# Operating Systems

Lecture 2

# The Process Concept

# The Process Concept

Program

- description of how to perform an activity
- consists of instructions and static data values

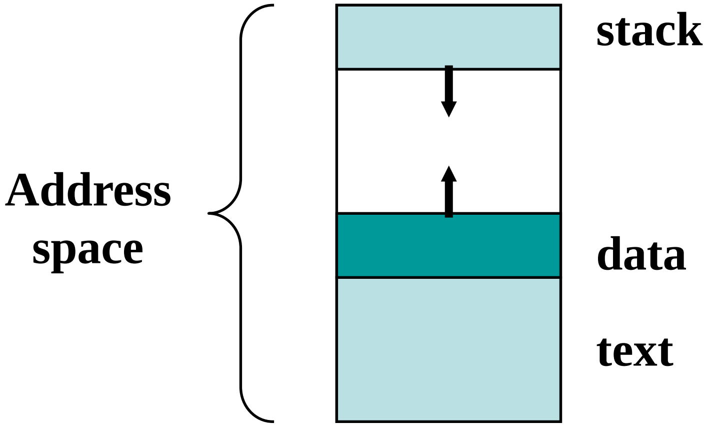
Process

- a snapshot (status) of a program in execution
- memory (program instructions, static and dynamic data values)
- CPU state (registers, PC, SP, etc)
- operating system state (open files, sockets etc)

# Process Address Space

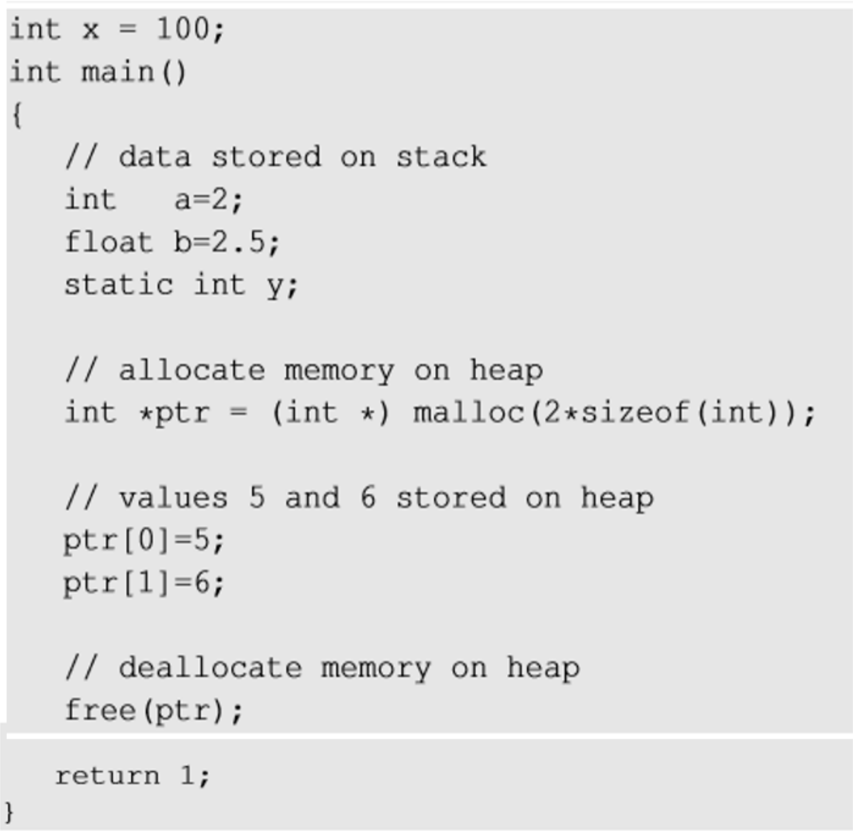
Each process runs in its own virtual memory *address space* that consists of:

- *Stack space* – used for function and system calls
- *Data space* – variables (both static and dynamic allocation)
- *Text* – the program code (usually read only)

