[12] CASE STUDY: UNIX

OUTLINE

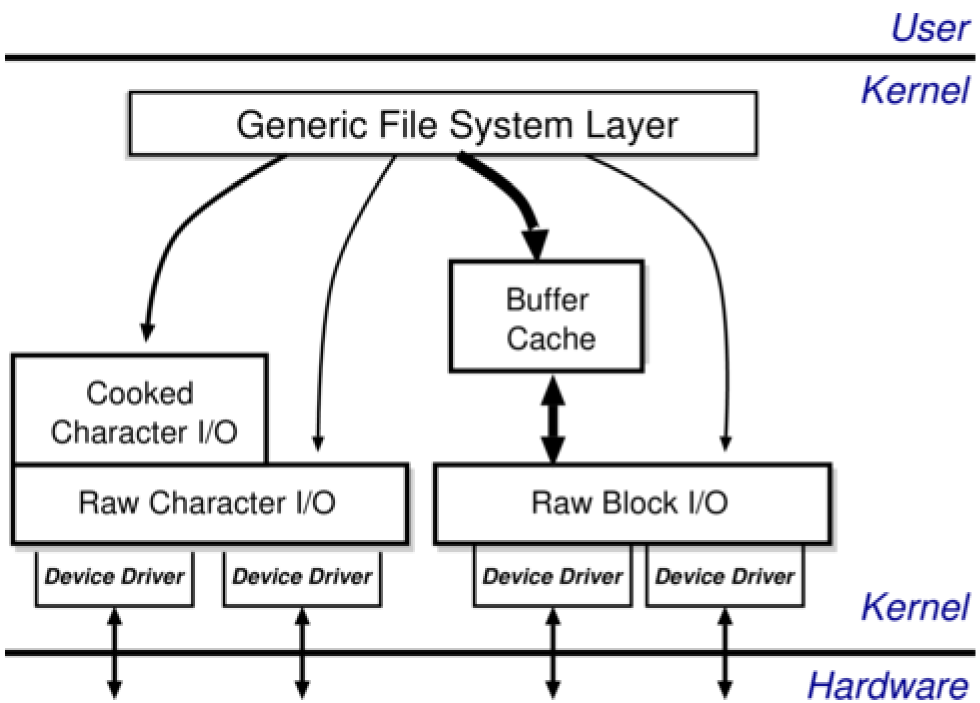
- IO
  - Implementation, The Buffer Cache
- Processes
  - Unix Process Dynamics, Start of Day, Scheduling and States
- The Shell
  - Examples, Standard IO
- Main Unix Features

IO

- **IO**
  - **Implementation, The Buffer Cache**
- Processes
- The Shell
- Summary

IO IMPLEMENTATION

- Everything accessed via the file system
- Two broad categories: block and character; ignoring low-level gore:
  - Character IO low rate but complex — most functionality is in the "cooked" interface
  - Block IO simpler but performance matters — emphasis on the buffer cache



THE BUFFER CACHE

Basic idea: keep copy of some parts of disk in memory for speed

On read do:

- Locate relevant blocks (from inode)
- Check if in buffer cache
- If not, read from disk into memory
- Return data from buffer cache

