Lecture 13:

Unix II: Processes

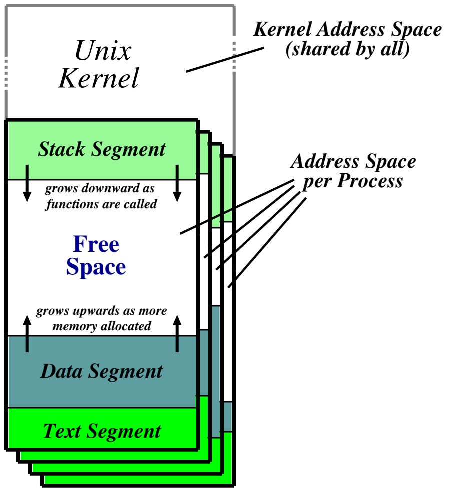
www.cl.cam.ac.uk/Teaching/2001/OSFounds/

Today's Lecture

Today we'll cover:

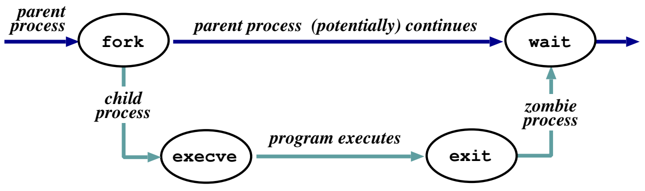
- Case Study: Unix Part II
  - Processes,
  - Shell, and
  - IPC: Pipes and signals.

Unix Processes



- Recall: a process is a program in execution.
- Have three **segments**: `text`, `data` and `stack`.
- Unix processes are heavyweight.

Unix Process Dynamics



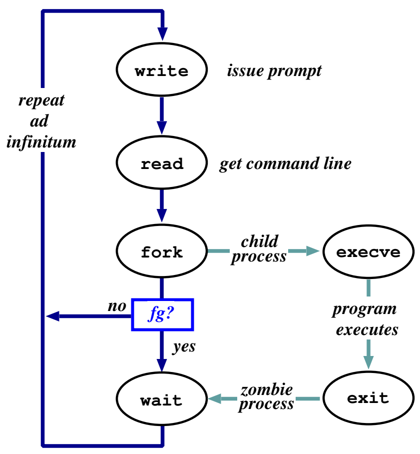
- Process represented by a **process id** (pid)
- Hierarchical scheme: parents create children.
- Four basic primitives:
  - *pid* = **fork** ()
  - reply = **execve**(*pathname*, *argv*, *envp*)
  - **exit**(*status*)
  - *pid* = **wait** (*status*)
- **fork()** nearly *always* followed by **exec()**

  ⇒ **vfork()** and/or COW.

Start of Day

- Kernel (/vmunix) loaded from disk (how?) and execution starts.
- Root file-system mounted.
- Process 1 (/etc/init) hand-crafted.
- init reads file /etc/inittab and for each entry:
  1. opens terminal - special file (e.g. /dev/tty0)
  2. duplicates the resulting fd twice.
  3. forks an /etc/tty process.
- each tty process next:
  1. initialises the terminal
  2. outputs the string "login:" & waits for input
  3. execve()'s /bin/login
- login then:
  1. outputs "password:" & waits for input
  2. encrypts password and checks it against /etc/passwd.
  3. if ok, sets uid & gid, and execve()'s shell.
- Patriarch init resurrects /etc/tty on exit.

The Shell



- Shell just a process like everything else.
- Uses **path** for convenience.
- Conventionally '&' specifies **background**.
- Parsing stage (omitted) can do lots. . .

Shell Examples

```
# pwd
/home/gmb
# ls -F
IRAM.micro.ps               gnome_sizes        prog-nc.ps
Mail/                       ica.tgz            rafe/
OSDI99_self_paging.ps.gz    lectures/          rio107/
TeX/                        linbot-1.0/        src/
adag.pdf                    manual.ps          store.ps.gz
docs/                       past-papers/
emacs-lisp/                 pbosch/
fs.html                     pepsi_logo.tif
# cd src/
# pwd
/home/gmb/src
# ls -F
cdq/         emacs-20.3.tar.gz  misc/      read_mem.c
emacs-20.3/  ispell/            read_mem*  rio007.tgz
# wc read_mem.c
     95     225    2262 read_mem.c
# ls -lF r*
-rwxrwxr-x   1 gmb  user    34956 Mar 21  1999 read_mem*
-rw-rw-r--   1 gmb  user     2262 Mar 21  1999 read_mem.c
-rw-------   1 gmb  user    28953 Aug 27 17:40 rio007.tgz
# ls -l /usr/bin/X11/xterm
-rwxr-xr-x   2 root   system 164328 Sep 24 18:21 /usr/bin/X11/xterm*
```

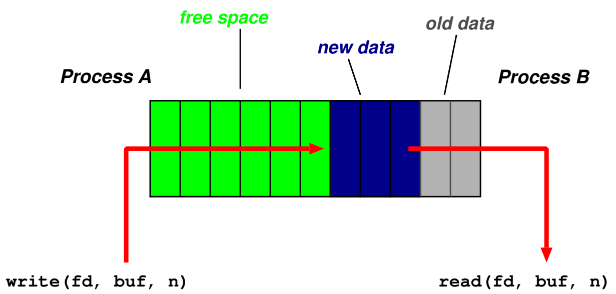
- Prompt is '#'.
- Use man to find out about commands.
- User friendly?

