V22.0480-002 – Advanced Operating Systems

Instructor: David Mazières
715 Broadway, #708

TA: Jinyuan Li
719 Broadway, #714

Staff Email: `<os-staff@scs.cs.nyu.edu>`

Administrivia

- **All assignments are on the web page**
  `http://www.scs.cs.nyu.edu/aos/`
- **Part of each class will be spent discussing papers**
  - Read the papers before class
- **Grading based on three factors**
  - Participation in discussion (so read the papers before class!)
  - Midterm and Final Quiz
  - Lab assignments

Handouts today

- **Account information form**
  - Will give you access to dedicated class machines for lab
  - Accounts will be created by tomorrow
  - Email me if you don't hear from me by Friday
- **Access form for 7th floor of 715 Broadway**
  - So you can come to my office hours
  - Only if you don't already have access
- **First lab goes on-line soon**

Course topics

- User/kernel APIs
- Kernel architectures
- Virtual memory
- Threads
- IPC & Synchronization
- Scheduling
- I/O implementation
- File systems
- OS security

Lab assignments

- **Build a UNIX shell**
- **Build minimal OS for PC hardware**
  - Bootstrap code
  - Memory management
  - Processes
  - Context switches/IPC
  - File system
- **Port your shell to your operating system**
- **Demo your OS in last class**

OS Platform

- **Your OS will run on a standard PC**
  - x86 architecture (Pentium, Athlon, etc.)
  - IDE disk, standard console, etc.
- **Developed mostly in C, some assembly language**
  - Use GCC `asm` extension for inline assembly
- **Class web page contains many references for PC hardware**
- **Will test and run code using bochs**
  - Faithful PC hardware simulator
  - Much easier to debug on than real hardware
  - But what runs on Bochs will run on real hardware

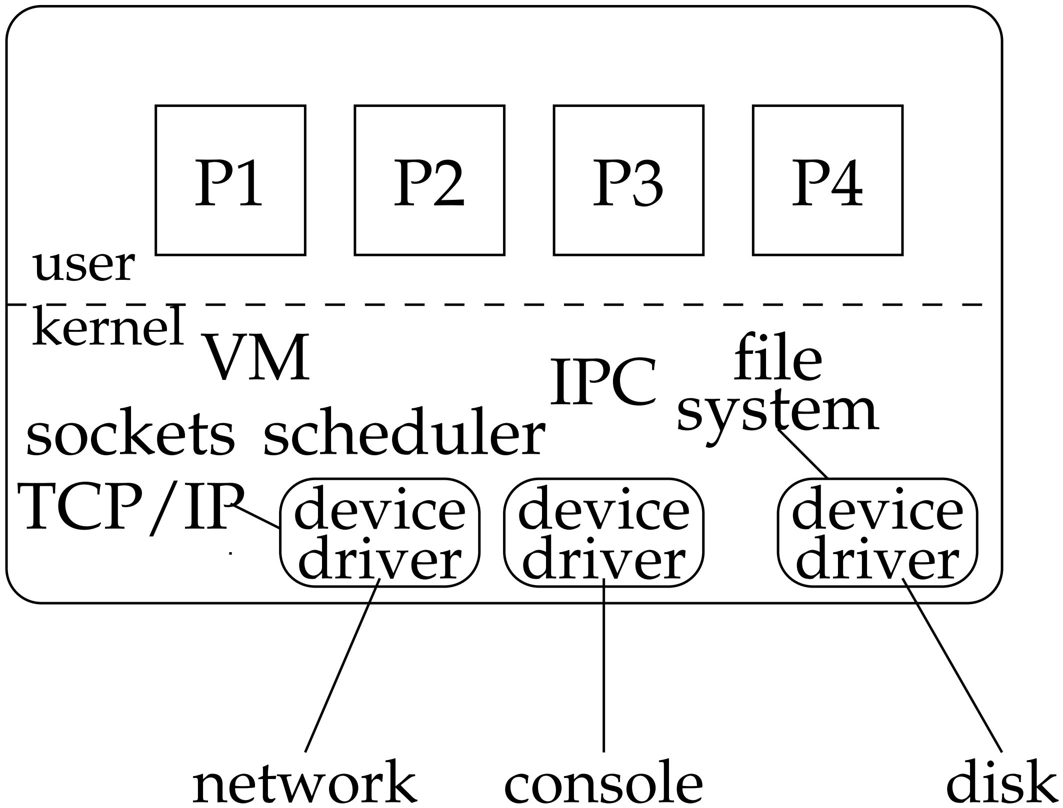
What is an operating system?

- **Makes hardware useful to the programmer**
- **Provides abstractions for applications**
  - Manages and hides details of hardware
  - Accesses hardware through low/level interfaces unavailable to applications
- **Provides protection**
  - Prevents one process/user from clobbering another

Why study operating systems?