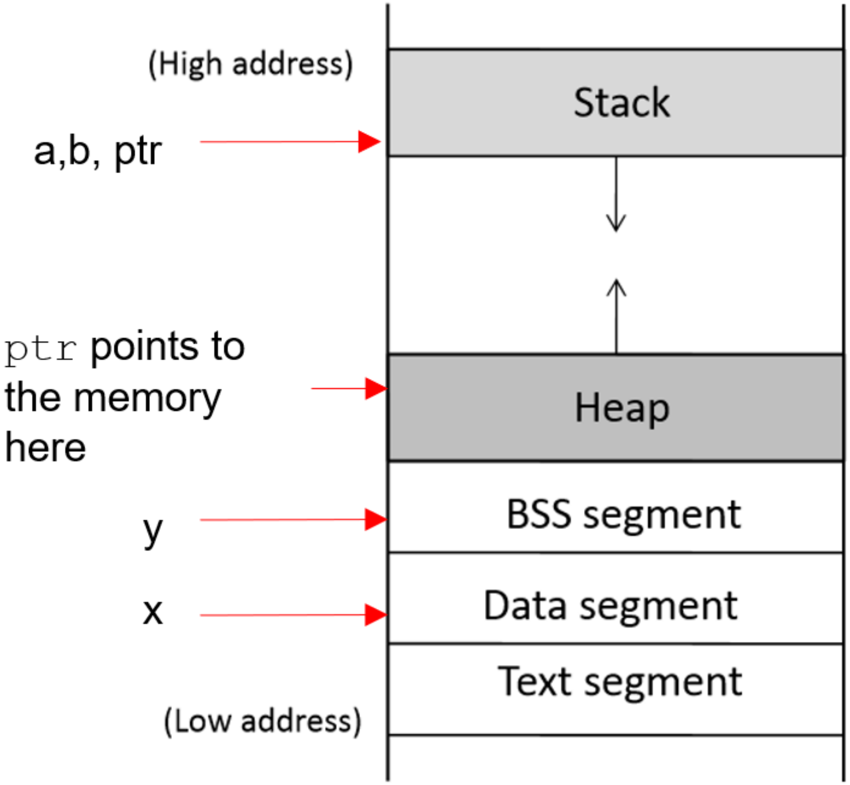


Invoking the same program multiple times results in the creation of multiple distinct address spaces

# Process memory layout

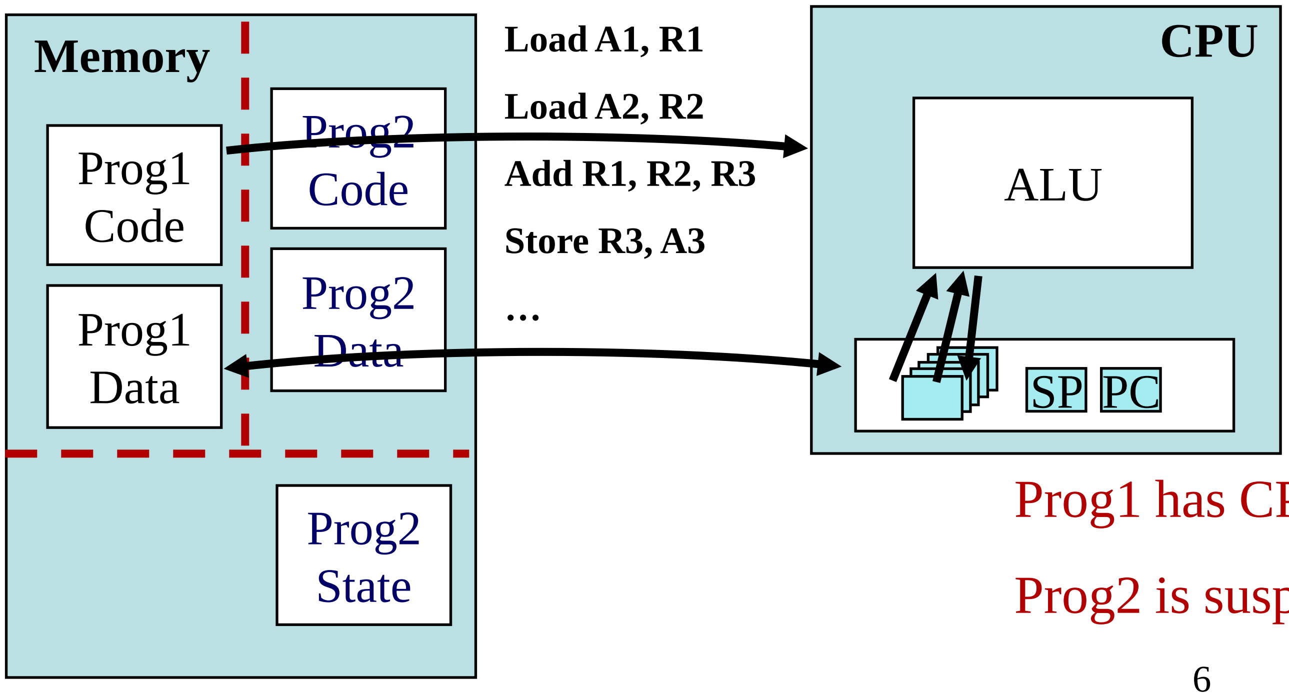


```
int x = 100;
int main()
{
    // data stored on stack
    int   a=2;
    float b=2.5;
    static int y;

    // allocate memory on heap
    int *ptr = (int *) malloc(2*sizeof(int));

    // values 5 and 6 stored on heap
    ptr[0]=5;
    ptr[1]=6;

    // deallocate memory on heap
    free(ptr);

   return 1;
}
```



