   /*   error or close   */
         close(serv_sk);  return len;
         }
```

- if there is some input data, write it to the server and log it

```
       else {
         write(serv_sk,buff,len);
         log_from_client( buff, len );
         }
       }
```

⑤ server input similar

```
    if ( FD_ISSET(serv_sk ,&read_copy) ) {
       int len = read( serv_sk , buff, buf_len );
       if ( len <= 0 ) {   /*   error or close   */
         close(client_sk);
         return len;
         }
       else {
         write(client_sk,buff,len);
         log_from_server( buff, len );
         }
       }
```

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

❇ the proxy server is a bit similar to a telnet client
both open a connection to a remote server
both echo from the user to the server . . .
. . . and from the server to the user
the major difference is that the proxy server
operates on the 'other end' of a network connection

☞ you are going make a simple telnet-like client

☞ copy `proxy.c` and `makefile` from `prog`
copy `proxy.c` and call it `raw-client.c`

❇ `proxy.c` reads and writes the client socket
you want to read from standard input (`0`)
and write to standard output (`1`)

☞ proceed as follows:

① remove the code to open the client connection
(passive open and accept)
② remove the parameter to `do_proxy` which
corresponds to the client socket
③ modify the `FD_SET` calls so that `select` waits
for standard input (`0`) rather than the client
④ change all `read` calls from the client so that
they read from standard input (`0`)
⑤ change all `write` calls to the client so that
they write to standard output (`1`)

☞ now compile and run your raw client, e.g.:

```
raw-client  biggles  7
```

(`biggles` is another UNIX box, 7 is the TCP/IP echo server)