

# Using an Operating System (CSE 422S)

Ken Wong  
Washington University

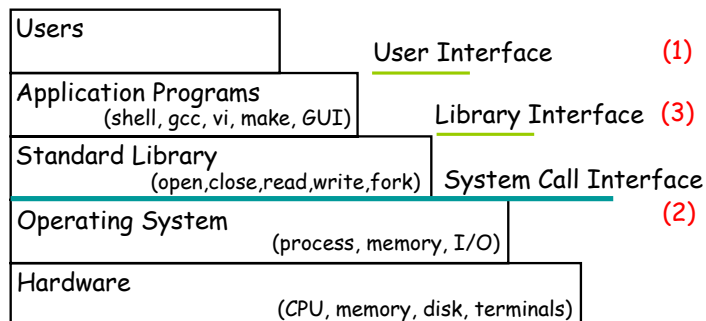
kenw@wustl.edu  
www.arl.wustl.edu/~kenw

## User's View of an OS

- An OS is a collection of system programs
  - » Allows user to build, run and debug programs (tools)
  - » Provides the user with an **abstract machine**
- Tools
  - » Compilers, Linker, Loader, Shell
  - » Debuggers, System Administration (e.g., file statistics)
- Abstractions
  - » Isolated, linear address space for each process
  - » Almost unlimited main memory (e.g., 32-bit addressing)
  - » Large, persistent, linear storage facilities
  - » Process control (start/stop/resume) and scheduling
  - » Controlled resource sharing

## UNIX Interfaces

Man Section



## Unix Man Pages

Name

open – open or create a file

SYNOPSIS

```
#include <sys/types.h>
```

```
#include <sys/stat.h>
```

```
#include <fcntl.h>
```

```
int open(const char *path, int flags);
```

```
int open(const char *path, int flags, mode_t mode);
```

DESCRIPTION

RETURN VALUE

open returns the new file descriptor, or -1 if ...

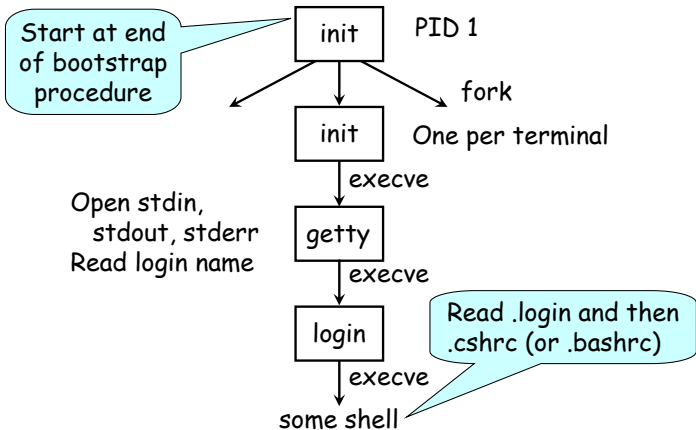
SEE ALSO

chmod(2), close(2), dup(2), lseek(2) ...

■ Linux: man 2 open      Solaris: man -s 2 open

■ 'man N intro' is man page for section N

## init And getty Processes



5 - Ken Wong, 1/24/2007

Washington University in St. Louis

## The UNIX Process Concept (1)

- A **process** is a program in execution
  - » Needs system resources (e.g., CPU, memory) to execute
- Every process has a unique **process ID (PID)**
  - » PID 0: Scheduler `sched` or `swapper`, a kernel process
  - » PID 1: `/sbin/init` started at the end of the bootstrap process and never dies
    - All other processes are descendants of `init`
  - » PID 2: Page Daemon `pageout` or `pagedaemon`
- Memory Layout ('`nm -ng a.out`')
  - » Text (Instructions)
  - » Heap (Global Variables)
  - » Stack (Local Variables)
  - » Command-line args, environment variables

6 - Ken Wong, 1/24/2007

Washington University in St. Louis

## The UNIX Process Concept (2)

- The execution environment is **multiprogrammed**
  - » Each process has a set of register values
  - » CPU only has one set of registers which is shared by all processes
    - OS kernel loads hardware registers with the register values of the currently running process
  - » Processes contend for system resources (CPU, memory, I/O devices)