# Switching Among Processes

Program instructions operate on operands in memory and (temporarily) in registers

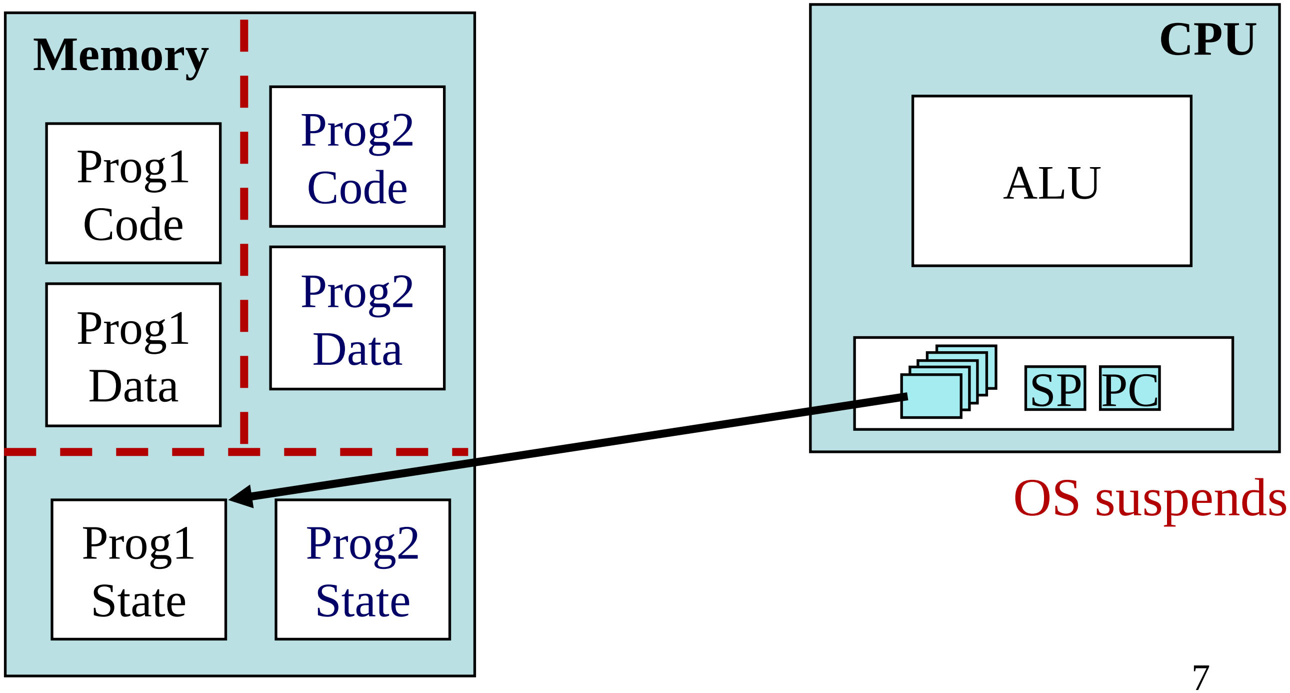


