Contents

Midterm Notes

Labs

- to see the return code: `echo $?`
- for syscall numbers `/usr/include/asm/unistd_64.h`

`true`

- returns 0 (C false)

`false`

- returns 1 (C true)

`yes`

- prints `y` or provided arguments separated by spaces forever
- use print, strlen

```
long strlen(const char *s) {
    long i;
```

```
    for (i = 0; s[i] != '\0'; i++) ;
    return i;
}
```

echo

- print space-separated arguments, ending in a newline

cat

- `openat` for file
- for each argument:

```
ssize_t bytes_read;
while((bytes_read = read(fd, buff, BUFF_SIZE)) > 0) {
    write(1, buff, bytes_read);
}
close(fd);
```

pwd

- `getcwd` syscall

touch

- if file does not exist: create empty file
  - `openat` with arguments – `O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK`
- if file exists – update time stamp
  - `openat` with arguments – `O_WRONLY|O_CREAT|O_NOCTTY|O_NONBLOCK`

mkdir

- `mkdir` with `0777` as argument
- user, group, other

mv

- `renameat2(AT_FDCWD, "test/", AT_FDCWD, "test-2/", RENAME_NOREPLACE) = 0`
- this is done for any type
- move all but the last file into the last location

rm

- for `rm -rf`

```
unlinkat(AT_FDCWD, "test-2/", AT_REMOVEDIR) = 0
```

- for `rm`

```
unlinkat(AT_FDCWD, "touch_2.c", 0) = 0
```

`cp`

- if `cp -r`
  - check if dir exists with fsstatat
    - if not, `mkdir`
  - for each file in the directory – open the file, read from it, then write to it
  - all with `openat`
- if just `cp`
  - stat both files to check existance
  - read from one file, write to the other one

`ls`

- `openat(AT_FDCWD, ".", O_RDONLY|O_NONBLOCK|O_CLOEXEC|O_DIRECTORY) = 3`
- then `fstat(3, {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0`
- `getdents64(3, 0x55dc15be8420 /* 9 entries */, 32768) = 224`
- then simply write the `names[]` to `stdout`

Kernel Project

`get_pids.c`

- see file

`get_task_info.c`

- see file

`task_info.h`

- see file

Presentations

Introduction

- most fundamental system program is the operating system
- it manages the computer's resources and provides a base for application development
- Hardware → Kernel → Applications → User
- it does resource management and machine abstraction
- resources:
  - CPU
  - memory
  - disks
  - other devices
- machine abstraction:
  - processes
  - threads

  - files
  - shell
- process, files systems, device abstraction can be modeled as trees
- system calls hide the complexity of the real machine
- single-tasking, multi-tasking, single-user, multi-user are what they sounds like
- real-time operating system – should not have a time delay on the input
- distributed – separate nodes that are networked
- embedded os – specific tasks for non-pcs
- library os – single address-space machine that has the bare minimum of stuff for programs to run and they are then compiled and run directly on hardware without an os
- microkernel: bare minimum runs in kernel space, the rest in user mode
- hybrid kernel: mix of things – Windows, OS X
- monolithic kernel: everything but applications run in kernel mode – Linux

History

1945 to 1955

- plug boards, punch cards

1955 to 1965

- transistors
- assembly languages
- programming languages
- jobs, job control systems, batch systems (jobs without user input than can be run as resources permit)

1965 to 1980

- integrated circuits
- major os projects: IBM OS/360, CTSS, MULTICS
- multi threading
- time sharing
- UNIX 1970, Ken Thompson, Dennis Ritchie
- CTSS -> Multics -> UNIX
- first in asm, then later in C

1980 to 1990

- large scale integration circuits – CPUs
- microcomputers
- PCs
- DOS (apple, microsoft,…)
- GNU, BSD

1990 to present

- internet

- mobile boom
- iot

Boot Process

1. power up
2. internal firmware
   - BIOS
   - UEFI
   - custom stuff
3. boot loader
   - multiple stages
   - MBR, GPT
4. kernel
   - CPU and hardware setup
5. initialization service
   - init, upstart, systemd, launchd
6. user space programs
   - login, shell

Processses

- executable format: structure of an exe
- produced by a linker
- format header, program header, then links to code, global and static data
- common formats: PE, ELF, Mach-O
- each process has: pid, state, priority, cpu state (registers, program counter, stack pointer), virtual memory (memory map), opened files, code, data, parent id, working directory, exit status,…
- the kernel maintains a list of all the process descriptors
- process states: running, ready, blocked for scheduling
- context switching to give all exes an equal amount of time
- processes can be created and exit when they are done
- io devices can interrupt, the clock can interrupt

Context Switching

1. save current cpu state
2. load a CPU state from next executable
3. run that one

Threads

- lightweight process that shares some elements with its parent process
- e.g. memory address space, opened files, global data, …
- they tend to have a private stack, registers, state
- same data structure for processes and threads

Scheduling

- either compute-bound or IO-bound processes
- the time-bottleneck is either computation or IO
- compute-bound: long CPU bursts, little IO
- IO-bound: short CPU bursts, a lot of IO
- a scheduler can start and stop these processes to make a efficient use of the system resources
- preemptive scheduling – mostly by priority
- non-preemptive goes by what processes finish when
- fairness and balance are common goals

Batch Scheduler

- maximize throughput
- minimize turnabout
- maximize CPU utilization
- first-come first-served
- shortest job first
- shortest remaining time next

Interactive Scheduler

- minimize response time
- round-robin scheduling
- shortest process next
- lottery scheduling
- fair-share scheduling

Real-time Scheduler

- meet deadlines
- hard real time vs soft real time:
    - hard real time have deadlines that must be met
    - soft real time has a certain tolerance but still not desirable
- periodic or non-periodic events