OS overview

Overview

- Goal of course:
    - Understand operating systems in detail by designing and implementing miminal OS
    - Hands-on experience with building systems ("Applying 6.033")
- What is an operating system?
    - a piece of software that turns the hardware into something useful
    - layered picture: hardware, OS, applications
    - Three main functions: fault isolate applications, abstract hardware, manage hardware
- Examples:
    - OS-X, Windows, Linux, *BSD, ... (desktop, server)
    - PalmOS Windows/CE (PDA)
    - Symbian, JavaOS (Cell phones)
    - VxWorks, pSOS (real-time)
    - ...
- OS Abstractions
    - processes: fork, wait, exec, exit, kill, getpid, brk, nice, sleep, trace
    - files: open, close, read, write, lseek, stat, sync
    - directories: mkdir, rmdir, link, unlink, mount, umount
    - users + security: chown, chmod, getuid, setuid
    - interprocess communication: signals, pipe
    - networking: socket, accept, snd, recv, connect
    - time: gettimeofday
    - terminal:
- Sample Unix System calls (mostly POSIX)
    - int read(int fd, void*, int)
    - int write(int fd, void*, int)
    - off_t lseek(int fd, off_t, int [012])
    - int close(int fd)
    - int fsync(int fd)
    - int open(const char*, int flags [, int mode])
        - O_RDONLY, O_WRONLY, O_RDWR, O_CREAT
    - mode_t umask(mode_t cmask)
    - int mkdir(char *path, mode_t mode);
    - DIR *opendir(char *dirname)
    - struct dirent *readdir(DIR *dirp)
    - int closedir(DIR *dirp)
    - int chdir(char *path)
    - int link(char *existing, char *new)
    - int unlink(char *path)
    - int rename(const char*, const char*)

    - int rmdir(char *path)
    - int stat(char *path, struct stat *buf)
    - int mknod(char *path, mode_t mode, dev_t dev)
    - int fork()
        - returns childPID in parent, 0 in child; only difference
    - int getpid()
    - int waitpid(int pid, int* stat, int opt)
        - pid==-1: any; opt==0||WNOHANG
        - returns pid or error
    - void _exit(int status)
    - int kill(int pid, int signal)
    - int sigaction(int sig, struct sigaction *, struct sigaction *)
    - int sleep (int sec)
    - int execve(char* prog, char** argv, char** envp)
    - void *sbrk(int incr)
    - int dup2(int oldfd, int newfd)
    - int fcntl(int fd, F_SETFD, int val)
    - int pipe(int fds[2])
        - writes on fds[1] will be read on fds[0]
        - when last fds[1] closed, read fds[0] retursn EOF
        - when last fds[0] closed, write fds[1] kills SIGPIPE/fails EPIPE
    - int fchown(int fd, uind_t owner, gid_t group)
    - int fchmod(int fd, mode_t mode)
    - int socket(int domain, int type, int protocol)
    - int accept(int socket_fd, struct sockaddr*, int* namelen)
        - returns new fd
    - int listen(int fd, int backlog)
    - int connect(int fd, const struct sockaddr*, int namelen)
    - void* mmap(void* addr, size_t len, int prot, int flags, int fd, off_t offset)
    - int munmap(void* addr, size_t len)
    - int gettimeofday(struct timeval*)

See the reference page for links to the early Unix papers.

Class structure

- Lab: minimal OS for x86 in an exokernel style (50%)
    - kernel interface: hardware + protection
    - libOS implements fork, exec, pipe, ...
    - applications: file system, shell, ..
    - development environment: gcc, bochs
    - lab 1 is out
- Lecture structure (20%)
    - homework
    - 45min lecture
    - 45min case study

- Two quizzes (30%)
  - mid-term
  - final's exam week

Case study: the shell (simplified)

- interactive command execution and a programming language
- Nice example that uses various OS abstractions. See Unix paper if you are unfamiliar with the shell.
- Final lab is a simple shell.
- Basic structure:

```
        while (1) {
            printf ("$");
            readcommand (command, args);   // parse user input
            if ((pid = fork ()) == 0) {  // child?
               exec (command, args, 0);
            } else if (pid > 0) {   // parent?
               wait (0);   // wait for child to terminate
            } else {
               perror ("Failed to fork\n");
            }
        }
```

  The split of creating a process with a new program in fork and exec is mostly a historical accident. See the assigned paper for today.