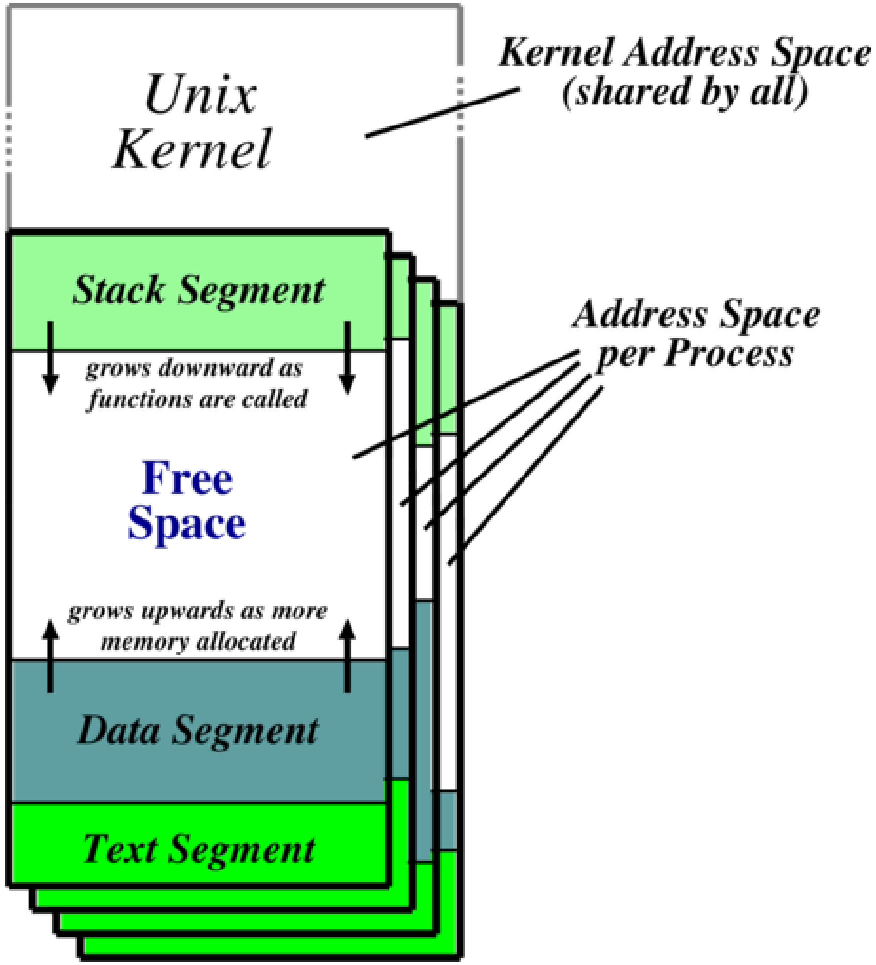
On write do same first three, and then update version in cache, not on disk

- "Typically" prevents 85% of implied disk transfers
- But when does data actually hit disk?
- Call `sync` every 30 seconds to flush dirty buffers to disk
- Can cache metadata too — what problems can that cause?

PROCESSES

- IO
- **Processes**
  - **Unix Process Dynamics, Start of Day, Scheduling and States**
- The Shell
- Main Unix Features

UNIX PROCESSES



Recall: a process is a program in execution

Processes have three segments: text, data and stack. Unix processes are heavyweight

Text: holds the machine instructions for the program

Data: contains variables and their values

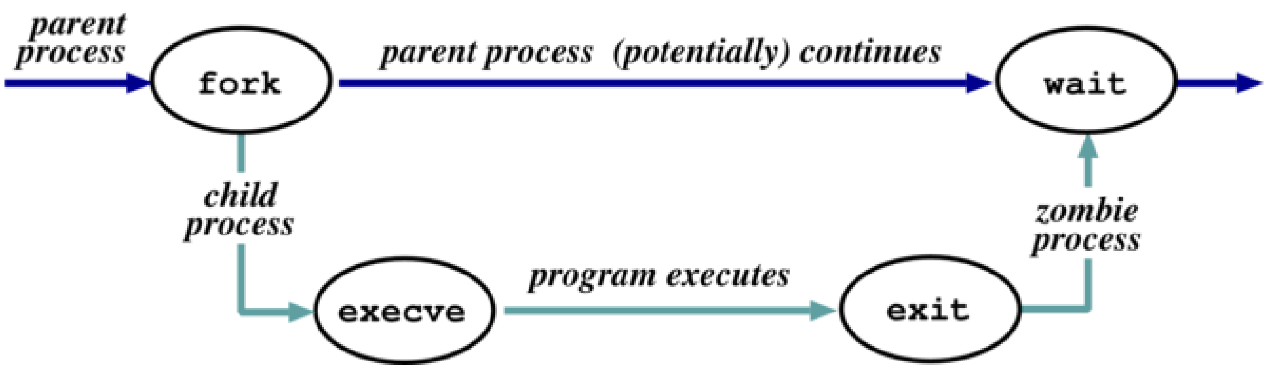
Stack: used for activation records (i.e. storing local variables, parameters, etc.)