7 - Ken Wong, 1/24/2007

Washington University in St. Louis

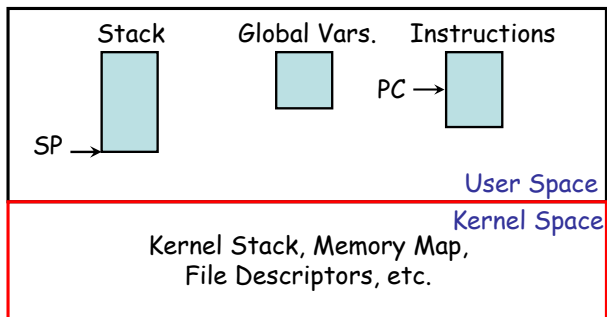
## Process Creation Events

- System initialization (created during bootup)
  - » *Background Processes: Daemons* (email, Web, printer)
  - » *Foreground Processes:* Interact with user (login-shell)
- Process creation system call (e.g., `fork`)
  - » Ex. 1: Concurrent server application forks child to handle request
  - » Ex. 2: Run `make` utility in parallel mode (`make -j 4`)
- User request
  - » Each command to a shell
- Batch job initiation
  - » Ex: User schedules job startup for 2AM using `crontab` or `at` command

8 - Ken Wong, 1/24/2007

Washington University in St. Louis

## Traditional Process



- A unit of *resource ownership* (address space, Files)
- A unit of *dispatching*
  - » Has an execution state; scheduled by the OS

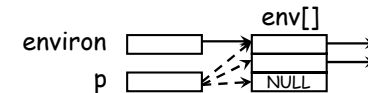
9 - Ken Wong, 1/24/2007

Washington University in St. Louis

## Command Line Args and Env

```
#include <unistd.h> // environ
#include <stdlib.h> // getenv
#include <stdio.h> // printf
int
main (int argc, char *argv[ ], char *env[ ]) {
    extern char **environ;
    for (int i=0; i<argc; i++) { printf("%2d %s\n", i, argv[i]); }
    for (int i=0; env[i] != NULL; i++) { printf("%s\n", env[i]); }
    for (char **p=environ; *p != NULL; p++) { printf("%s\n", *p); }
    printf("HOME=%s\n", getenv("HOME"));
}
```

Library Function



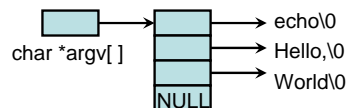
10 - Ken Wong, 1/24/2007

Washington University in St. Louis

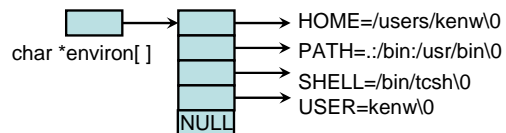
## Command Line Args and Environment

cec> /bin/echo Hello, World

### Command Line Arguments



### Environment Variables



11 - Ken Wong, 1/24/2007

Washington University in St. Louis

## UNIX Shells (1)

- A shell is a command line interpreter
  - » Translates commands typed at a terminal (or in a file)
  - » Input is from the terminal (stdin) or file
  - » User's /etc/passwd entry indicates the interactive shell
- Some Shells
  - » Bourne Shell (sh): What I use for shell scripts
  - » C-Shell, TC-Shell (csh, tcsh): Typical interactive shell
  - » Bash Shell (bash): Default Linux shell (superset of sh)
- Environment Variables (see 'printenv')
  - » PATH: Search path for executables
  - » PWD: Current working directory
  - » Others: SHELL, DISPLAY, MANPATH, DOMAIN, ...

12 - Ken Wong, 1/24/2007

Washington University in St. Louis

## Unix Shells (2)

- Have redirection
  - » Stdin Cmnd < File
  - » Stdout Cmnd > File
  - » Append Cmnd >> File
  - » Shell input Cmnd << EndWord
- Subshells "(...)"
- Pipeline Concept (e.g., C1 | C2 | C3 ...)
- Arbitrary Variables and Assignment
- Command Substitution (e.g., `pwd`)
- Control Structures (e.g., for, case, while, if)
- Scripting

13 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Unix Commands

### ■ A Command Sequence

```
grep -i 'cs.*422' 422.email* > x1 # extract info
grep -i -v subj < x1 > x2 # del subj lines
cut -f 2,3 x2 > x3 # retain col 2,3
sort x3 > 422roster.txt # sort last name
```

### ■ Pipeline

```
grep -i 'cs.*422' 422.email* | grep -i -v subj | \
cut -f 2,3 x2 | sort x3 > 422roster.txt &
```

### ■ A Shell Script (Usage: `mkroster 422.email\* &`)

```
#!/bin/sh
grep -i 'cs.*422' $* | grep -i -v subj | \
cut -f 2,3 x2 | sort x3 > 422roster.txt
```

