



Process

Lawrence Angrave
Lecture 6

Content of This Lecture

Goals:

Start the basic system concept

Things covered in this lecture

Process

Administrative

This week

SMP1 due Monday at 9am.

Lecture quiz sometime this week on course content

Compass self Assessment Quiz 2

MP Quiz

SMP1 - experiment with shells

Implement a shell using *fork* and *exec*, *waitpid*

Use it as a workbench to experiment with
processes and process control

background processes

environment path

signals and process control

sub-shells & recursion

reasons for built-in shell commands

Example questions...

Can a shell kill itself? Can a shell within a shell kill the parent shell?

What happens to background processes when you exit from the shell?

What would happen if this program did not use the *fork* function, but just used *execv* directly?

And now the slides....

Users, Programs, Processes

Users have accounts on the system

Users launch programs

- Many users may launch same program

- One user may launch many instances of the same program

Processes:

- an executing program

Analogy

Program: steps for attending the lecture

Step1: walk to Siebel Center Building

Step2: enter 1404 Lecture Room

Step3: find a seat

Step4: listen and take notes

Process: attending the lecture

Action

You are all in the middle of a process

Processes

A **process** is an abstraction for sequence of operations that implement a computation/program. A process may be manipulated, suspended, scheduled and terminated

Process is a unit of work in a modern computer

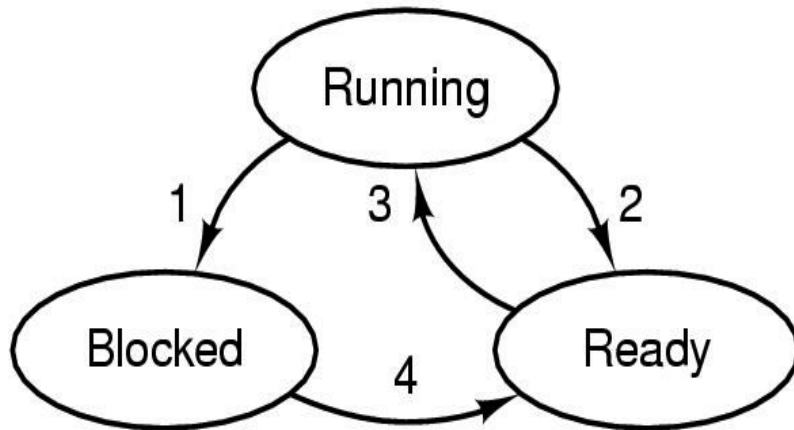
Process Types:

- OS processes executing system code program

- User processes executing user code program

Processes are executed concurrently with CPU multiplexing among them

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Possible process states

Running (occupy CPU)

Blocked

Ready (does not occupy CPU)

Other states: suspended, terminated

Transitions between states

Question: in a single processor machine, how many process can be in running state?