Standard I/O

- 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.
- Actual file not always appropriate; e.g. consider:

  ls >temp.txt;
  wc <temp.txt >results
- **Pipeline** is better (e.g. ls | wc >results)
- Most Unix commands are *filters* ⇒ can build almost arbitrarily complex command lines.
- Redirection can cause some buffering subtleties.

Pipes

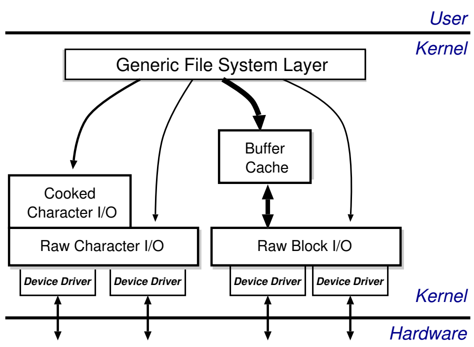


- One of the basic Unix IPC schemes.
- Logically consists of a pair of fds
- e.g. reply = **pipe**(int fds[2])
- Concept of "full" and "empty" pipes.
- Only allows communication between processes with a common ancestor. Why?
- Named pipes address this.

Signals

- Problem: pipes need planning ⇒ use **signals**.
- Similar to a (software) interrupt.
- Examples:
  - `SIGINT` : user hit Ctrl-C.
  - `SIGSEGV` : program error.
  - `SIGCHLD` : a death in the family. . .
  - `SIGTERM` : . . . or closer to home.
- Unix allows processes to **catch** signals.
- e.g. Job control:
  - `SIGTTIN`, `SIGTTOU` sent to bg processes
  - `SIGCONT` turns bg to fg.
  - `SIGSTOP` does the reverse.
- Cannot catch `SIGKILL` (hence `kill -9`)
- Signals can also be used for timers, window resize, process tracing, . . .

I/O Implementation



- Recall:
  - everything accessed via the file system.
  - two broad categories: block and char.
- Low-level stuff gory and machine dep. ⇒ ignore.
- Character I/O low rate but complex ⇒ most functionality in the "cooked" interface.
- Block I/O 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:
  1. Locate relevant blocks (from inode)
  2. Check if in buffer cache.
  3. If not, read from disk into memory.
  4. 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.
- Question: when does data actually hit disk?
- Answer: call `sync` every 30 seconds to flush dirty buffers to disk.
- Can cache metadata too — problems?

Unix Process Scheduling

- Priorities 0–127; user processes $\geq$ PUSER $= 50$.
- Round robin within priorities, quantum 100ms.
- Priorities are based on usage and *nice*, i.e.

$$P_j(i) = \texttt{PUSER} + \frac{CPU_j(i-1)}{4} + 2 \times nice_j$$

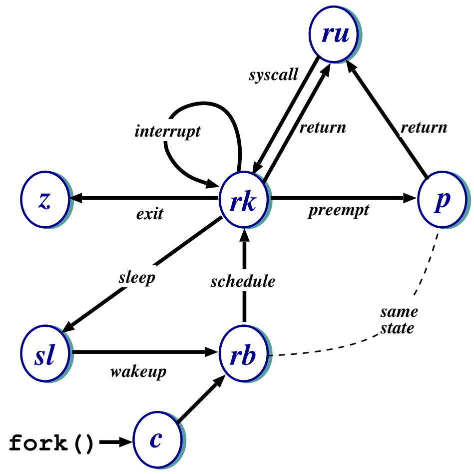
gives the priority of process j at the beginning of interval i where:

$$CPU_j(i) = \frac{2 \times load_j}{(2 \times load_j) + 1} CPU_j(i-1) + nice_j$$

and $nice_j$ is a (partially) user controllable adjustment parameter $\in [-20, 20]$.

- $load_j$ is the sampled average length of the run queue in which process j resides, over the last minute of operation
- so if e.g. load is 1 $\Rightarrow$ $\sim$ 90% of 1 seconds CPU usage "forgotten" within 5 seconds.

Unix Process States



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

- Note: above is simplified — see CS section 23.14 for detailed descriptions of all states/transitions.

Summary

- Main Unix features are:
  - 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)
    - can recursively mount filesystems
  - heavy-weight processes
  - IPC: pipes & signals
  - I/O: 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.

Next lecture: Case Study II: Windows NT