UNIX Systems Programming I

UNIX Systems Programming I

UNIX Systems Programming I

UNIX Systems Programming I

Short Course Notes

Alan Dix

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

UNIX Systems Programming I

UNIX Systems Programming I

UNIX Systems Programming I

Course Outline

Alan Dix

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

Session 1	UNIX basics	the UNIX API, system calls and manuals
Session 2	file I/O and filters	standard input and output, read, write and open
Session 3	makefiles and arguments	make, argc & argv and environment variables
Session 4	file manipulation	creating and deleting files and directories, links and symbolic links
Session 5	terminal I/O	similarities to file I/O, tty drivers, stty & gtty, termcap and curses
Session 6	signals and time	catching signals and problems that arise, delays and finding the time
Session 7	mixing C and scripts	shell scripts, getting information between C and shell, putting them together

UNIX Systems Programming I

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.

UNIX Systems Programming I

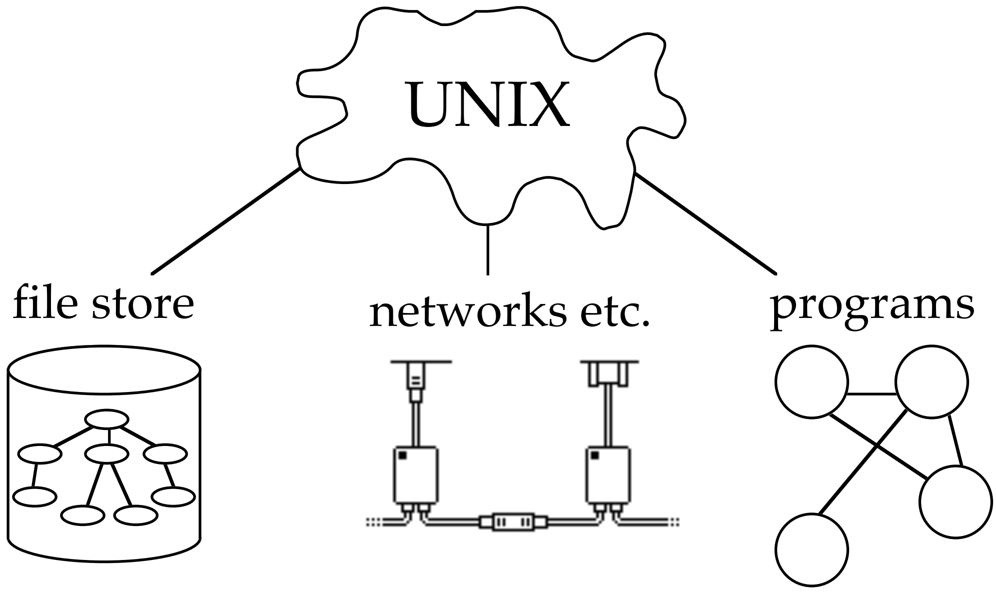
Session 1

UNIX basics

- the nature of UNIX
- the UNIX API
- system calls and library calls
- system call conventions
- how they work
- UNIX manuals and other info

UNIX

UNIX is an operating system



It manages:

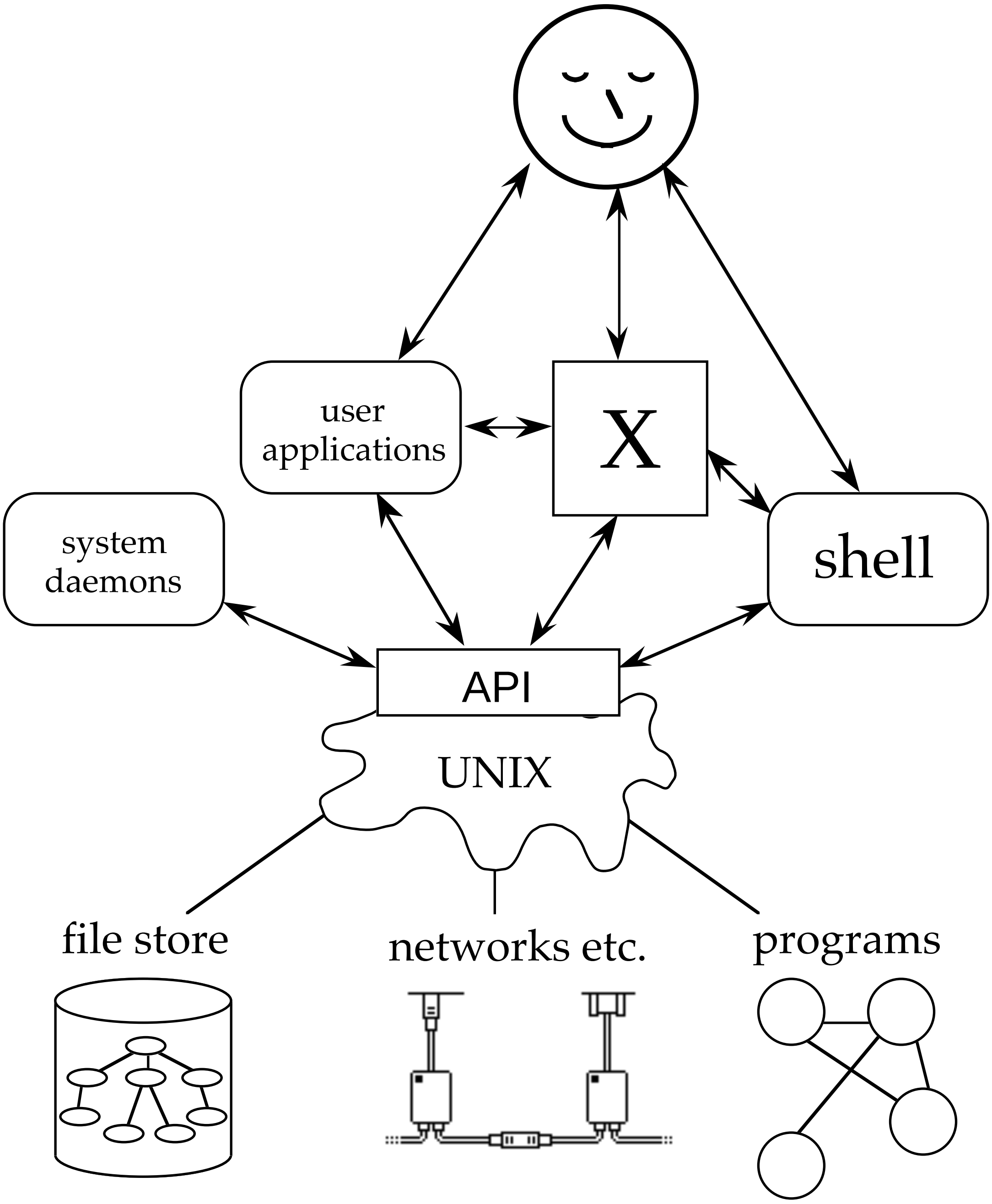
- files and data
- running programs
- networks and other resources