1 CPU can run Multiple Processes

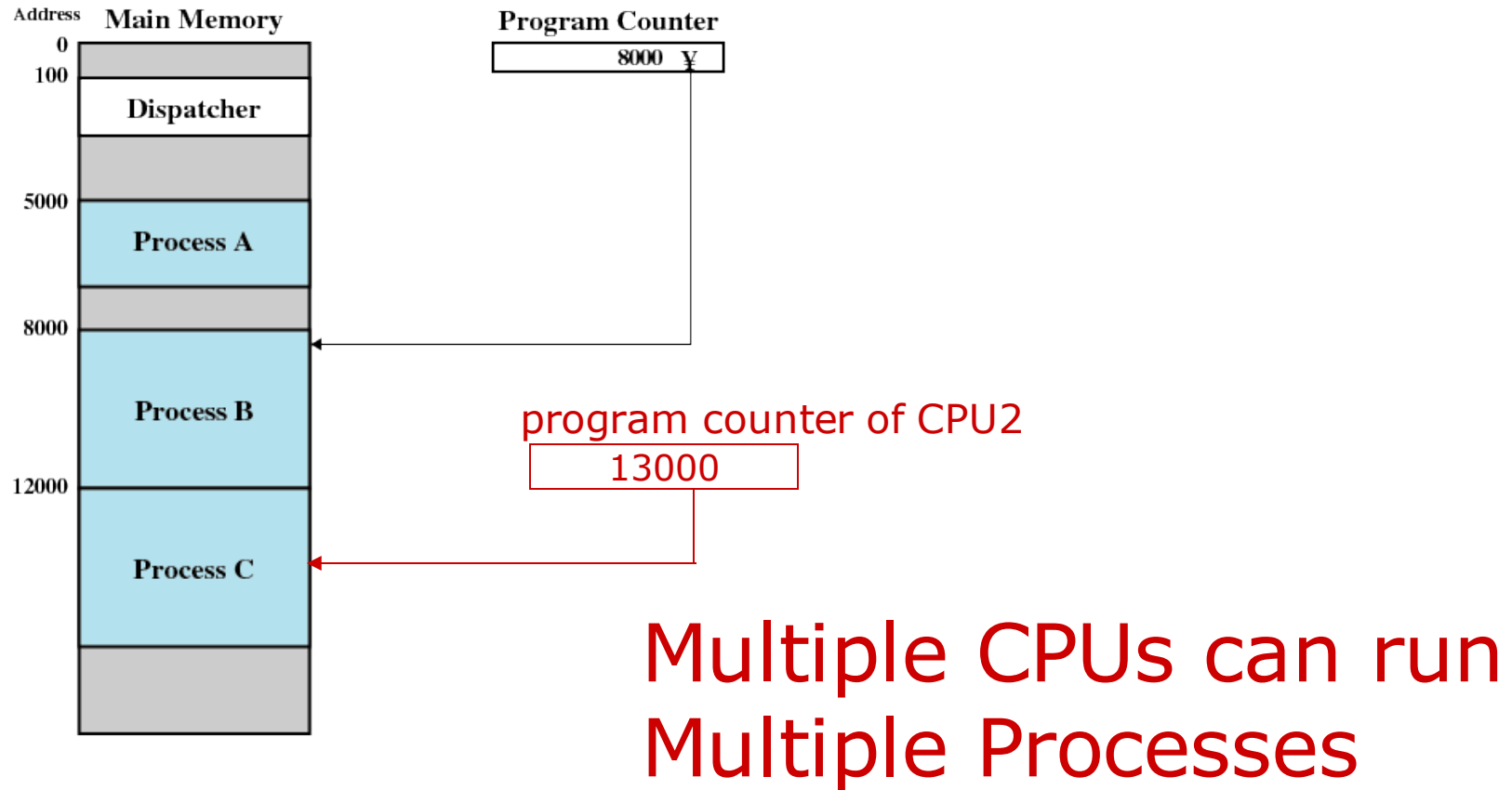


Figure 3.2 Snapshot of Example Execution (Figure 3.4)
at Instruction Cycle 13

Questions

Using “working on a MP” as an example

What corresponds to “running”?

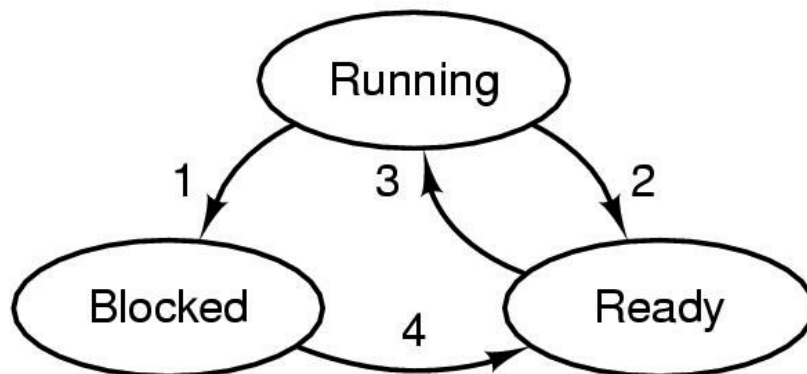
What corresponds to “blocked”?

What corresponds to “ready”?

Why not the following transitions?

Ready to blocked

Blocked to running



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Linux 5 State Process Model (S:118)