14 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Unix Command Notes

- Command Sequence results in sequential execution of programs (Processes)
  - » Arguments are passed into process via argv[ ]
  - » '\*' is a file matching symbol that matches 0 or more characters
  - » Quote symbols are used to nullify the meaning of metasymbols
  - » stdin, stdout, and redirection
- Alternative Commands
  - » Pipe
  - » Background Job
  - » 'ps' command allows you to inspect status of processes
  - » 'jobs' command allows you to inspect status of backgrounded jobs
- Shell Script
  - » First line of shell script can be used to indicate command interpreter
  - » Need to make the script executable ('chmod +x mkroster') or execute as 'sh mkroster 422.email\* &'
  - » Can also: 1) Check on the exit status of a process; 2) Wait for a background process to finish

15 -Ken Wong, 1/24/2007

Washington University in St. Louis

## A Bourne Shell Script

```
01 #!/bin/sh # which interpreter
02 # Usage: t_proc.sh
03 usage ( ) { # function definition
04     echo "Usage: t_proc.sh"; exit 1;
05 }
06 # global variable value
07 t_proc ( ) {
08     echo "=====" "t_proc $C 1000";
09     t_proc $C 1000;
10 }
11 while true # the command /bin/true; exit code = 0
12 do
13     read cmd;
14     case $cmd in # end of case
15         f|fo|for|fork) echo "do_fork: "; C="fork"; do t; ;;
16         e|ex|exe|exec) echo "do_execve: "; C="execve"; do t; ;;
17         s|sh) echo "do_sh: "; C="sh"; do t; ;;
18         q|qu|qui|quit) break;
19         *) echo '*** Invalid Command' $cmd; ;;
20     esac
21 done # pattern matching
```

Only 1 data type: string !!!

16 -Ken Wong, 1/24/2007

Washington University in St. Louis

## UNIX System Calls (1)

- See Section 2 of the Manual Pages
- Process Management
  - » `pid = fork()`: Create a child process identical to parent
  - » `pid = wait(&status)`: Wait for any child to terminate
  - » `status = exec(path, arg0, ...)`: Replace a process image
  - » `exit(status)`: Terminate process and return status
- File Management
  - » `fd = open(path, how, ...)`: Open a file for reading, writing, or both
  - » `status = close(fd)`: Close a file
  - » `m = read(fd, buf, n)`: Read `n` bytes from a file into buffer
  - » `m = write(fd, buf, n)`: Write `n` bytes to a file from buffer
  - » `offset = lseek(fd, offset, whence)`: Move the file pointer

17 - Ken Wong, 1/24/2007

Washington University in St. Louis

## UNIX System Calls (2)

- Directory and File System Management
  - » `status = mkdir(path, mode)`: Create a new directory
  - » `status = rmdir(path)`: Remove an empty directory
  - » `status = link(path1, path2)`: Create new file `path2` pointing to `path1`
  - » `status = unlink(path)`: Remove directory entry
  - » `status = mount(special, path, flag)`: Mount a file system
- Miscellaneous
  - » `status = chdir(path)`: Change working directory
  - » `status = chmod(path, mode)`: Change a file's access permissions
  - » `status = kill(pid, signal)`: Send a signal to a process
  - » `status = gettimeofday(&timeval, 0)`: Get time of day

18 - Ken Wong, 1/24/2007

Washington University in St. Louis

## UNIX fork and wait

```
#include "myinc.h" // <sys/types.h>, <unistd.h>
int status;
if ( (pid_t) pid = fork( ) < 0 ) ... fork error ...
else if ( pid == 0 ) { // child process
    ... child code ...
    exit(0);
}
if ( wait(&status) != pid ) ... wait error ...
... parent continues here ...
```

} child code

} parent code

- fork
  - » Create a child process that is a "copy" of parent
  - » Returns 0 in child; PID of child in parent; -1 if error
- wait
  - » Blocks if all children are still running
  - » Returns PID of "a" child; -1 if no children

19 - Ken Wong, 1/24/2007

Washington University in St. Louis

## UNIX exec

```
#include "myinc.h" // <sys/wait.h>
if ( (pid_t) pid = fork( ) < 0 ) ... fork error ...
else if ( pid == 0 ) { // child process
    if ( execl("/tmp/mytest", "mytest", "foo.txt", (char *) 0) < 0 )
        { ... execl error ... }
    ... child should not get here ...
}
... parent continues here ...
```

