Comp 204: Computer Systems and Their Implementation

Lecture 3: Processes (2)

Today

- Interrputs
- Process execution scenarios
- UNIX processes

System Calls

- Request action of OS
- Set of system calls acts as Application Programmer Interface (API)
  - e.g. Win32 API
- System calls can be made from languages such as C
  - Note Java does not support system calls directly
- Form of software interrupt

Interrupt-Driven Execution

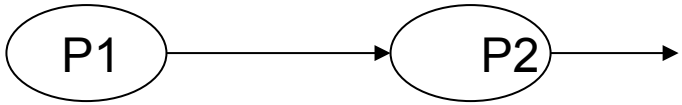
- A modern OS is interrupt-driven
- If there are no events to service, OS remains idle
  - e.g. Windows idle process
- An event may be
  - Peripheral device interrupt
  - Error condition (e.g. divide by zero)
  - System call
  - Timer elapse
- An interrupt causes transfer to an Interrupt Service Routine
  - Address is contained in interrupt vector

Process Control - UNIX

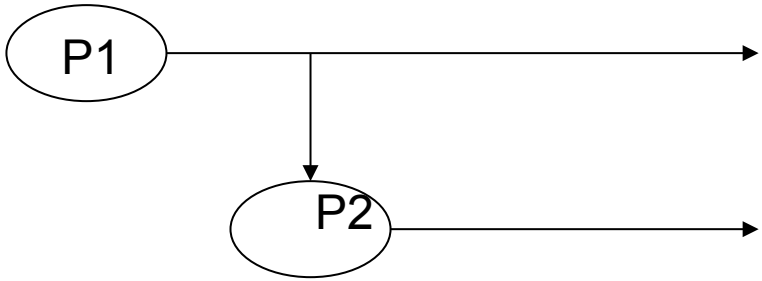
- The exec() system call allows one program to execute another
  - stay in same process
  - new program overwrites caller
- The fork() call creates (spawns) a new process
  - caller is parent, new process is child
  - child is identical to parent
- The wait() call suspends parent until child terminates

Execution Scenarios

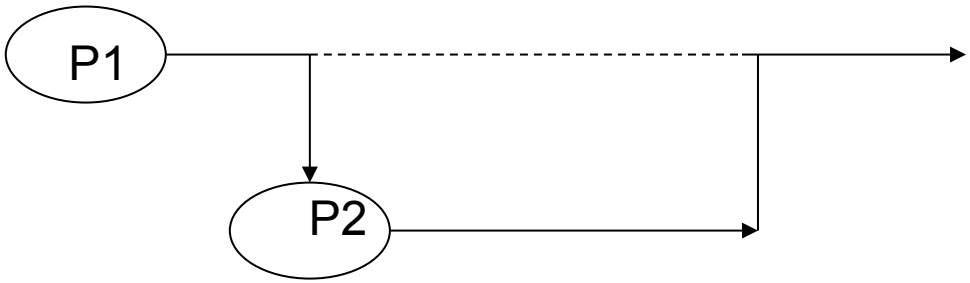
1. P1 execs P2



2. P1 forks, then child execs P2



3. As 2, but now P1 issues wait() call



Question

- If you type ‘cat prog.c’ at a UNIX command prompt, which of the following sequences of system calls would be invoked?

a) The shell makes an exec() call
b) The shell calls fork(); the child process calls exec() and the parent calls wait()
c) The shell calls fork(); the child calls wait() and the parent calls exec()
d) The shell calls exec() and then wait() and then fork()
e) The shell calls wait() then fork(), creating a child which calls exec()

Answer: b
The shell calls fork(); the child process calls exec() and the parent calls wait(). The shell is just another process that can take a string as standard input, look for the program referenced by the string, and then run this program. Unless the program is put in the background, the shell will wait until the program has finished.

Parents and Children

- The value returned by fork() can be:
  < 0 indicates error
  == 0 This is the child
  > 0 This is the parent; value is pid of child

- So, code often looks something like:

```
int pid;
pid = fork();
if (pid < 0)
   /* error */
else if (pid == 0)  /* child */
   exec(a program);
else /* parent */
   perhaps wait for child;
```

Exercise

- What will the following print?

```
main()
{
 printf("Hello ");
 fork();
 printf("world");
}
```