Prog1 has CPU

Prog2 is suspended

# Switching Among Processes

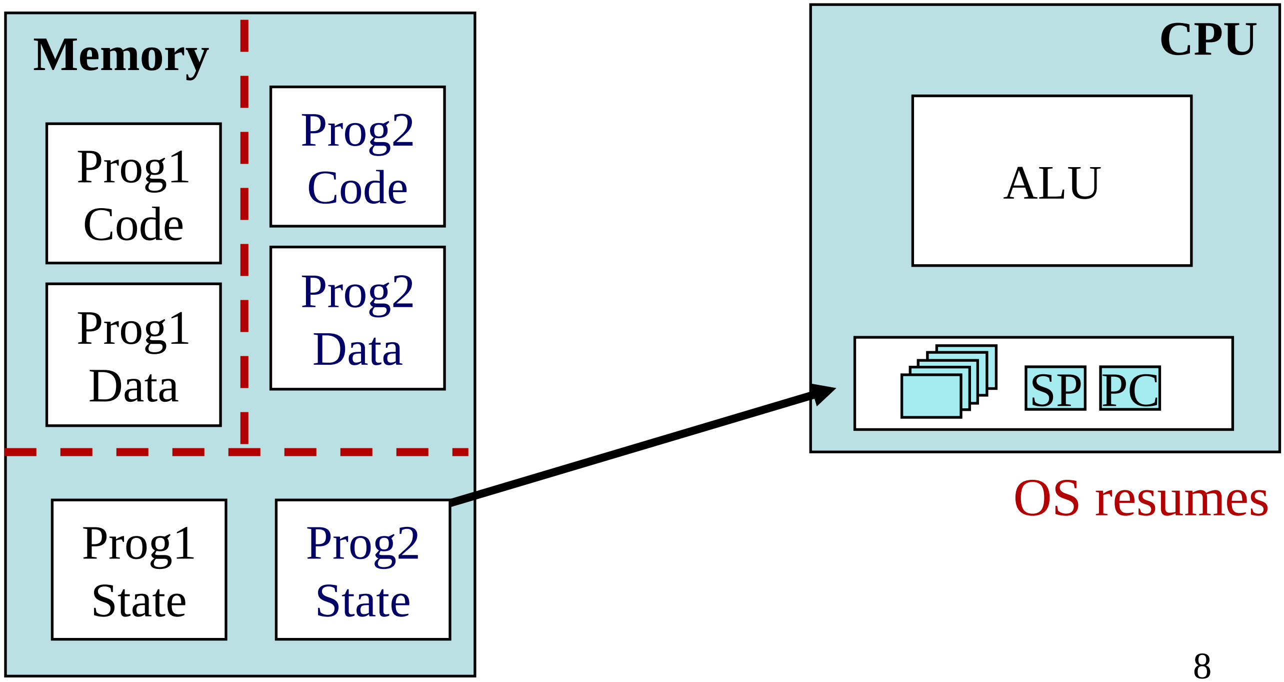
Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*