- execl
  - » Replace existing process by a new process
  - » Begin execution at its main function
  - » Return -1 if an error; else no return if successful
  - » Various forms of exec (`execv`, `execl`, `execvp`)
    - 'l' stands for list argument format; 'v' stands for vector format
    - 'e' means that the environment is also passed as a vector
    - ➔ • 'p' means to search PATH for executable if 1<sup>st</sup> arg isn't pathname

20 - Ken Wong, 1/24/2007

Washington University in St. Louis

## fork, exec and the shell (1)

### ■ Example: Interactive Commands

```
cec> fib 10 > fib10.out
cec> fib 2000 > fib2000.out &
```

### ■ Your interactive shell (tcsh, bash) interprets commands

### ■ Shell processing of 'fib 10'

- » Shell looks for the 'fib' executable in a directory listed in the PATH environment variable **colon separates pathnames**
  - e.g., `"/usr/home/kenw/bin:/usr/bin:/usr/local/bin"`
- » Forks a copy of the shell; i.e., creates a child process
- » Child redirects stdout to the file 'fib10.out'
- » Child execs 'fib' passing in command-line arguments
  - 'fib' process replaces the shell's child process
- » Parent shell waits for 'fib' process to terminate
- » Display prompt after 'fib' terminates

21 -Ken Wong, 1/24/2007

Washington University in St. Louis

## fork, exec and the shell (2)

### ■ Shell processing of 'fib 2000 > fib2000.out &'

- » Processing is essentially the same as for 'fib 10 ...' except that parent does not wait for child to terminate

### ■ Extended Example:

- » Shell later waits for child

```
> fib 10 > fib10.out
> fib 2000 > fib2000.out &
> B=$!      # process # of last bg command
... other commands ...
> wait $B
```

22 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Unix I/O Devices

### ■ All I/O devices look like byte arrays

- » Can be accessed with the same read/write calls
- » Each device has a special file path name (e.g., /dev/sd0a, /dev/tty0, /dev/lp)

### ■ Block Special File (usually used for disks)

- » A sequence of numbered blocks
- » A block can be accessed directly (jump to position)

### ■ Character Special File

- » Character stream devices (keyboard, printer, mouse)
- » Must access characters sequentially (not directly)

### ■ There is a device driver for handling each special file

23 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Buffered and Unbufferd I/O (1)

### ■ Unbuffered I/O Using Stdin/Stdout

```
#include <unistd.h>
int main (void) {
    int n;    char buf[N];
    while ( (n = read(STDIN_FILENO,buf,N)) > 0)
        if (write(STDOUT_FILENO,buf,n) != n) ... error ...
    if (n < 0) ... error ...
}
```

File Descriptor (non-neg integer)

### ■ Buffered Version (Adds concept of a line, etc )

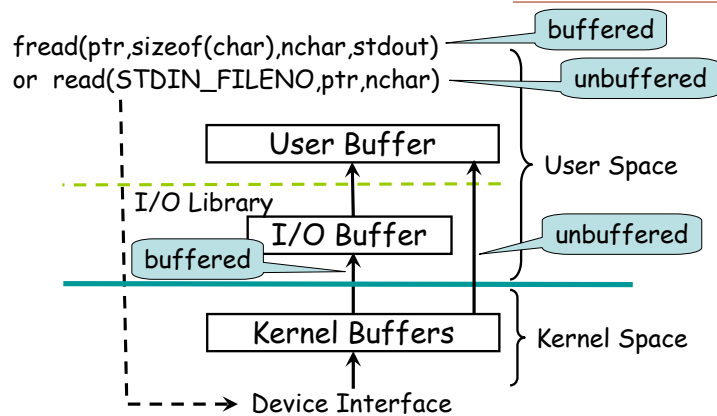
```
#include <stdio.h>
int main (void) {
    char line[N];
    while ( (fgets(line,N,stdin)) != EOF)
        if (fputs(line,stdout) == EOF) { ... error ... }
}
```

I/O Stream (ptr: FILE \*)

24 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Buffered and Unbuffered I/O (2)



25 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Unix Input and Output

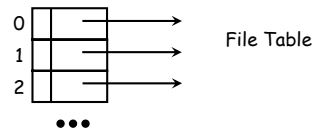
- File Descriptor (Unbuffered I/O)
  - » Small non-negative integer (file descriptor table index)
  - » Every successful file open returns a file descriptor
- All shells open 3 file descriptors
  - » (0) stdin, (1) stdout, (2) stderr
  - » All 3 are normally connected to the terminal
- stdin, stdout, and stderr can be redirected
  - » e.g., `ls > outfile`
  - » Append: `ls >> outfile`