It is defined by

- its behaviour (on files etc.)
- its application programmers interface – API

UNIX API – the system calls

- ultimately everything works through system calls



first system call – exit

```
void  exit(int status)
```

- program ends!
- its exit code is set to `status`
- available to shell:

`$?`	–	Bourne shell
`$status`	–	C shell

- actually not a *real* system call!
  - ❍ does some tidying up
  - ❍ real system call is `_exit`
- example:
  - ❍ does some tidying up
  - ❍ program `test-exit.c`:

```
main()
{
    exit(3);
}
```

  - ❍ run it:

```
$ cc -o test-exit test-exit.c
$ test-exit
$ echo $?
3
$
```

system calls and library calls

- system calls
  - ❍ executed by the operating system
  - ❍ perform simple single operations

- library calls
  - ❍ executed in the user program
  - ❍ may perform several tasks
  - ❍ may call system calls

- distinction blurs
  - ❍ often a thin layer
  - ❍ compatability with older UNIX calls (e.g. pipe)

- kinds of library:
  - ❍ UNIX functions – layers and O/S utilities
  - ❍ stdio & ANSI C libraries
    – platform independent functions
  - ❍ other libraries
    – for specialist purposes (e.g. NAG)

system call conventions

- library functions often return pointers

  `FILE * fp = fopen("harry","r");`

  ⇒ `NULL` return for failure

- system calls usually return an integer

  `int res = sys_call(some_args)`

- return value
  - ❍ `res >= 0` – OK
  - ❍ `res <  0` – failure

- opposite way round!

  ⇒ cannot use as Boolean:

  `if ( sys_call(some_args) ) { ...` ✘ wrong

- see the global variable `errno` for more info

- many system calls take simple arguments

- but some take special structures

how they work

① program gets to the system call in the user's code

```
int res = sys_call(some_parameters)
```

② puts the parameters on the stack
③ performs a system 'trap' – hardware dependent

☆ ☆ now in system mode ☆ ☆

④ operating system code may copy large data structures into system memory
⑤ starts operation
   if the operation cannot be completed immediately
   UNIX may run other processes at this point
⑥ operation complete!
⑦ if necessary copies result data structures back to user program's memory

⑧ ☆ ☆ return to user mode ☆ ☆

⑨ user program puts return code into `res`
⑩ program recommences

- UNIX tries to make it as cheap and fast as possible
- but system calls are still 'expensive'
  (compared to ordinary functions)

finding out

- don't expect to remember everything
  . . . I don't!

- even if you did versions differ

- places to look:
  - ❍ manuals
    - – paper or using `man` command
    - – may need to give `man` the section:
      e.g. `man 2 stat`
  - ❍ `apropos`
    - – especially to find the right `man` page
      e.g. `apropos directory`
  - ❍ look at the source!
    - – read the include file
    - – find the right include file:

```
fgrep dirent /usr/include/*.h
fgrep dirent /usr/include/sys/*.h
```

UNIX manuals

- divided into sections

  1 – shell commands
  e.g. mv, ls, cat

  2 – system calls
  e.g. _exit, read, write

  3 – library calls (including stdio)
  e.g. exit, printf

  4 – device and network specific info
  e.g. mv, ls, cat

  5 – file formats
  e.g. passwd, termcap

  6 – games and demos
  e.g. fortune, worms

  7 – miscellaneous
  e.g. troff macros, ascii character map

  8 – admin functions
  e.g. fsck, network daemons

- UNIX manual reading . . .
  . . . a bit of an art

Session 2

file I/O and filters

- standard input and output
- filters
- using read and write
- opening and closing files
- low-level I/O vs. stdio
- mixing them

☞ using it

input and output

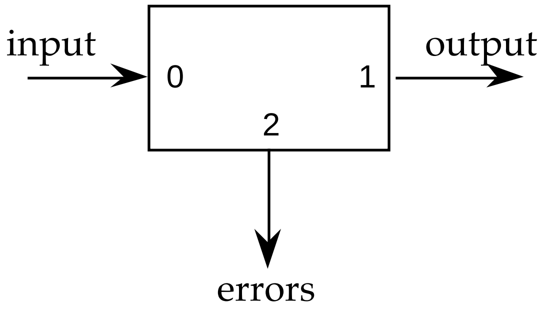
each running program has numbered input/outputs:

0 – standard input
- often used as input if no file is given
- default input from the user terminal

1 – standard output
- simple program's output goes here
- default output to the user terminal