- **Operating systems are a maturing field**
  - Most people use a handful of mature OSes
  - Hard to get people to switch operating systems
  - Hard to have impact with a new OS
- **High-performance servers are an OS issue**
  - Face many of the same issues as OSes
- **Resource consumption is an OS issue**
  - Battery life, radio spectrum, etc.
- **Security is an OS issue**
  - Hard to achieve security without a solid foundation
- **New "smart" devices need new OSes**

Typical OS structure



- **Most software runs as user-level processes**
- **OS kernel handles "privileged" operations**
  - Creating/deleting processes
  - Access to hardware

The different Unix contexts

- **User-level**
- **Kernel “top half”**
  - System call, page fault handler, kernel-only process, etc.
- **Software interrupt**
- **Device interrupt**
- **Timer interrupt (hardclock)**
- **Context switch code**

Transitions between contexts

- **User $\rightarrow$ top half: syscall, page fault**
- **User/top half $\rightarrow$ device/timer interrupt: hardware**
- **Top half $\rightarrow$ user/context switch: return**
- **Top half $\rightarrow$ context switch: sleep**
- **Context switch $\rightarrow$ user/top half**

Top/bottom half synchronization

- **Top half kernel procedures can mask interrupts**

```
int x = splhigh ();
/* ... */
splx (x);
```

- **splhigh disables all interrupts, but also splnet, splbio, splsoftnet, . . .**
- **Masking interrupts in hardware can be expensive**
  - Optimistic implementation – set mask flag on splhigh, check interrupted flag on splx

Kernel Synchronization

- **Need to relinquish CPU when waiting for events**
  - Disk read, network packet arrival, pipe write, signal, etc.
- `int tsleep(void *ident, int priority, ...);`
  - Switches to another process
  - `ident` is arbitrary pointer—e.g., buffer address
  - `priority` is priority at which to run when woken up
  - PCATCH, if ORed into `priority`, means wake up on signal
  - Returns 0 if awakened, or ERESTART/EINTR on signal
- `int wakeup(void *ident);`
  - Awakens all processes sleeping on `ident`
  - Restores SPL a time they went to sleep
    (so fine to sleep at splhigh)

V22.0480-002 Kernel

- **Asynchronous interface, not like UNIX**
  - Only one kernel stack
  - Interrupts always disabled in kernel (except in idle loop)
  - Kernel never sleeps (except in idle loop)
- **Why do away with threads in kernel?**
  - Vastly complicates programming (more error-prone)
  - Ill-suited to certain user-level applications
  - Conversely, can simulate traditional synchronous kernel interface at user-level in terms of asynchronous interface

System calls

- **Goal: invoke kernel from user-level code**
  - Like a library call, but into more privileged OS code
- **Applications request operations from kernel**
- **Kernel supplies well-defined *system call* interface**
  - Applications set up syscall arguments and *trap* to kernel
  - Kernel performs operation and returns result
- **Higher-level functions built on syscall interface**
  - `printf`, `scanf`, `gets`, etc. all user-level code
- **Example: POSIX/UNIX interface (rest of lecture)**
  - Your kernel system call interface will be lower-level
  - But can build POSIX-like functions in libraries

I/O through the file system

- **Applications "open" files/devices by name**
  - I/O happens through open files
- `int open(char *path, int flags, ...);`
  - `flags`: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - `O_CREAT`: create the file if non-existent
  - `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - `O_TRUNC`: Truncate the file
  - `O_APPEND`: Start writing from end of file
  - `mode`: final argument with `O_CREAT`
- **Returns file descriptor—used for all I/O to file**

Error returns

- **What if** open **fails? Returns -1 (invalid fd)**
- **Most system calls return -1 on failure**
  - Specific kind of error in global int errno
- #include <sys/errno.h> **for possible values**
  - 2 = ENOENT "No such file or directory"
  - 13 = EACCES "Permission Denied"
- perror **function prints human-readable message**
  - perror ("initfile");
    → "initfile: No such file or directory"

Device nodes

- **File namespace also gives access to some devices**
  - Open what looks like a file, to gain access to device
- **Examples (on my machine, others will vary):**
  - `/dev/null` – reads like EOF, writes like a data sink
  - `/dev/zero` – reads like an infinite stream of 0 bytes
  - `/dev/tty` – reads from or writes to current terminal
  - `/dev/rwd0c` – access raw disk sectors
  - `/dev/rcd0c` – CD-ROM device
  - `/dev/audio` – send audio samples to sound card
  - `/dev/wsmouse` – mouse
  - `/dev/bpf` – lets you snoop packets on the network

Permissions

- **Not every process can open every file**
- **Each process has a set of credentials**
  - User ID (typically 32-bit number, unique per login account)
  - Group ID, group list (32-bit numbers)
- **Files have permissions, too. E.g.,:**
  - (Link count = 1), User ID is 0, group ID 7

    ```
    -r-xr-xr-x  1 0  7  79 Apr 14 10:32 /usr/bin/true
    ```
- **Three sets of "rwx" bits, for user, group, and other**
  - read/write/execute on normal files
  - on directories, "x" means traverse (cd or access any file)
  - on dirs, must have "w" to create, rename, or delete files