26 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Unix Unbuffered File I/O

- Most I/O can be done with five functions
  - » `open`, `read`, `write`, `lseek`, `close`
  - » Can only access open files
- Refer to an open file through its file descriptor
  - » A file descriptor is a non-negative integer
    - Returned from `open` and `create`
    - Initially, 0, 1 and 2 always refer to `stdin`, `stdout` and `stderr`
      - » Should use `STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`
      - » Defined in `<unistd.h>`; i.e., `/usr/include/unistd.h`
  - » A file descriptor is an index into a table



27 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Unbuffered I/O to a File

```
#include <unistd.h> // write
#include <stdlib.h> // exit
#include <stdio.h> // fprintf
#include <sys/types.h> // open
#include <sys/stat.h> // .
#include <fcntl.h> // ... e.g., O_CREAT
int main (int argc, char *argv[]) {
    int fd;
    char msg[] = "hello";
    fd = open("./junk.txt", O_CREAT | O_WRONLY,
              S_IRUSR | S_IWUSR);
    if (fd < 0) { ... Error ... }
    rc = write(fd, &msg, sizeof(msg));
    if (rc != sizeof(msg)) { ... Error ... }
    close(fd);
}
```

Annotations in the code block:

- pathname**: points to `./junk.txt`
- flags**: points to `O_CREAT | O_WRONLY`
- mode**: points to `S_IRUSR | S_IWUSR`
- equal to 4 here**: points to the value `4` in the `open` function call.

28 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Files

### Types of Files

- » Regular File: Unstructured sequence of bytes
- » Directory: A set (grouping) of files
- » Link (Hard or Symbolic): Points to another file
- » FIFO: Used for communication between processes
- » Special File (Block or Character): A device file

### File Names

- » Absolute Pathname: Begins with "/"
- » Relative Pathname: Doesn't begin with "/"
  - Begin searching for file in current directory
  - "." means current directory; ".." means parent directory

### File Attributes ('ls' command or stats() call)

- » Access mode, Size (bytes)
- » Number of links, Owner name, Group name
- » Date/Time last accessed, modified

29 -Ken Wong, 1/24/2007

Washington University in St. Louis

## File Access Permissions

```
> ls -l /etc/passwd
-rw-r--r-- 1 root root 1481 Jul 18 2005 /etc/passwd
```

file type

mode bits

### 9 permission bits for each file

- » read/write/execute for user (owner), group and other
- » Modified by chmod command or system call
  - e.g., `chmod 664 junk.txt # rw-rw-r--`
  - e.g., `chmod ug-w junk.txt # r--r--r--`; i.e., remove write perm
- » To open `/usr/include/unistd.h`
  - Need execute permission for directories `/`, `/usr`, `/usr/include`
    - i.e., Execute permission means search permission here
  - Then need read or write permission for `/usr/include/unistd.h`
- » To create/delete a file in `/usr/include`
  - Need write and execute permission in `/usr/include`

30 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Unix UID And GID

### UID and GID

- » Every user has a unique User ID (UID) in password file
- » Every user belongs to at least one group, each with a unique Group ID (GID) in `/etc/group` file
- » Affects file access permissions

### The 'root' user or superuser

- » UID = 0
- » Can access files regardless of file protection
- » Can run restricted commands (e.g., `dump`, `fsck`)

### Effective UID and GID

- » Normally same as UID and GID
- » Change by `su`'ing to another user account; or Run a command that has `setuid` or `setgid` flag set
  - e.g., `"-r-s-x-x root root /usr/bin/passwd"`
  - `chmod 4611 myfile` or `"chmod u=srwx,g=x,o=x myfile"`

31 -Ken Wong, 1/24/2007

Washington University in St. Louis

## Summary

### A shell is a command-line interpreter

- » Functions like a macro processor (eval by substitution)
- » Builtin versus Non-builtin commands
- » Ideas: Process, redirection, pipeline, concurrency

### Unix interfaces

- » Commands, libraries, system calls

### Unbuffered versus Buffered I/O

- » Byte stream associated with file descriptor or stream

### Files

- » Have a pathname, even devices
- » Look like byte streams
- » Type, Access mode (user (owner), group, other)

32 -Ken Wong, 1/24/2007

Washington University in St. Louis