2 – standard error
- error messages from user
- default output to the user terminal

these numbers are called file descriptors
- used by system calls to refer to files

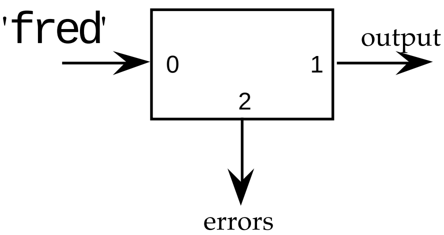


redirecting from the shell

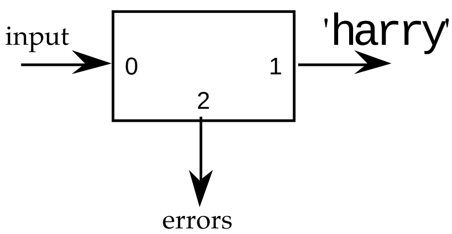
default input/output is user's terminal

redirection to or from files:

- `command <fred`
  - standard input from file '`fred`'



- `command >harry`
  - standard output goes to file '`harry`'



  - file is created if it doesn't exist

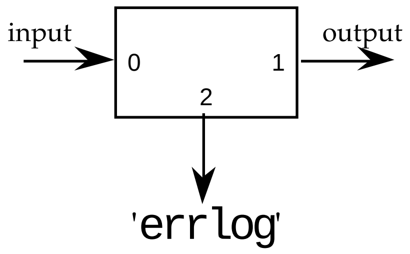
  N.B. C shell prevents overwriting

- `command >>harry`
  - similar, but appends to end of '`harry`'

redirection of standard error

❍ `command 2>errlog`

– standard error goes to file '`errlog`'

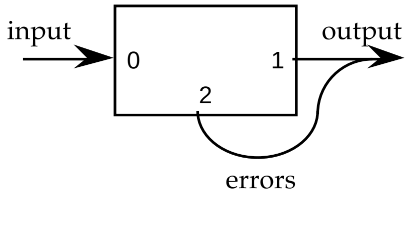


❍ `command 2>>errlog`

– standard error appends to end of '`errlog`'

❍ `command 2>&1`

– standard error goes to <u>current</u> destination of standard output



filters

- some commands only work on named files:

  e.g. copying – `cp` *`from-file to-file`*

- many take standard input as default

  `cat`, `head`, `tail`, `cut`, `paste`, etc.

- these are called <u>filters</u>

  – very useful as part of pipes

- also very easy to program!

✔ don't have to worry about
  - ❍ command line arguments
  - ❍ opening or closing files

- just read-process-write

read & write

```
ret  =  read(fd,buff,len)
```

`int fd`	–	a file descriptor (int), open for reading
`char *buff`	–	buffer in which to put chars
`int len`	–	maximum number of bytes to read
`int ret`	–	returns actual number read

- `ret` is `0` at end of file, negative for error
- `buff` is not NULL terminated
  leave room if you need to add '\0'!

```
ret  =  write(fd,buff,len)
```

`int fd`	–	a file descriptor (int), open for writing
`char *buff`	–	buffer from which to get chars
`int len`	–	number of bytes to write
`int ret`	–	returns actual number written

- `ret` is negative for error, 0 means "end of file"
  `ret` may be less than `len` e.g. if OS buffers full
  * should really check and repeat until all gone *
- `buff` need not be NULL terminated
  if `buff` is a C string, use `strlen` to get its length

N.B. Both may return negative after interrupt (signal)

example – file translation

- a Macintosh → UNIX text file conversion program

```
main() {
    char buf[256];
    for(;;) {
        int i
        int n = read(0,buf,256);              ①
        if ( n <= 0 ) exit(-n);               ②
        for ( i=0; i<n; i++ )
            if ( buff[i] == '\r' )            ③
                buff[i] = '\n';
        write(1,buf,n);                       ④
    }
    exit(0);
}
```

① read from file descriptor `0` – standard input
buffer size is `256` bytes
number of bytes read goes into `n`

② end of file (`0`) or error (`n<0`) both exit
the `-n` means that end of file gives a zero exit code

③ Macintosh uses carriage return `'\r'` to end lines
UNIX uses newline `'\n'`

④ writing to `1` – standard output
remember only write `n` bytes not `256`!

opening and closing files

```
#include <fcntl.h>
```

```
int fd = open(name,flags)
```

`char *name`	–	name of the file to open
`int flags`	–	read/write/append etc.
`int fd`	–	returns a file descriptor

- in simple use `flags` is one of:

`O_RDONLY`	–	read only	(0)
`O_WRONLY`	–	write only	(1)
`O_RDWR`	–	read and write	(2)

  but can 'or' in other options

- negative result on failure
  - ❍ file doesn't exist
  - ❍ do not have permissions

- closing files is simpler!

```
int res = close(fd)
```

`int fd`	–	an open file descriptor		
`int ret`	–	returns:	`0`	OK
			`-1`	failed

low-level I/O vs. stdio

- why use low-level I/O?
  - ❍ less layers ≈ more efficient
  - ❍ more control – e.g. random access
  - ❍ sometimes have to – e.g. networks
- can you mix low-level I/O and stdio?
  - ✔ yes
  - ✘ with care!
- different files/devices
  - ✔ no problem
- same file
  - ✘ stdio is buffered

```
printf("hello ");
write(1,"there ",6);
printf("world\n");
```

  - ➥ output:

```
there hello world
```

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ write a simple cypher filter: `cypher.c`

☞ the filter should copy standard input to standard output adding 1 to each byte:
a→b, b→c, c→d, d→e, etc.

