NAME - PRADIP KUMAR RAM

DEPT - CSE

SEM - 5th

ROLL NO - 33200118025

TOPIC - THREADS AND SYSTEM CALLS

System Calls

- "Kernel functions" that perform privileged operations on behalf of the process. As an OS designer, one of the goals is to minimize the system call interface.
- UNIX's fork() and exec() versus Windows' CreateProcess()
- pipe() system call
- UNIX process inheritance and file sharing

Case study: Unix/xv6 shell (simplified)

- provides an interactive command execution and programming language
- typically handles login session, runs other processes
- look at some simple examples of shell operations, how they use different OS abstractions, and how those abstractions fit together.

Basic structure:

```
while (1) {
   write (1, "$ ", 2);                    // 1 = STDOUT_FILENO
   readcommand (0, command, args);        // parse user input, 0 = STDIN_FILENO
   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");
   }
}
```

-

- system calls: read, write,   fork, exec, wait. conventions: -1 return value signals error, error code stored in errno, perror prints out a descriptive error message based on errno.
- What's the shell doing? fork, exec, wait: process diagram (PID, address space -- memory of the process, parent links). fork returns twice, in some sense!

The split of process creation into fork and exec turns out to have been an inspired choice, though that might not have been clear at the time

- 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.)
- Example:

```
$ ls
```

- how does ls know which directory to look at?
- how does it know what to do with its output?
- I/O: process has file descriptors, numbered starting from 0.
- system calls: open, read, write, close
- numbering conventions:

file descriptor 0 for input (e.g., keyboard). read_command:

```
read (0, buf, bufsize)
```

file descriptor 1 for output (e.g., terminal)

```
write (1, "hello\n", strlen("hello\n"))
```

  - file descriptor 2 for error (e.g., terminal)
- on fork, child inherits open file descriptors from parent (show in process diagram).
- on exec, process retains file descriptors, except those specifically marked as close-on-exec: fcntl(fd, F_SETFD, FD_CLOEXEC)

How does the shell implement:

```
$ ls > tmp1
```

just before exec insert:

```
close(1);

creat("tmp1", 0666);   // fd will be 1
```

The kernel always uses the first free file descriptor, 1 in this case. Could use dup2() to clone a file descriptor to a new number.

- Good illustration for why fork + exec vs. CreateProcess on Windows. (CreateProcess takes 10 arguments.)

  What if you run the shell itself with redirection?

  ```
      $ sh < script > tmp1
  ```

  If for example the file script contains

  ```
      echo one

      echo two

          dup(1);                  // fd will be 2
  ```

  or in bourne shell syntax,

  ```
     $ ls f1 f2 nonexistant-f3 > tmp1 2>&1
  ```

- Read Chapter 3 of *Advanced Programming in the UNIX Environment* by W. Richard Stevens for a detailed understanding of how file descriptors are implemented. In particular, read Section 3.10 to understand how file sharing works.
- Linux has a nice representation of a process and its FDs, under /proc/PID/
  - maps: VA range, perms (p=private, s=shared), offset, dev, inode, pathname
  - fd: symlinks to files pointed to by each fd. (what's missing in this representation?)
  - can do fd manipulation in shell and see it reflected in /proc/$$/fd

  how to run a series of programs on some data?

  ```
          $ sort < file.txt > tmp1

          $ uniq tmp1 > tmp2

          $ wc tmp2

          $ rm tmp1 tmp2
  ```

  can be more concisely done as:

  ```
        $ sort < file.txt | uniq | wc
  ```

-

- A pipe is a one-way communication channel. Here is a simple example:

  ```
       int fdarray[2];

        char buf[512];
  ```

```
int n;


pipe(fdarray);

write(fdarray[1], "hello", 5);

n = read(fdarray[0], buf, sizeof(buf));

// buf[] now contains 'h', 'e', 'l', 'l', 'o'
```

-

  file descriptors are inherited across fork(), so this also works:

```
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));

}
```

- 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, and waits for the last process of the pipeline to exit. Here's a sketch of what the shell does, in the child process of the fork() we already have, to set up a pipe:

```
    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");
}
```

-
- Who waits for whom? (draw a tree of processes)
- Why close read-end and write-end? ensure that every process starts with 3 file descriptors, and that reading from the pipe returns end of file after the first command exits.

  How do you create a background job?

  ```
  $ compute &
  ```

-
- How does the shell implement "&", backgrounding? (Don't call wait immediately).

Threads

Figure on process address space: code, static data, stack, heap. On fork, the whole address space gets replicated. On thread create, the created thread has a different program counter, registers, and stack (through stack pointer). Everything else is shared between threads.

Kernel-level threads are just processes minus separate address spaces. Discuss the kernel scheduler which is invoked at every timer interrupt. Each thread is an independent entity for the kernel.

Write the cswitch function for processes and threads. Notice that switching among threads requires no privileged operations. Switching the stack can be done by switching the sp register. A cswitch needs to be fast (typically a few 100 microseconds).

Advantages of threads over processes

- *Much* more lightweight than processes. Faster creation, deletion, switching.
- Much faster communication among threads: allow shared data structures to be maintained.

User-level threads can be implemented inside a process by writing scheduler() and cswitch() functions. The scheduler can be called periodically using SIGALRM signal.

Pros of user-level threads:

- Lighter-weight
- Faster cswitch
- Do not need kernel's permission

Cons of user-level threads:

- Will not be scheduled on different cores, because they look like one process to the kernel.
- If one thread blocks (e.g., on I/O), all threads block.

Threading models (slide)

Thread pools (slide)