Add states for creating and deleting process

Add transitions Timeout, Dispatch, Event Occurs

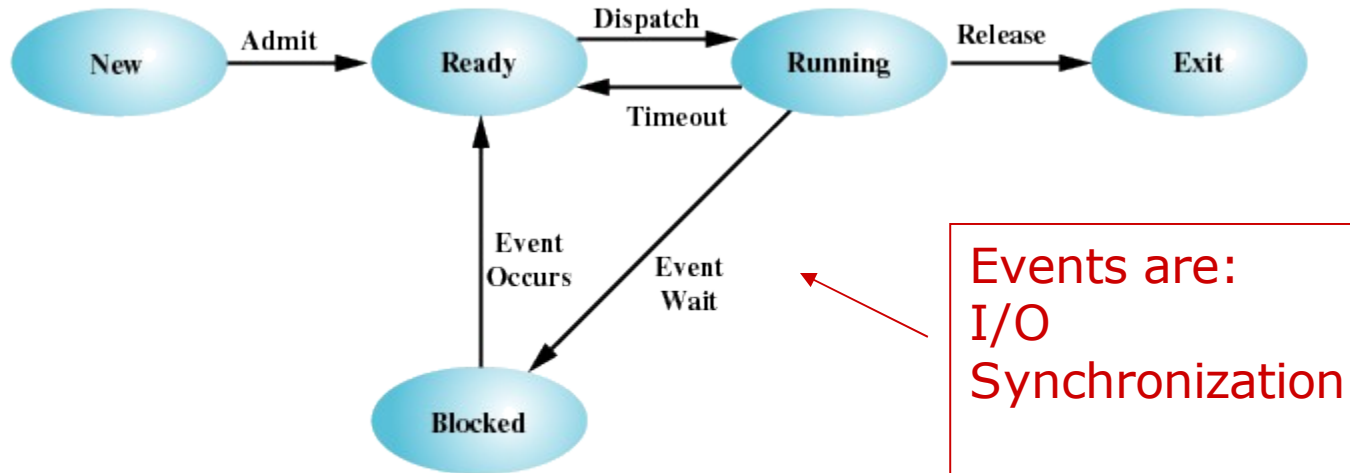


Figure 3.6 Five-State Process Model

Windows Task Manager

Windows Task Manager

File Options View Help

Applications Processes Performance Networking

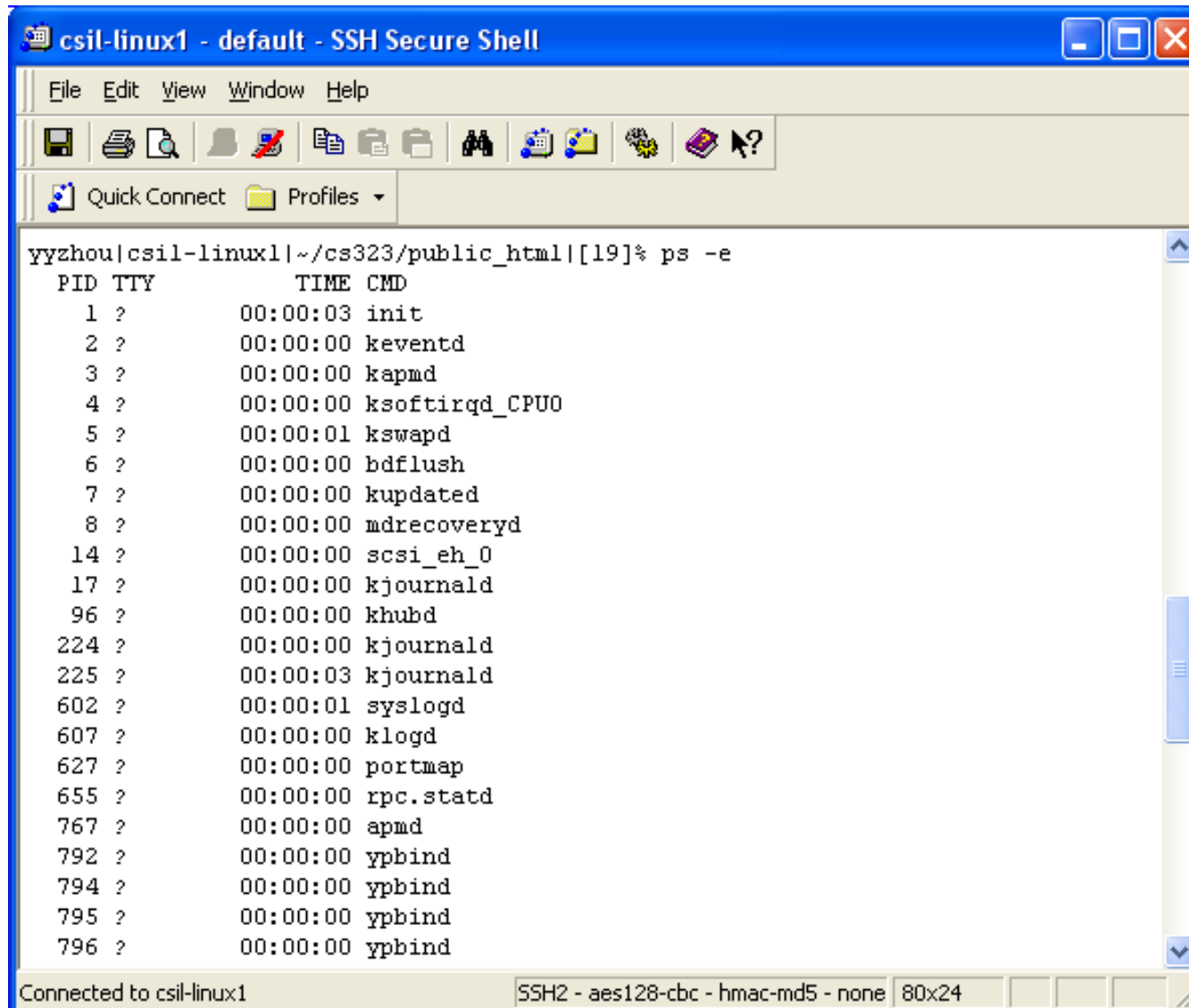
Image Name	User Name	C...	Mem U...
HelpHost.exe	Yuanyuan Zhou	00	6,456 K
AGENTSVR.EXE	Yuanyuan Zhou	00	240 K
POWERPNT.EXE	Yuanyuan Zhou	00	4,040 K
HelpSvc.exe	SYSTEM	00	11,528 K
IEXPLORE.EXE	Yuanyuan Zhou	00	18,944 K
MSTSC.EXE	Yuanyuan Zhou	00	2,604 K
TASKMGR.EXE	Yuanyuan Zhou	00	1,800 K
hposts07.exe	Yuanyuan Zhou	00	4,948 K
hpofxm07.exe	Yuanyuan Zhou	00	5,080 K
CMD.EXE	Yuanyuan Zhou	00	700 K
SVCHOST.EXE	SYSTEM	00	2,728 K
hpoevm07.exe	Yuanyuan Zhou	00	3,664 K
jconfigdnt.exe	SYSTEM	00	1,604 K
inetd32.exe	SYSTEM	00	2,344 K
ati2evxx.exe	SYSTEM	00	1,672 K
WZQKPICK.EXE	Yuanyuan Zhou	00	2,348 K
SPOOLSV.EXE	SYSTEM	00	4,632 K
hpgs2wnf.exe	Yuanyuan Zhou	00	3,248 K
SVCHOST.EXE	LOCAL SERVICE	00	3,576 K
SVCHOST.EXE	NETWORK SERVICE	00	2,260 K