☞ it will be similar to the Mac→UNIX translator except that the loop should add 1 rather than replace carriage returns with line feeds

☞ to make a better cypher, you could add a different number or have a 256 byte array to act as the cypher

☞ compile either with '`cc`' – the old K&R C compiler:

```
cc -o cypher cypher.c
```

☞ or with '`acc`' – the ANSI C compiler:

```
cc -o cypher cypher.c
```

UNIX Systems Programming I

Session 3

makefiles and arguments

- make
- argv and argc
- environment variables

☞ using it

make

'`make`' is a UNIX† command which:

- automates program construction and linking
- tracks dependencies
- keeps things up-to-date after changes

to use it:

- ❍ construct a file with rules in it
  you can call it anything, but '`makefile`' is the default
- ❍ run 'make' itself
  `make target`
  – (uses the default `makefile`)
  `make -f myfile target`
  – (uses the rule file `myfile`)
  either rebuilds the program '`target`' if necessary

- each makefile consists of:
  - ❍ definitions
  - ❍ rules
- rules say how one thing depends on another
  they are either:
  - ❍ specific – e.g. to make `mail-client` do this ...
  - ❍ generic – e.g. to make any '`.o`' from its '`.c`' ...

† `make` is also available in many other programming environments

makefile format

Definitions

- general form:

  variable = *value*

- example:

```
SDIR = tcp
MYLIBS = $(SDIR)/lib
```

  N.B. one variable used in another's definition

- make variables are referred to later using `$`

  e.g. `$(SDIR), $(MYLIBS)`

- expanded like `#defines` or shell variables

  (some versions of make will expand shell variables also)

Rules (just specific rules)

- general form:

  target `:` *dependent1 dependent2* ...
  command-line

  ↑ N.B. this must be a tab

- example:

```
myprog: myprog.o another.o
        cc -o myprog myprog.o another.o $(MYLIBS)
```

  this says:

  to make `myprog` you need `myprog.o` and `another.o`
  if either of them is newer than `myprog` rebuild it using the
  then rebuild it using the command: "`cc -o myprog` …"

argc & argv

```
    int main( int argc, char **argv ) …
or: int main( int argc, char *argv[ ] ) …
```

- one of the ways to get information into a C program

- in UNIX you type:

  ```
  myprog a "b c" d
  ```

  the program gets:

  ```
  argc     =  4         – length of argv
  argv[0]  =  "myprog"  – program name
  argv[1]  =  "a"
  argv[2]  =  "b c"     – single second argument
  argv[3]  =  "d"
  argv[4]  =  NULL      – terminator
  ```

N.B. ❍ DOS is identical (except `argv[0]` is `NULL` early versions)

❍ `argc` is one less than the number of arguments!

- other ways to get information in (UNIX & DOS):
  - ❍ configuration file (known name)
  - ❍ standard input
  - ❍ environment variables using `getenv()`
    or (UNIX only) third arg to main:

    ```
    main(int argc, char **argv, char **envp)
    ```

environment variables

- set of `name=value` mappings
- most created during start-up (.profile, .login etc.)

setting a variable from the shell:

```
myvar=hello
var2=" a value with spaces needs to be quoted"
export myvar
```

- no spaces before or after '=' sign
- variables need to be exported to be seen by other programs run from the shell
- in C shell: "`set name=val`" and no export

listing all variables from the shell:

```
$ set
HOME=/home/staff2/alan
myvar=hello
PATH=/local/bin:/bin:/local/X/bin
USER=alan
  . . .
$
```

environment variables – 2

- accessing from a program – 3 methods

  ① extra argument to main
  `main(int argc,char **argv,char **envp)`

  ② global variable
  `extern char **environ`

  ③ system function
  `char *value = getenv(name);`

- both ① and ② give you a structure similar to `argv`
  - a null terminated array of strings
  - but environment strings are of the form
    `name=value`

- the `getenv` function ③ rather different
  - fetches the value associated with `name`
  - returns `NULL` if `name` not defined

- also a `putenv` function
  - only available to this process and its children
  - not visible to the shell that invoked it

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ write a program to produce a list of arguments as in the 'argc & argv' slide

☞ the core of your program will look something like:

```
for(i=0; i<argc; i++)
    printf("argv[%d] = %s\n",argv[i]);
```

☞ if you use ‘cc’ then the ‘main’ program begins:

```
main(argc,argv)
  int    argc;
  char **argv;
```

the slide shows the ANSI C declaration

☞ do a similar program for environment variables

Session 4

file manipulation

- creating new files
- ‘deleting’ files
- linking to existing files
- symbolic links
- renaming files
- creating / removing directories

☞ using it

creating files

```
int fd = creat(path,mode)
```

`char *path`	–	the name/path of the (new) file
`int mode`	–	permissions for the new file
`int fd`	–	returns a file descriptor

- file is created if it doesn't exist
- if it already exists, acts like an open for writing
- negative return on failure
- mode sets the initial permissions, e.g.

  `mode = S_RWXUSR | S_IRGRP | S_IXGRP | S_IXOTH`

  – read/write/execute for user (`S_RWXUSR`)

  – read/execute for group (`S_IRGRP | S_IXGRP`)

  – execute only for others (`S_IXOTH`)
- when created, file descriptor is open for writing

  ❈ even if permissions do <u>not</u> include write access

- alternative – use `open` with special flags:

  `int fd = open( path, O_WRONLY|O_CREAT|O_TRUNC, mode )`
- `O_CREAT` flag says "create if it doesn't exist"
- note extra `mode` parameter

deleting files

- UNIX `rm` command 'deletes' files
- it uses the `unlink` system call

```
int res  =  unlink(path)
```

`char *path`	–	the name/path of the file to remove			
`int mode`	–	permissions for the new file			
`int res`	–	returns an integer	`0`	–	OK
			`-1`	–	fail