UNIX PROCESS DYNAMICS

Process is represented by an opaque process id (`pid`), organised hierarchically with parents creating children. Four basic operations:

- `pid = ` **`fork`**`()`
- `reply = ` **`execve`**`(pathname, argv, envp)`
- **`exit`**`(status)`
- `pid = ` **`wait`**`(status)`

`fork()` nearly always followed by `exec()` leading to `vfork()` and/or copy-on-write (COW). Also makes a copy of entire address space which is not terribly efficient



START OF DAY

Kernel (`/vmunix`) loaded from disk (how — where's the filesystem?) and execution starts. Mounts root filesystem. Process 1 (`/etc/init`) starts hand-crafted

`init` reads file `/etc/inittab` and for each entry:

- Opens terminal special file (e.g. `/dev/tty0`)
- Duplicates the resulting fd twice.
- Forks an `/etc/tty` process.

Each tty process next: initialises the terminal; outputs the string `login:` & waits for input; `execve()`'s `/bin/login`

login then: outputs "password:" & waits for input; encrypts password and checks it against `/etc/passwd`; if ok, sets uid & gid, and `execve()` shell

Patriarch init resurrects `/etc/tty` on exit

UNIX PROCESS SCHEDULING (I)

- Priorities 0—127; user processes ≥ PUSER = 50. Round robin within priorities, quantum 100ms.
- Priorities are based on usage and `nice`, i.e.

$$P_j(i) = \mathrm{Base}_j + \frac{\mathrm{CPU}_j(i-1)}{4} + 2 \times \mathrm{nice}_j$$

  gives the priority of process *j* at the beginning of interval *i* where:

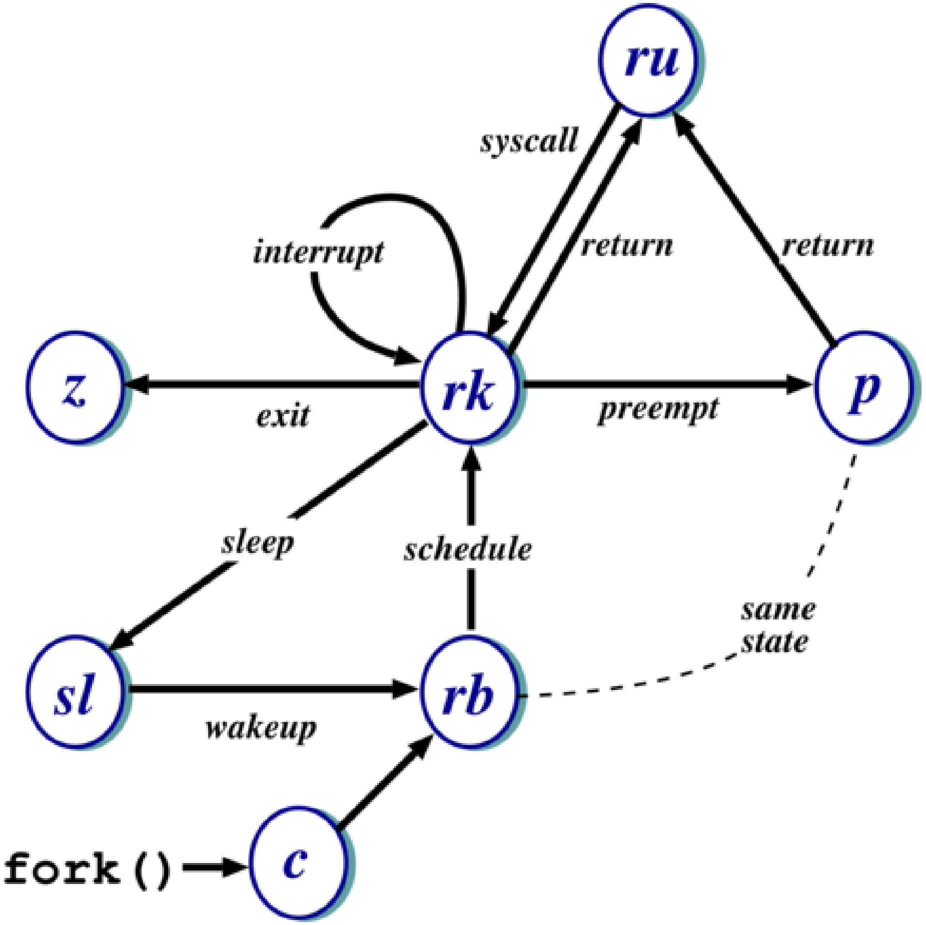
$$\mathrm{CPU}_j(i) = \frac{2 \times \mathrm{load}_j}{(2 \times \mathrm{load}_j) + 1}\mathrm{CPU}_j(i-1) + \mathrm{nice}_j$$