OS suspends Prog1

# Switching Among Processes

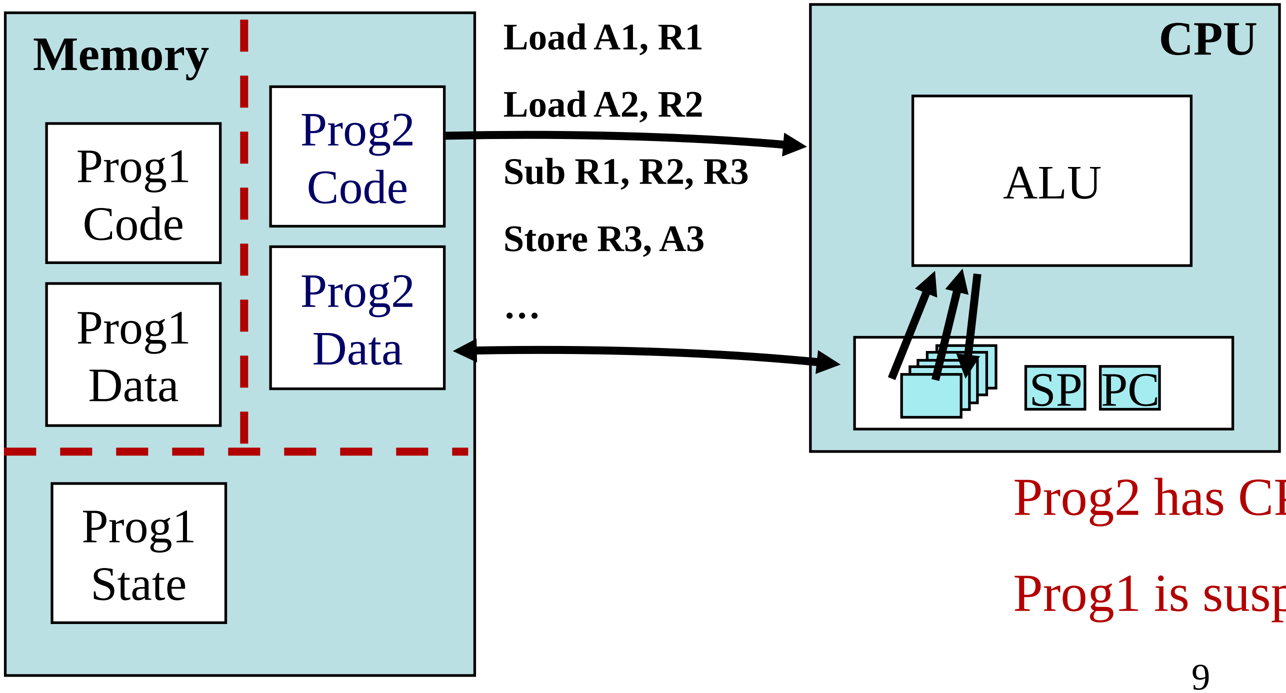
Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*



OS resumes Prog2

# Switching Among Processes

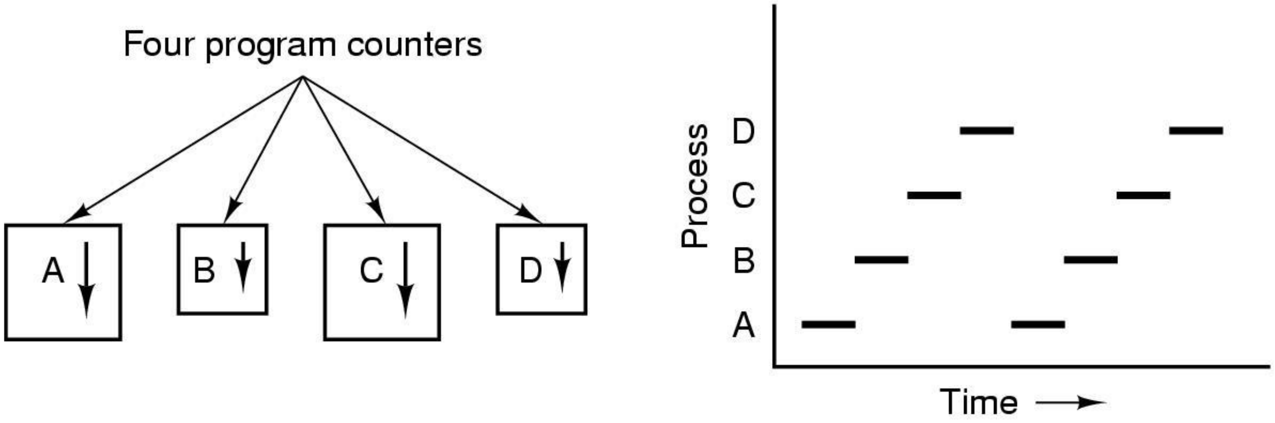
Program instructions operate on operands in memory and in registers