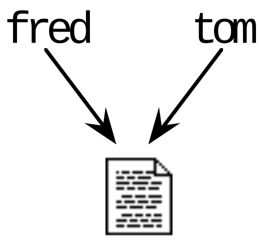
- doesn't necessarily delete file!
- but neither does rm
- UNIX filenames are pointers to the file
- there may be more than one pointer

hard links

- linking to an existing file:
  - ❍ shell – `ln tom fred`
  - ❍ system call – `link("tom","fred")`

- file `tom` must already exist

- `fred` points to the <u>same</u> file as `tom`



- `unlink` simply removes a pointer

- file destroyed only when last link goes

symbolic links

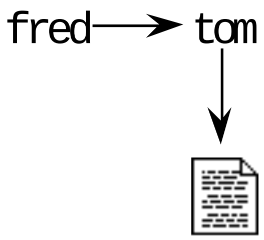
- ‘hard’ links are limited:
  - ❍ cannot link to a different disk
  - ❍ only one link to a directory

    (actually not quite true as there are all the “..” links)

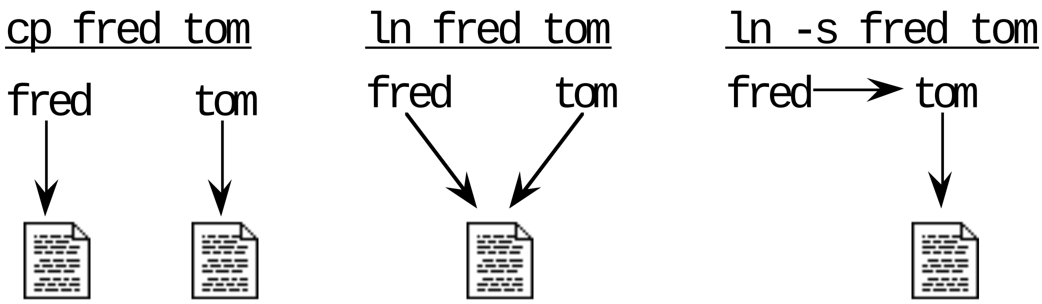
- symbolic links are more flexible
  - ❍ shell – `ln -s tom fred`
  - ❍ system call – sym`link("tom","fred")`

- `tom` need not exist

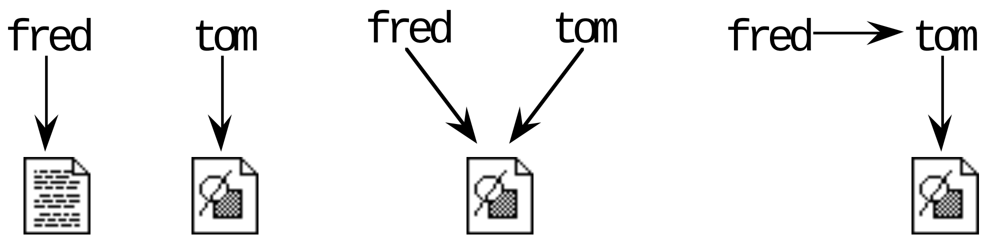
- `fred` points to the <u>name</u> ‘`tom`’ – an alias



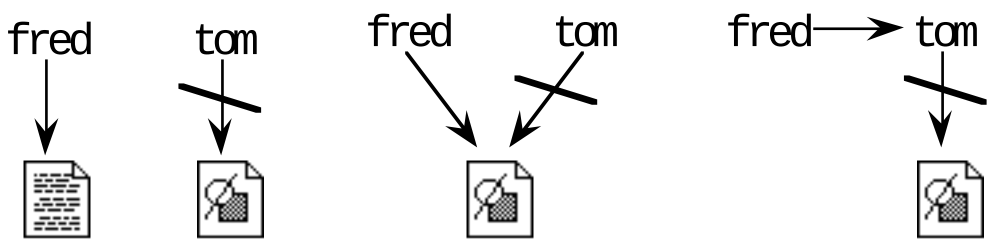
links and updates



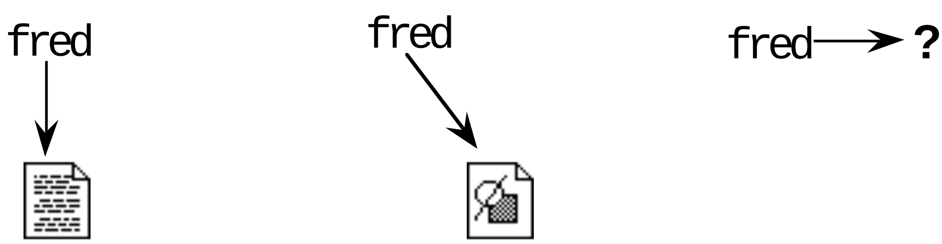
- update file tom



- delete file tom – `unlink("tom")`



- what is in fred?



renaming files

```
int res  =  rename(path1,path2)
```

`char *path`	–	the name/path of the (new) file
`int   fd`	–	returns a file descriptor

- system call used by UNIX `mv` command

- only possible within a file system

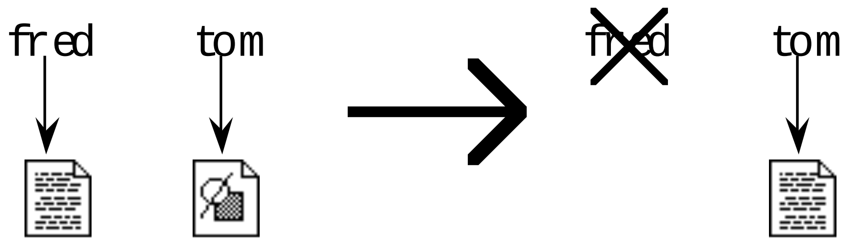
① `path2` is unlinked

② `path2` is linked to the file pointed to by `path1`

③ `path1` is unlinked

- result: `path2` points to the file `path1` used to point to

e.g. `rename("fred","tom")`



directories

- special system calls
  to create and remove directories

```
int res  =  mkdir(path,mode)
```

`char *path`	–	the path of the new directory			
`int mode`	–	permissions for the new directory			
`int res`	–	returns an integer	`0`	–	OK
			`-1`	–	fail

- `mkdir` rather like `creat`

```
int res  =  rmdir(path)
```

`char *path`	–	the path of the directory to remove			
`int res`	–	returns an integer	`0`	–	OK
			`-1`	–	fail

- <u>unlike</u> `unlink` does delete directory!
- but only when empty

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ `rm` has various options

☞ so it is hard to delete files with strange names such as '`-b`' – I know I got one once!