  and nice_j is a (partially) user controllable adjustment parameter in the range $[-20, 20]$
- load_j is the sampled average length of the run queue in which process *j* resides, over the last minute of operation

UNIX PROCESS SCHEDULING (II)

- Thus if e.g. load is 1 this means that roughly 90% of 1s CPU usage is "forgotten" within 5s
- Base priority divides processes into bands; CPU and nice components prevent processes moving out of their bands. The bands are:
  - Swapper; Block IO device control; File manipulation; Character IO device control; User processes
  - Within the user process band the execution history tends to penalize CPU bound processes at the expense of IO bound processes

UNIX PROCESS STATES



ru	=	running (user-mode)	rk	=	running (kernel-mode)
z	=	zombie	p	=	pre-empted
sl	=	sleeping	rb	=	runnable
c	=	created			

NB. This is simplified — see *Concurrent Systems* section 23.14 for detailed descriptions of all states/transitions

THE SHELL

- IO
- Processes
- **The Shell**
  - **Examples, Standard IO**
- Main Unix Features

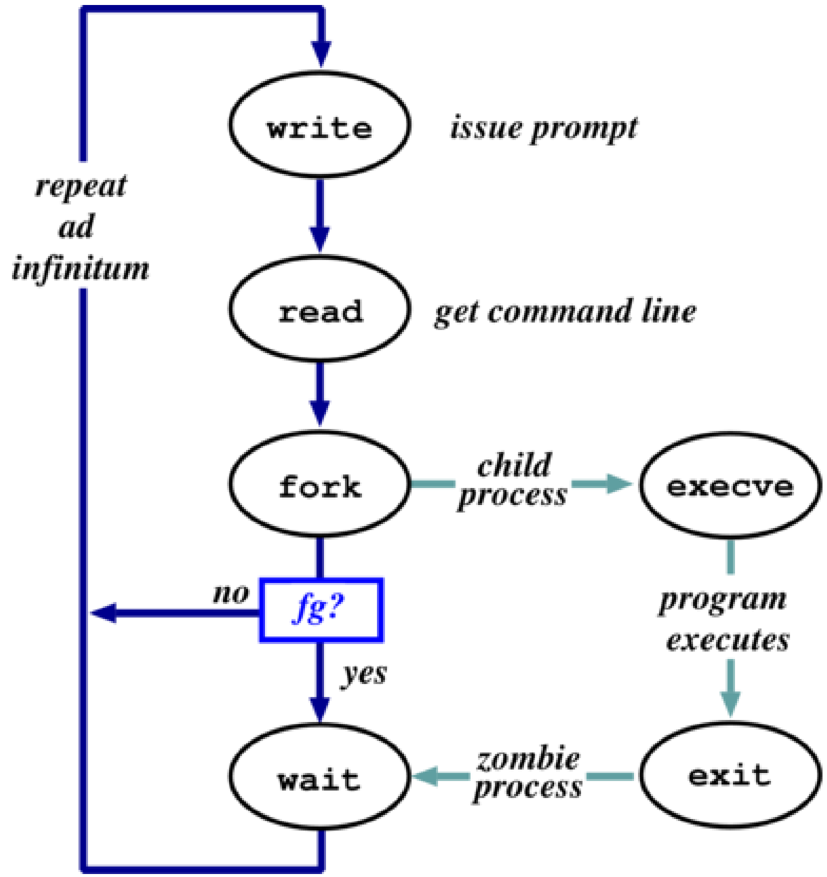
THE SHELL

Shell just a process like everything else.
Needn't understand commands, just files

Uses path for convenience, to avoid needing fully qualified pathnames

Conventionally `&` specifies background

Parsing stage (omitted) can do lots: wildcard expansion ("globbing"), "tilde" processing



SHELL EXAMPLES

```
$ pwd
/Users/mort/src
$ ls -F
awk-scripts/    karaka/         ocamllint/          sh-scripts/
backup-scripts/ mrt.0/          opensharingtoolkit/ sockman/
bib2x-0.9.1/    ocal/           pandoc-templates/   tex/
c-utils/        ocaml/          pttcp/              tmp/
dtrace/         ocaml-libs/     pyrt/               uon/
exapraxia-gae/  ocaml-mrt/      python-scripts/         vbox-bridge/
external/       ocaml-pst/      r/
junk/           ocaml.org/      scrapers/
$ cd python-scripts/
/Users/mort/src/python-scripts