Show processes from all users

End Process

Processes: 39 CPU Usage: 1% Commit Charge: 173M

Unix Example: ps



```
csil-linux1 | ~/cs323/public_html | [19]% ps -e
PID TTY          TIME CMD
  1 ?             00:00:03 init
  2 ?             00:00:00 keventd
  3 ?             00:00:00 kapmd
  4 ?             00:00:00 ksoftirqd_CPU0
  5 ?             00:00:01 kswapd
  6 ?             00:00:00 bdflush
  7 ?             00:00:00 kupdated
  8 ?             00:00:00 mdrecoveryd
 14 ?             00:00:00 scsi_eh_0
 17 ?             00:00:00 kjournald
 96 ?             00:00:00 khubd
224 ?             00:00:00 kjournald
225 ?             00:00:03 kjournald
602 ?             00:00:01 syslogd
607 ?             00:00:00 klogd
627 ?             00:00:00 portmap
655 ?             00:00:00 rpc.statd
767 ?             00:00:00 apmd
792 ?             00:00:00 ypbind
794 ?             00:00:00 ypbind
795 ?             00:00:00 ypbind
796 ?             00:00:00 ypbind
```

Connected to csil-linux1 | SSH2 - aes128-cbc - hmac-md5 - none | 80x24

Process Identification (RR: pp 60)

UNIX identifies processes via unique value

Process ID

Each process has also parent process ID since each process is created from a parent process.

Root process is the 'init' process

'*getpid*' and '*getppid*' – functions to return process ID (PID) and parent process ID (PPID)

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
int main (void) {
```

```
    printf("I am process %ld\n", (long)getpid());
```

```
    printf("My parent id %ld\n", (long)getppid());
```

```
    return 0;
```

```
}
```

Creating a Process - Fork

Creating a process and executing a program are two different things in UNIX

Fork duplicates a process so that instead of one process you get two--- But the code being executed doesn't change!!!

Fork returns

- 0 if child

- 1 if fork fails

- Child's PID if parent process

Child gets new program counter, stack, file descriptors, heap, globals, pid!

Creating a Process in Unix