☞ write a program `raw-rm.c` which has one command line argument, a file to delete, and performs an `unlink` system call on it

☞ modify `raw-rm` so that it can take a list of files:

```
raw-rm tom dick harry
```

☞ write a version of `mv` that doesn't use the `rename` system call, but only `link` and `unlink`

☞ N.B. if you get the order wrong you'll loose the file!

Session 5

terminal I/O

- terminals are just files?
- tty drivers
- `stty` and `gtty`
- handling input
- the `termcap` database
- toolkits

☞ using it

terminals are easy?

- terminal is default input/output
- read and write just like any file
    - ❍ use read/write system calls
    - ❍ or stdio
- interactive programs – a doddle

```
printf("what is your name? ");
gets(buff);
printf("hello %s how are you today\n",buff);
```

✔ get line editing for free

✘ paper-roll model of interaction
- ❍ only see user input in whole lines

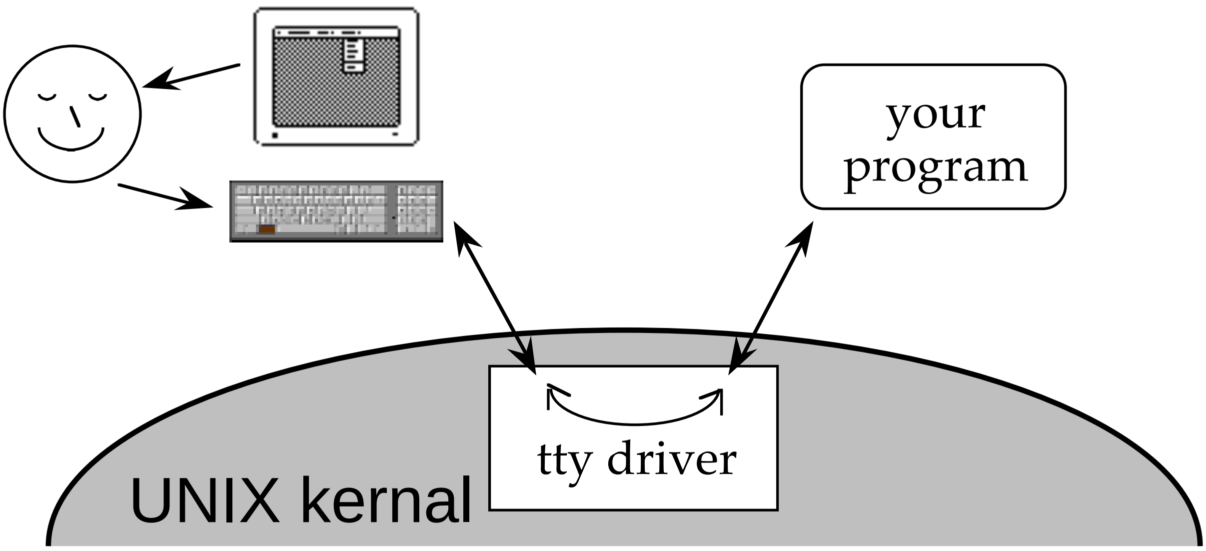
- terminals not quite like other files:
    - ❍ write anywhere on screen
    - ❍ cursor movement
    - ❍ special effects (flashing, colours etc.)
    - ❍ non-standard keys: ctrl, alt, F3 etc.

tty drivers

- never connected directly to terminal

- tty driver sits between
  - ❍ does line editing
  - ❍ handles break keys and flow control
    (ctrl-C, ctrl-\, ctrl-S/ctrl-Q)
  - ❍ translates delete/end-of-line chars

- not always wanted!



stty command

- control the tty driver from the shell

```
$ stty everything
$ stty -echo
$            < type something  – no echo>
$ reset
```

- `stty` differs between UNIXs!

- can control
  - echoing
  - parity, stop bits etc. for modems
  - carriage return / newline mapping
  - delete key mapping (delete/backspace)
  - break key activity
  - line editing

  . . . and more

gtty & stty functions

- you can do the same from a program

```
#include <sgtty.h>
int echo_off(tty_fd)
    int tty_fd;
{
    struct sgttyb buf;
    gtty(tty_fd,&buf);
    buf.sg_flags &= ~ECHO;
    return stty(tty_fd,&buf);
}
```

- `sg_flags` – a bitmap of option flags

- the `sgttyb` structure has other fields
  - ❍ line speeds (`sg_ispeed`, `sg_ospeed`)
  - ❍ erase and kill chars (`sg_erase`, `sg_kill`)
    (word and line delete)

- `gtty` and `stty` depreciated now
  - ❍ more options available through `ioctl`
  - ❍ but they are the 'traditional' UNIX calls
  - ❍ and simpler!

raw, cbreak and cooked

- normal tty processing
  - echoing of all input
  - some output translation
  - line editing
  - break characters

  called ‘cooked’ mode

- visual programs do not want this
  - use `stty` or `ioctl` to control

- raw mode

```
buf.sg_flags |= RAW;
```

  - suppress all input and output processing
  - echoing still occurs – use `ECHO` bit