Unix root user

- **Unix user ID 0 is privileged “root” user**
  - Can perform most system calls without access checks
  - E.g., open any file
  - Can change owner of files
  - Can Change its own UID or group list
- **Not to be confused with privileged kernel**
  - Kernel runs with CPU in special “privileged” mode
  - Allows access to special instructions, I/O registers, etc.
  - root-owned processes are still just regular user processes

Example: Unix login process

- **Login process runs with UID 0 (root)**
- **Asks for username and password**
  - Checks against system password file
  - Keeps asking until valid password supplied
- **Once password matches**
  - Look up numeric UID and GIDs in system files
  - Set the GID list
  - Set the UID (this drops privileges)
  - Execute the user's shell

Operations on file descriptors

- `int read (int fd, void *buf, int nbytes);`
  - Returns number of bytes read
  - Returns 0 bytes at end of file, or -1 on error
- `int write (int fd, void *buf, int nbytes);`
  - Returns number of bytes written, -1 on error
- `off_t lseek (int fd, off_t pos, int whence);`
  - `whence`: 0 – start, 1 – current, 2 – end
    - Returns previous file offset, or -1 on error
- `int close (int fd);`
- `int fsync (int fd);`
  - Guarantee that file contents is stably on disk

File descriptor numbers

- **File descriptors are inherited by processes**
  - When one process spawns another, same fds by default
- **Descriptors 0, 1, and 2 have special meaning**
  - 0 – "standard input" (`stdin` in ANSI C)
  - 1 – "standard output" (`stdout, printf` in ANSI C)
  - 2 – "standard error" (`stderr, perror` in ANSI C)
  - Normally all three attached to terminal

The rename system call

- `int rename (const char *p1, const char *p2);`
  - Changes name `p2` to reference file `p1`
  - Removes file name `p1`
- **Guarantees that `p2` will exist despite any crashes**
  - `p2` may still be old file
  - `p1` and `p2` may both be new file
  - but `p2` will always be old or new file
- **`fsync`/`rename` idiom used extensively**
  - E.g., emacs: Writes file `.#file#`
  - Calls `fsync` on file descriptor
  - `rename (".#file#", "file");`

Creating processes

- `int fork (void);`
  - Create new process that is exact copy of current one
  - Returns *process ID* of new proc. in "parent"
  - Returns 0 in "child"
- `int waitpid (int pid, int *stat, int opt);`
  - `pid` – process to wait for, or -1 for any
  - `stat` – will contain exit value, or signal
  - `opt` – usually 0 or `WNOHANG`
  - Returns process ID or -1 on error

Deleting processes

- `void exit (int status);`
  - Current process ceases to exist
  - `status` shows up in `waitpid` (shifted)
  - By convention, `status` of 0 is success, non-zero error
- `int kill (int pid, int sig);`
  - Sends signal `sig` to process `pid`
  - `SIGTERM` most common value, kills process by default (but application can catch it for “cleanup”)
  - `SIGKILL` stronger, kills process always

Running programs

- `int execve (char *prog, char **argv, char **envp);`
  - `prog` – full pathname of program to run
  - `argv` – argument vector that gets passed to `main`
  - `envp` – environment variables, e.g., `PATH`, `HOME`
- **Generally called through a wrapper functions**
- `int execvp (char *prog, char **argv);`
  - Search `PATH` for prog
  - Use current environment
- `int execlp (char *prog, char *arg, ...);`
  - List arguments one at a time, finish with `NULL`

Manipulating file descriptors

- `int dup2 (int oldfd, int newfd);`
  - Closes `newfd`, if it was a valid descriptor
  - Makes `newfd` an exact copy of `oldfd`
  - Two file descriptors will share same offset (`lseek` on one will affect both)
- `int fcntl (int fd, F_SETFD, int val)`
  - Sets *close on exec* flag if `val` = 1, clears if `val` = 0
  - Makes file descriptor non-inheritable by spawned programs

Example: run prog w. /dev/null stdin

```
if (!(pid = fork ())) {
  int fd = open ("/dev/null", O_RDONLY);
  if (fd > 0) {
    dup2 (fd, 0);
    close (fd);
  }
  execlp ("prog", "prog", "arg1", NULL);
  perror ("prog");
  _exit (1);
}
waitpid (pid, &stat, 0);
printf ("prog exited %snormally\n", stat ? "ab" : "");
```

[note: no error checking here]

Pipes

- `int pipe (int fds[2]);`
  - Returns two file descriptors in `fds[0]` and `fds[1]`
  - Writes to `fds[1]` will be read on `fds[0]`
  - When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - Returns 0 on success, -1 on error
- **Operations on pipes**
  - `read`/`write`/`close` – as with files
  - When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - When `fds[0]` closed, `write(fds[1])`:
    - Kills process with `SIGPIPE`, or if blocked
    - Fails with EPIPE