```
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int x;

    x = 0;
    fork();
    x = 1;
    printf("I am process %ld and my x is %d\n", (long)getpid(), x);
    return 0;
}
```

What does this print?

UNIX Example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
    pid_t parentpid;
    pid_t childpid;

    if ((childpid = fork()) == -1) {
        perror("can't create a new process");
        exit(1);
    }
    else if (childpid == 0) { /* child process executes */
        printf("child: childpid = %d, parentpid = %d \n", getpid(),
getppid());
        exit(0);
    }
    else { /*parent process executes */
        printf("parent: childpid = %d, parentpid = %d \n", childpid,
getpid());
        exit(0);
    }
}
```

UNIX Example

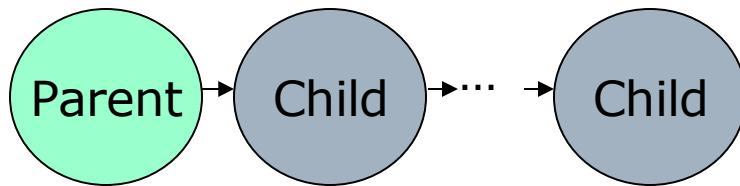
```
childpid = fork()
```

```
if (childpid == 0) {  
    printf("child: childpid = %d, parentpid = %d \n",  
          getpid(), getppid());  
    exit(0);  
  
} else {  
    printf("parent: childpid = %d, parentpid = %d \n",  
          childpid, getpid());  
    exit(0);  
}
```

Chain and Fan (RR P69)

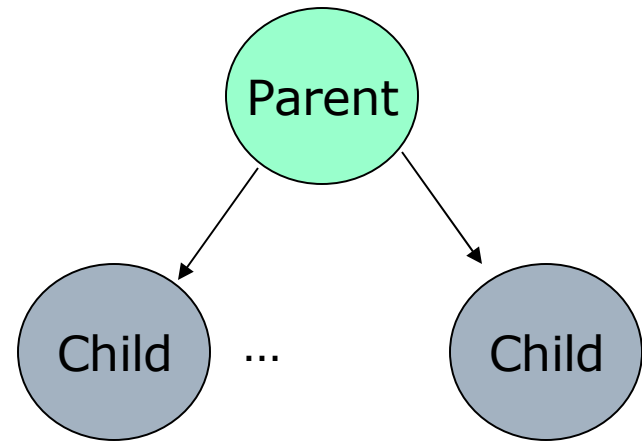
Chain

```
pid_t childpid = 0;  
for (i=1;i<n;i++)  
  if (childpid = fork())  
    break;
```



Fan

```
pid_t childpid = 0;  
for (i=1;i<n;i++)  
  if ((childpid = fork()) <= 0)  
    break;
```



Process Operations (Creation)

When creating a process, we need resources such as CPU, memory files, I/O devices

Process can get resources from the OS or from the parent process

Child process is restricted to a subset of parent resources

Prevents many processes from overloading system

Execution possibilities are

Parent continues concurrently with child

Parent waits until child has terminated

Address space possibilities are:

Child process is duplicate of parent process

Child process has a new program loaded into it

Process Termination

Normal exit (voluntary)

End of main()

Error exit (voluntary)

exit(2)

Fatal error (involuntary)

Divide by 0, core dump / seg fault

Killed by another process (involuntary)

Kill proclD, end task

Process Operations (Termination)

When a process finishes last statement, it automatically asks OS to delete it

Child process may return output to parent process, and all child's resources are de-allocated.

Other termination possibilities

Abort by parent process invoked

Child has exceeded its usage of some resources

Task assigned to child is no longer required

Parent is exiting and OS does not allow child to continue without parent

Process Hierarchies

Parent creates a child process, a child process can create its own processes

Forms a hierarchy

UNIX calls this a "process group"

Windows has no concept of process hierarchy

all processes are created equal

wait() Function

wait function allows parent process to wait (block) until child finishes

wait function causes the caller to suspend execution until child's status is available

waitpid function allows a parent to wait for a particular child

errno	cause
ECHILD	Caller has no unwaited-for children
EINTR	Function was interrupted by signal
EINVAL	Options parameter of <i>waitpid</i> was invalid

Waiting for a child to finish – C Manual

```
#include <errno.h>
#include <sys/wait.h>
```

```
pid_t childpid;
```

```
childpid = wait(NULL);
```

```
if (childpid != -1)
```

```
    printf("waited for child with pid %ld\n", childpid);
```

Summary

Read R&R : pp: 60-83, pp. 812-814

Read S: Chapter 3:108-118, 153-156

SMP1

Week 2 Self Assessment Quizzes