- cbreak (half-cooked) mode

```
buf.sg_flags |= CBREAK;
```

  - as for raw
  - but break key (ctrl-C) enabled

☠ remember to save and reset the mode!

handling input

- with raw & cbreak
  - ❍ don't have to wait for a line end
  - ❍ get characters as they are typed
  - ❍ including delete key

- key → input mapping?
  - ❍ simple for ascii keys: A key → 'a' etc.
  - ❍ others → single char: backspace → 0x8
  - ❍ some → lots ⬆ key → ESC [A

- prefixes
  - ❍ one key may be a prefix of another!

    e.g. function key F3 → ESC[19~

    escape key → ESC
  - ❍ you read ESC
  - ? how do you know which key

- solutions
  - ① wait for next character
    - ✘ could wait a long time!
  - ② assume that the whole code is read at once
    - ✘ not always true
  - ③ as ① but with timeout
    - ✘ best solution
    - ✘ introduces delays
    - ✘ may still fail (swopped out)

termcap

- different kinds of terminals
  - ❍ different screen sizes (80x25?)
  - ❍ different capabilities (flashing, underline, ...)
  - ❍ different keyboards
  - ❍ different screen control characters

- how do you know what to do?

✘ write terminal specific code

✔ use `termcap`

① environment variable `TERM`
gives terminal type e.g. `vt100`, `vt52` etc.

② `/etc/termcap` database
gives capabilities and codes for each type

```
d0|vt100|vt100-am|vt100am|dec vt100:\
        :do=^J:co#80:li#24:cl=50\E[;H\E[2J:sf=5\ED:\
        :le=^H:bs:am:cm=5\E[%i%d;%dH:nd=2\E[C:up=2\E[A:\
                 < 7 more lines! >
```

❍ each item gives a code/capability

e.g. `do=^J` – send ctrl-J to move cursor down
`co#80` – 80 columns

termcap library

- read `/etc/termcap` directly?
- `termcap` library has functions to help

```
cc -o my-vis  my-vis.c -ltermcap
```

  - `tgetent(val,tname)`
    - get the info for terminal `tname`
  - `tgetnum(id)`
    - return the number for capability `id`
  - `tgetflag(id)`
    - test for capability `id`
  - `tgetstr(id)`
    - return the string value for `id`
  - `tgoto(code,col,line)`
    - generate a cursor addressing string
  - `tputs(str,lines_affected,output_f)`
    - outputs with appropriate padding

- not exactly a high-level interface

curses and toolkits

- various high-level toolkits available

  e.g. curses, C-change

- `curses` is the UNIX old-timer!

```
cc -o my-cur  my-cur.c -lcurses -ltermcap
```

- many functions including:
  - `initscr()` & `endwin()`
    - start and finish use
  - `move(line,col)`
    - cursor positioning
  - `printw(fmt, ...)`
    - formatted output
  - `mvprintw(line,col,fmt, ...)`
    - both together!
  - `mvgetch(l,c)`, `mvgetstr(l,c,buf)`
    - read user input
  - `mvinch()`
    - read screen
  - `clear()`, `refresh()`
    - clear and refresh screen
  - `cbreak()`, `nocbreak()`, `echo()`, `raw()`,
    - setting terminal mode

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ use `stty` at the shell to set echo, cbreak and raw

```
$ cat
        < type a few lines to see what it normally does >
^D
$ stty cbreak
$ cat
        < type a bit >
^D
```

```
$ stty raw
$ echo hello
```

☞ use `cat` to see what codes your terminal produces

```
$ cat
        < type lots of keys and things >
^D
$
```

☞ try entering and running the following:

```
#include <curses.h>

main()
{
        initscr();
        clear();
        mvprintw(10,30,"hello world!");
        move(20,5);
        refresh();
        endwin();
}
```

☞ what happens if you leave out the `refresh()` call

UNIX Systems Programming I

Session 6

signals and time

- what are signals
- the `signal` system call
- problems of concurrency
- finding the time
- going to sleep

☞ using it

signals

- interacting with the world
  - ❍ file/tty input – what
  - ❍ signals – when

- signals happen due to:
  - ❍ errors

    `SIGFPE` – floating point error
    `SIGSEGV` – segment violation (usually bad pointer!)
  - ❍ interruptions

    `SIGKILL` – forces process to die
    `SIGINT` – break key (ctrl-C)
  - ❍ things happen

    `SIGALRM` – timer expires
    `SIGCHLD` – child process has died
  - ❍ I/O event

    `SIGURG` – urgent data from network
    `SIGPIPE` – broken pipe

catching signals

- default action for signals
  - ❍ some abort the process
  - ❍ some ignored

- you can do what you like
  - ❍ so long as you catch the signal
  - ❍ and it's not `SIGKILL` (signal 9)

① write the code that you want run

```
int my_handler()
{
   my_flag = 1\n";
}
```

② use the `signal` system call to tell UNIX about it

```
signal(SIGQUIT,my_handler);
```

③ when the signal occurs UNIX calls `my_handler`

④ when you no longer require a signal to be caught

```
signal(SIGQUIT,SIG_IGN);
signal(SIGFPE,SIG_DFL);
```

care with signals

- signal handlers can run at any time

```
int i = 0;

int my_handler()
{
  i = i + 1;
}

main()
{
  signal(SIGINT,my_handler);
  for(;;)
       if ( i > 0 ) {
                 do_something();
                 i = i - 1;
                 }
}
```

- intention:

  execute `do_something` once per interrupt

- what actually happens:

  ① interrupt processed (`i=1`)
  ② `do_something` executes
  ③ main calculates `i-1` gets result `0`
  ④ before it stores the result . . .
  . . . another interrupt (`i=2`)
  ⑤ `main` stores result (`i=0`)

working with time

- processes need to tell the time
  - ❍ absolute time: 15:17 on Thursday 21st March
  - ❍ elapsed time: that took 3.7 seconds

- and time their actions
  - ❍ delays: wait for 5 seconds
  - ❍ alarms: at 11.15pm do the backup

- delays easy
  - ❍ `sleep(t)` system call
  - ❍ waits for `t` seconds
  - ❍ at least!

```
sleep(5);     /*  waits for at least 5 seconds */
```

- cannot guarantee time
  - ❍ other processes may be running
  - ❍ not a real-time operating system

- alarms covered in UNIX Systems Programming II

finding the time

- UNIX time started on 1st January 1970!