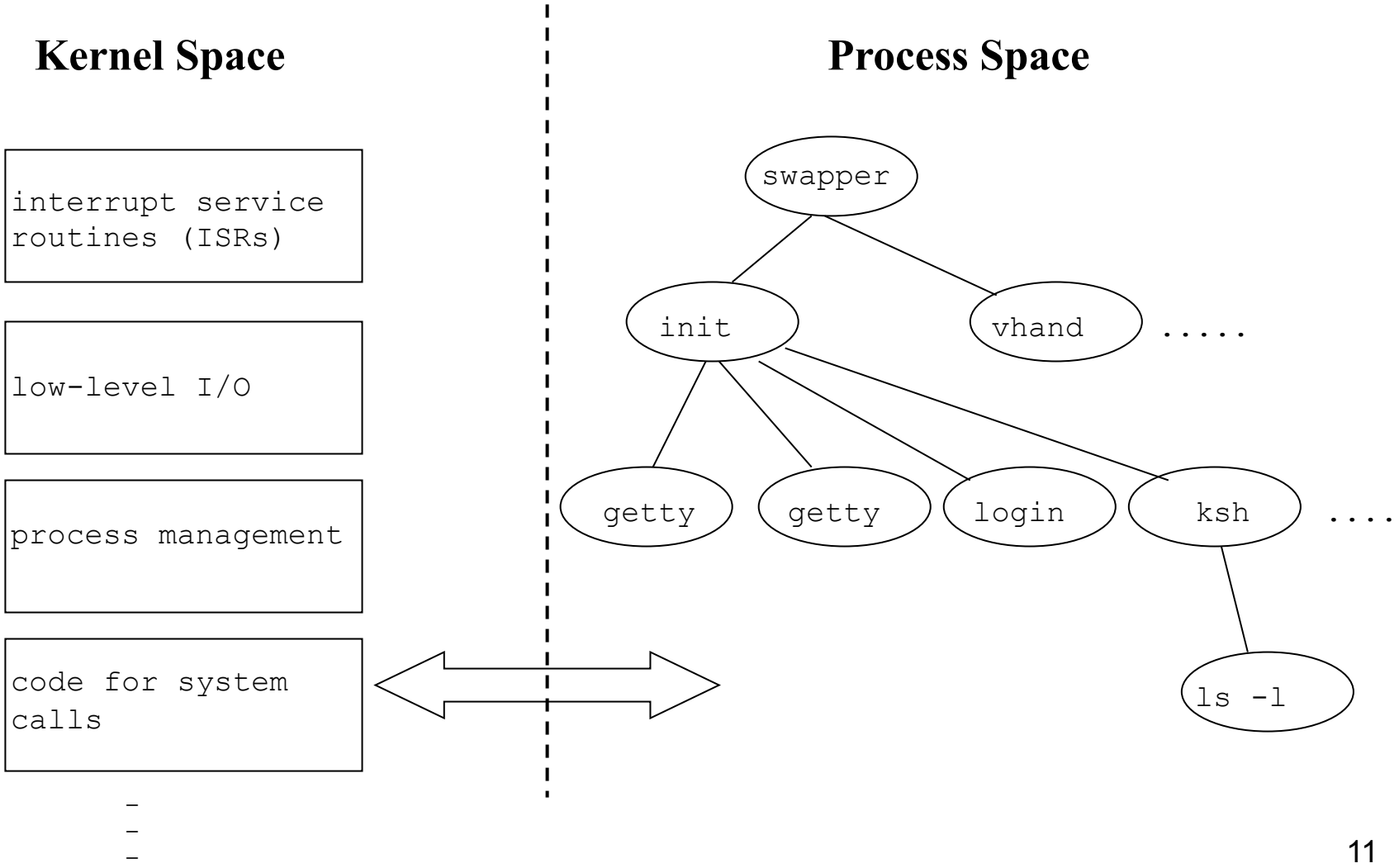
- What is wrong with the following?

```
if (pid == 0)
{
 exec(a program);
 printf("Finished");
}
```

System Initialisation

- On power-up or reset, system executes bootstrap program contained in ROM
  - Initialises machine, runs diagnostic checks, clears memory etc.
  - Loads kernel of OS and hands over control
- On some systems, a more sophisticated boot program can be loaded from disk (boot block)
  - Allows it to be changed on disk

UNIX Structure



UNIX Processes

- At startup, kernel creates process 0
  - kernel-level process, so can access kernel code and data structures directly
  - eventually becomes swapper, responsible for moving processes to/from disk
  - before this, it spawns other kernel-level processes
    - e.g. vhand, for reclaiming pages
- Process 1 is ‘init’
  - runs some initialisation programs, then forks a copy of itself on each terminal
  - each child execs the ‘getty’ program

UNIX Processes

- getty prints ‘login’ message, waits for username, then execs /bin/login
- login asks for password, checks these against details in /etc/passwd, then execs shell
- shell forks and execs commands specified by user
- at logout, shell process dies
  - init is informed, and creates a new getty

Zombies and Orphans

- Parent processes usually wait for their children to die (!)
- If the death of a child is not acknowledged by the parent (via a wait call), the child becomes a ‘zombie’
  - No resources, but still present in process table
- If the parent dies before its children, the children become ‘orphans’
  - Get ‘adopted’ by Init process

Daemons

- Not associated with any user or terminal
- Run permanently in the background
- Perform tasks requested of them by other processes
- Examples
  - printer spooler/scheduler
  - various servers