OS abstractions

2/25/08

Frans Kaashoek
MIT
kaashoek@mit.edu

Why a code-based OS class?

- Operating systems can be misleadingly simple
  - Clean simple abstractions, easily understandable in isolation
  - Complexity is in how their implementations interact
- Learn by doing, focus on interactions
  - How do hardware interrupts interact with kernel and user-level processes?
  - How to use locks to coordinate different activities?
- This lecture: OS abstractions
  - Illustrated by an shell implementation

sh: shell

- Interactive command interpreter
- Interface (“the shell”) to the operating system
- Examples of shell commands:
  - $ ls # create *process*
  - $ ls > tmp1 # write output to *file*
  - $ sh < script > tmp1 # run sh script
  - $ sort tmp | uniq | wc # process communicate with *pipe*
  - $ compute-pi & # run program in background
  - $ ….

OS ideas: *isolation*, *concurrency*, *communication, synchronization*

shell implementation

```
while (1) {
  printf("$");
  readcommand(command, args);
  pid = fork();                    // new process; concurrency
  if (pid == 0) {                    // child?
     exec (command, args, 0);  // run command
  } else if (pid > 0) {   // parent?
     r = wait (0);                   // wait until child is done
  } else {
     perror("Failed to fork\n");
  }
}
```

Input/Output (I/O)

- I/O through file descriptors
  - File descriptor may be for a file, terminal, …
- Example calls;
  - read(fd, buf, sizeof(buf));
  - write(fd, buf, sizeof(buf));
- Convention:
  - 0: input
  - 1: output
  - 2: error
- Child inherits open file descriptors from parents

I/O redirection

- Example: “ls > tmp1”
- Modify sh to insert before exec:

```
close(1);                          // release fd 1
fd = create(“tmp1”, 0666);         // fd will be 1
```

- No modifications to “ls”!
- “ls” could be writing to file, terminal, etc., but programmer of “ls” doesn’t need to know

Pipe: one-way communication

```
int fdarray[2];
char buf[512];
int n;

pipe(fdarray);                          // returns 2 fd's
write(fdarray[1], "hello", 5);
read(fdarray[0], buf, sizeof(buf));
```

- buf contains ‘h’, ‘e’, ‘l’, ‘l’, ‘o’

Pipe between parent & child

```
int fdarray[2];
char buf[512];
int n, pid;

pipe(fdarray);
pid = fork();
if(pid > 0) {
    write(fdarray[1], "hello", 5);
} else {
    n = read(fdarray[0], buf, sizeof(buf));
}
```

- *Synchronization* between parent and child
  - read blocks until there is data
- How does the shell implement “a | b”?

Implementing shell pipelines

```
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");
 }
```

OS abstractions and ideas

- Processes (fork & exec & wait)
- Files (open, create, read, write, close)
- File descriptor (dup, ..)
- Communication (pipe)
- Also a number of OS ideas:
  - Isolation between processes
  - Concurrency
  - Coordination/Synchronization

Your job: implement abstractions and understand ideas

What will you know at the end?

- Understand OS abstractions in detail
- Intel x86
- The PC platform
- The C programming language
- Unix abstractions
- Experience with building system software
  - Handle complexity, concurrency, etc.

Have fun!