- `time` system call returns seconds 1/1/1970

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

    time_t t = time(NULL);
```

- `ftime` gives you milliseconds and timezone too

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

struct timeb tmb;
    int res = ftime(&tmb);
```

- the process does not run all the time
  `clock` gives cpu time used in µsecs

```
    long cpu_t = clock();
```

N.B. `times` gives you more information about cpu usage

telling the time

- users don't measure time in seconds since 1/1/1970!

- collection of functions to do conversions between four time formats
  - ① seconds since 1/1/1970
  - ② `struct timeb` (from `ftime`)
  - ③ `struct tm` (gives year, day, hour etc.)
  - ④ ascii representation as C string:

    `"Sun Sep 16 01:03:52 1973\n"`

  ① → ③ `localtime, gmtime`

  ③ → ④ `asctime`

  ① → ④ `ctime`

  ③ → ① `timelocal, timegm`

  ❍ also `dysize(yr)` – number of days in year `yr`

- `local` variants give local time
  `gm` variants give Greenwich Mean Time

- see `man 3 ctime` for more details

☞ ☞ ☞ ☞ Hands on ✍ ✍ ✍ ✍

☞ write a program to see how ‘lazy’ `sleep` is!

☞ it should:
① get the time using both `clock` and `time`
② print both
③ do a `sleep(5)`
④ get the times again
⑤ and print them

☞ run it several times and record the results

☞ printing at ② adds a delay,
modify the above plan to make it right
and also get it to print the time elapsed as
measured by `clock` and `time`

☞ run this version and compare results with the first

☞ try the program in the “care with signals” slide

Session 7

mixing C and scripts

- shell scripts
- what are they good for?
- information shell → C
- results C → shell

☞ example

shell

- the shell is a programming language
  - ❍ data:
    environment variables (character strings)
    whole files
  - ❍ control flow:
    similar + special features
  - ❍ procedures:
    shell scripts

- shell and C:
  - ❍ shell:
    - ✔ good at manipulating files and programs
    - ✘ slow for intensive calculation
  - ❍ C:
    - ✔ fast and flexible
    - ✘ longer development cycle

- use them together

shell scripts

- shell scripts are files:
  - ① starting with:
    #!/bin/sh
  - ② containing shell commands
  - ③ made executable by
    chmod a+x
- executed using a copy of the shell

```
$ cat >my-first-script
#!/bin/sh
echo hello world
$ chmod a+x my-first-script
$ my-first-script
hello world
$
```

it’s good to talk

- shell and C need to communicate

- shell → C
  - ❍ standard input:
    large amounts of data
  - ❍ command line arguments:
    file names, options
  - ❍ environment variables:
    long-term preferences & settings

- C → shell
  - ❍ standard output:
    large amounts of data
  - ❍ standard error:
    - ✘ normally only to the user
  - ❍ exit code:
    success/failure or single number

shell → C

- standard input
  – not just files!
  - single line – use `echo` and pipe

```
echo hello | myprog
```

  - lots of lines – use HERE file

```
my-prog <<HERE
this is two lines
> of text
> HERE
```

- command line arguments
  - shell pattern matching is great
  - let it check and pass good args in

- environment variables
  - inwards only!

C → shell

- standard output
  - redirect to file

```
my-prog some-args > fred.out
```

  - pipe it

```
my-prog some-args | more-progs
```

  - or use backquotes!

```
myvar=`my-prog some-args`
```

- exit code
  - remember: 0 = success
  - use `if`, `while` etc.

```
if my-prog some-args
then
    echo OK
else
    echo failed
fi
```

  - or use $?

```
my-prog some-args
echo returned $?
```

☞ ☞ ☞ ☞ example ✍ ✍ ✍ ✍

☞ the following `numc.c` is a filter for numbering lines

```
#include <stdio.h>
char buff[256];
main()
{
     int lineno;
     for ( lineno=1; gets(buff); lineno++ )
          printf("%4d: %s\n",lineno,buff);
}
```

☞ we would like to give it arguments

```
$ numc fred
   1: fred's first line
   2: the second line of fred
```

✘ too much work!

✔ use a shell script

☞ we'll call it `num`

```
#!/bin/sh
case $# in
  0)  numc; exit 0;;   # filter mode
esac
for i    # loops through all arguments
do
    echo; echo "---- $i ----"
    numc <$i
done
```

random access

- normally `read`/`write` are serial
  - move forward byte by byte through the file

- `lseek` allows random access

```
off_t pos  =  lseek(fd,offset,flag)
```

`int fd`	–	an open file descriptor
`off_t offset`	–	number of bytes to seek (negative means back)
`int flag`	–	where to seek from: `L_SET` – beginning of file; `L_INCR` – current position; `L_XTND` – end of file
`off_t pos`	–	the old position in the file (N.B. `off_t` is actually a long int!)

- moves the I/O position forward or back by `offset`

- finding where you are without moving:
  - move zero (`0L`) from current position (`L_INCR`)
  - `tell` function – used to be a system call!