$ ls -lF
total 224
-rw-r--r--   1 mort  staff  17987  2 Jan  2010 LICENSE
-rw-rw-r--   1 mort  staff   1692  5 Jan 09:18 README.md
-rwxr-xr-x   1 mort  staff   6206  2 Dec  2013 bberry.py*
-rwxr-xr-x   1 mort  staff   7286 14 Jul  2015 bib2json.py*
-rwxr-xr-x   1 mort  staff   7205  2 Dec  2013 cal.py*
-rw-r--r--   1 mort  staff   1860  2 Dec  2013 cc4unifdef.py
-rwxr-xr-x   1 mort  staff   1153  2 Dec  2013 filebomb.py*
-rwxr-xr-x   1 mort  staff   1059  2 Jan  2010 forkbomb.py*
```

Prompt is `$`. Use `man` to find out about commands. User friendly?

STANDARD IO

Every process has three fds on creation:

- `stdin`: where to read input from
- `stdout`: where to send output
- `stderr`: where to send diagnostics

Normally inherited from parent, but shell allows redirection to/from a file, e.g.,

- `ls >listing.txt`
- `ls >&listing.txt`
- `sh <commands.sh`

Consider: `ls >temp.txt; wc <temp.txt >results`

- Pipeline is better (e.g. `ls | wc >results`)
- Unix commands are often filters, used to build very complex command lines
- Redirection can cause some buffering subtleties

MAIN UNIX FEATURES

- IO
- Processes
- The Shell
- **Main Unix Features**

MAIN UNIX FEATURES

- File abstraction
  - A file is an unstructured sequence of bytes
  - (Not really true for device and directory files)
- Hierarchical namespace
  - Directed acyclic graph (if exclude soft links)
  - Thus can recursively mount filesystems
- Heavy-weight processes
- IO: block and character
- Dynamic priority scheduling
  - Base priority level for all processes
  - Priority is lowered if process gets to run
  - Over time, the past is forgotten
- But V7 had inflexible IPC, inefficient memory management, and poor kernel concurrency
- Later versions address these issues.

SUMMARY

- IO
  - Implementation, The Buffer Cache
- Processes
  - Unix Process Dynamics, Start of Day, Scheduling and States
- The Shell
  - Examples, Standard IO
- Main Unix Features