- Example:

```
        $ ls
```

- why call "wait"? to wait for the child to terminate and collect its exit status. (if child finishes, child becomes a zombie until parent calls wait.)
- I/O: file descriptors. Child inherits open file descriptors from parent. By convention:
  - file descriptor 0 for input (e.g., keyboard). read_command:
  - `read (1, buf, bufsize)`
  - file descriptor 1 for output (e.g., terminal)
  - `write (1, "hello\n", strlen("hello\n")+1)`
  - file descriptor 2 for error (e.g., terminal)
- How does the shell implement:

```
      $ls > tmp1
```

  just before exec insert:

```
            close (1);
     fd = open ("tmp1", O_CREAT|O_WRONLY);   // fd will be 1!
```

  The kernel will return the first free file descriptor, 1 in this case.

- How does the shell implement sharing an output file:
- ```
  $ls 2> tmp1 > tmp1
  ```

  replace last code with:

  ```
  close (1);
  close (2);
  fd1 = open ("tmp1", O_CREAT|O_WRONLY);   // fd will be 1!
  fd2 = dup (fd1);
  ```

  both file descriptors share offset

- how do programs communicate?
- ```
  $ sort file.txt | uniq | wc
  ```

  or

  ```
  $ sort file.txt > tmp1
  $ uniq tmp1 > tmp2
  $ wc tmp2
  $ rm tmp1 tmp2
  ```

  or

  ```
  $ kill -9
  ```

- A pipe is an one-way communication channel. Here is an example where the parent is the writer and the child is the reader:
- ```
  int fdarray[2];

  if (pipe(fdarray) < 0) panic ("error");
  if ((pid = fork()) < 0) panic ("error");
  else if (pid > 0) {
    close(fdarray[0]);
    write(fdarray[1], "hello world\n", 12);
   } else {
    close(fdarray[1]);
    n = read (fdarray[0], buf, MAXBUF);
    write (1, buf, n);
   }
  ```
- How does the shell implement pipelines (i.e., cmd 1 | cmd 2 |..)? We want to arrange that the output of cmd 1 is the input of cmd 2. The way to achieve this goal is to manipulate stdout and stdin.
- The shell creates processes for each command in the pipeline, hooks up their stdin and stdout correctly. To do it correct, and waits for the last process of the pipeline to exit. A sketch of the core modifications to our shell for setting up a pipe is:
-
  ```

```
int fdarray[2];

if (pipe(fdarray) < 0) panic ("error");
if ((pid = fork ()) == 0) {  child (left end of pipe)
   close (1);
   tmp = dup (fdarray[1]);   // fdarray[1] is the write end, tmp will be 1
   close (fdarray[0]);       // close read end
   close (fdarray[1]);       // close fdarray[1]
   exec (command1, args1, 0);
} else if (pid > 0) {        // parent (right end of pipe)
   close (0);
   tmp = dup (fdarray[0]);   // fdarray[0] is the read end, tmp will be 0
   close (fdarray[0]);
   close (fdarray[1]);       // close write end
   exec (command2, args2, 0);
} else {
   printf ("Unable to fork\n");
 }
```

- Why close read-end and write-end? multiple reasons: maintain that every process starts with 3 file descriptors and reading from an empty pipe blocks reader, while reading from a closed pipe returns end of file.
- How do you background jobs?
- `$ compute &`
- How does the shell implement "&", backgrounding? (Don't call wait immediately).
- More details in the shell lecture later in the term.