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:
  - \( pid = \text{fork}() \)
  - \( \text{reply} = \text{execve}(\text{pathname}, \text{argv}, \text{envp}) \)
  - \( \text{exit}() \)
  - \( pid = \text{wait}(\text{status}) \)
- \( \text{fork}() \) nearly **always** followed by \( \text{exec}() \)
  \( \Rightarrow \text{vfork}() \) and/or COW.
Start of Day

- Kernel (/vmlinux) 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.

Shell Examples

```bash
# pwd
/home/guth
# ls -l
RJAR микро.png genna.pie prog.png
Ma/l/ нвнw Бабаг.нь.png по. гп
Text/ login-1. o/ input/ store.png
Gnu/ mand-1.png/ готов.png
fg, init/ peg@_регa_tif
# cat gac
/home/guth/gac
# ls -l
udg/ macc2.37.png Chop/ read.png, c
udg/ macc2.37.png Chop/ read.png, c
w/ read.png, c
# 225 2207 read.png, c
# ls -l +
-rw-r--r-- 1 guth Ubuntu 14955 Mar 21 1989 read.png, c
-rw-r--r-- 1 guth Ubuntu 2267 Mar 21 1989 read.png, c
-rw------- 1 guth user 20983 Aug 27 17:40 read.png, c
# ls -l /usr/bin/ll/Atom
-1000- 2 root system 16426 Sep 24 18:21 /usr/bin/ll/Atom*
```

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:
  - SIGTIN, SIGTOUT sent to bg processes
  - SIGINT 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 ≥ \( \text{PUSER} = 50 \).
- Round robin within priorities, quantum 100ms.
- Priorities are based on usage and \textit{nice}, i.e.

\[
P_j(i) = \text{PUSER} + \frac{CPU_j(i-1)}{4} + 2 \times \text{nice}_j
\]

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

\[
CPU_j(i) = \frac{2 \times \text{load}_j}{2 \times \text{load}_j + 1} \times CPU_j(i-1) + \text{nice}_j
\]

and \( \text{nice}_j \) is a (partially) user controllable adjustment parameter \( \in [-20, 20] \).

- \( \text{load}_j \) 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 ⇒ ~ 90% of 1 seconds CPU usage "forgotten" within 5 seconds.

**Unix Process States**

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