Prog2 has CPU

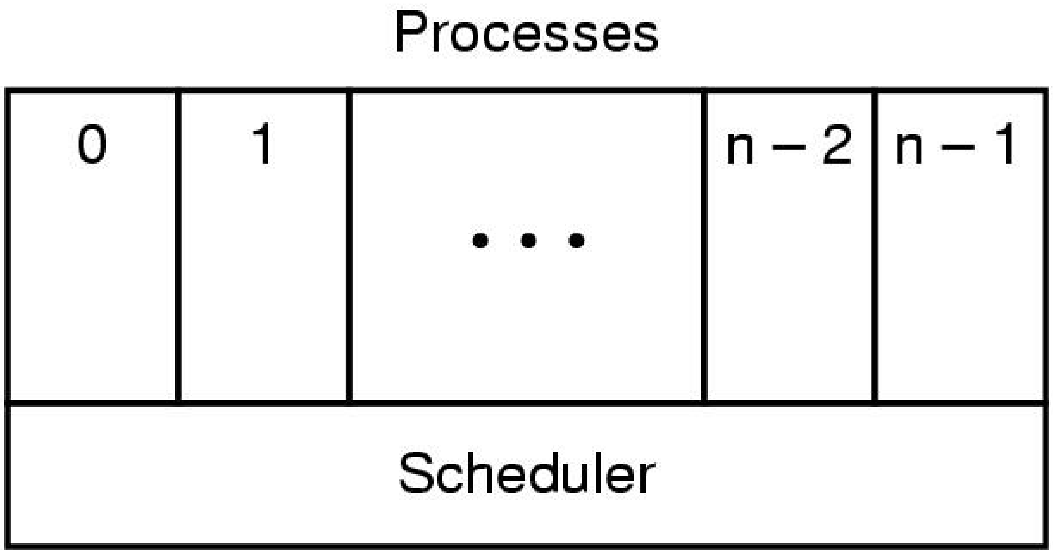
Prog1 is suspended

# The Process Abstraction



Conceptual model of 4 independent, sequential processes
Only one program active at any instant

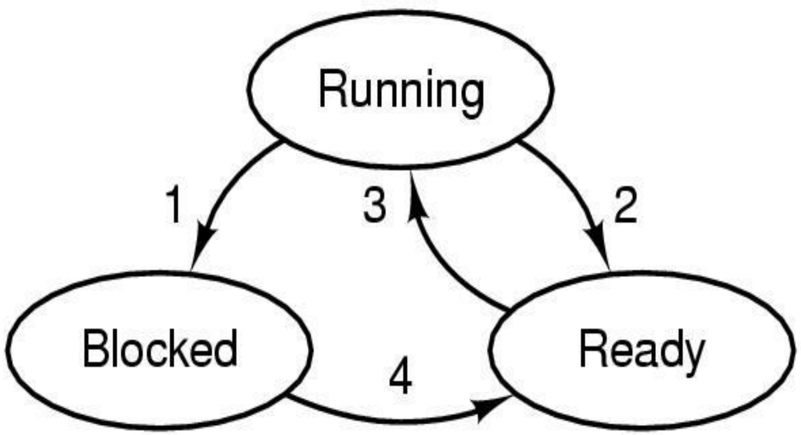
# The Scheduler



Lowest layer of process-structured OS

- handles scheduling of processes

# Process States



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

# How Are Processes Created?

Events that cause process creation:

- System startup
- User request to create a new process
- Execution of a process creation system call from another process

# Process Hierarchies

Parent process creates child process

- each process is assigned a unique identifying number or process ID (PID)
- system calls for communicating with and waiting for child processes

Child processes can create their own child processes

- UNIX calls this hierarchy a "process group‟

# How Do Processes Terminate?

Conditions that terminate processes:

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)

# Process Creation in UNIX

All processes have a unique process id

- ❖ *getpid(), getppid()* system calls allow processes to get their information

Process creation

- ❖ *fork()* system call creates a copy of a process and returns in both processes (parent and child), but with a different return value
- ❖ *exec()* replaces an address space with a new program

Process termination, signaling

- ❖ *signal(), kill()* system calls allow a process to be terminated or have specific signals sent to it

# Process Creation (fork)

Fork creates a new process by *copying* the calling process

The new process has its own

- Memory address space (copied from parent)
  - Instructions (same program as parent!)
  - Data
  - Stack
- Register set (copied from parent)
- Process table entry in the OS

# Process Creation Example

csh (parent)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

# Process Creation Example

csh (parent)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (child)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

# Process Creation Example

csh (parent)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (child)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

# Process Creation Example

csh (parent)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

csh (child)

```
…

pid = fork()
if (pid == 0) {
   // child…
   …
   exec();
   }
else {
   // parent
   wait();
   }
…
```

# Thanks