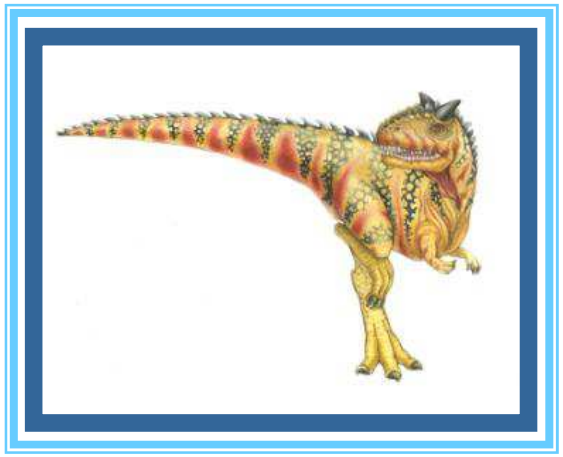
Topic 2: Processes



Silberschatz, Galvin and Gagne ©2015

Rudowsky ©2005

Walpole ©2010

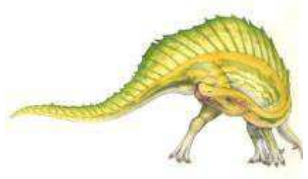
Kubiatowicz ©2010

Stallings ©2015

- Chapter 3: Process Concept
- Chapter 4: Multithreaded Programming
- Chapter 5: Process Scheduling

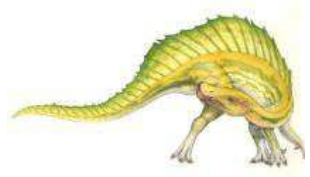
Contents

- **Processes**
  - Process concept
  - Process states
  - Process Control Block (PCB)
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**



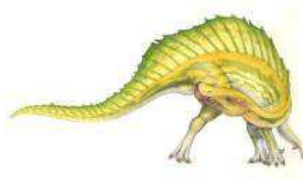
Objectives

- To introduce the notion of a process -- a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms



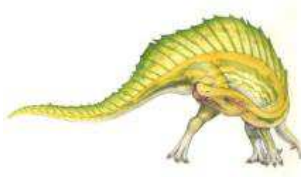
Contents

- **Process**
  - Process concept
  - Process states
  - Process Control Block (PCB)
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**



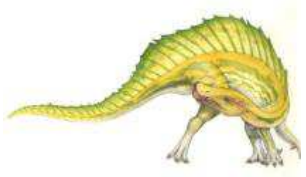
Process Concept

- An operating system executes a variety of programs
- Program:
  - description of how to perform an activity (algorithm)
  - instructions and static data values
  - static file (image)
- Process:
  - a program in execution; process execution must progress in sequential fashion
  - a snapshot of a program in execution
  - an instance of a program running in a computer



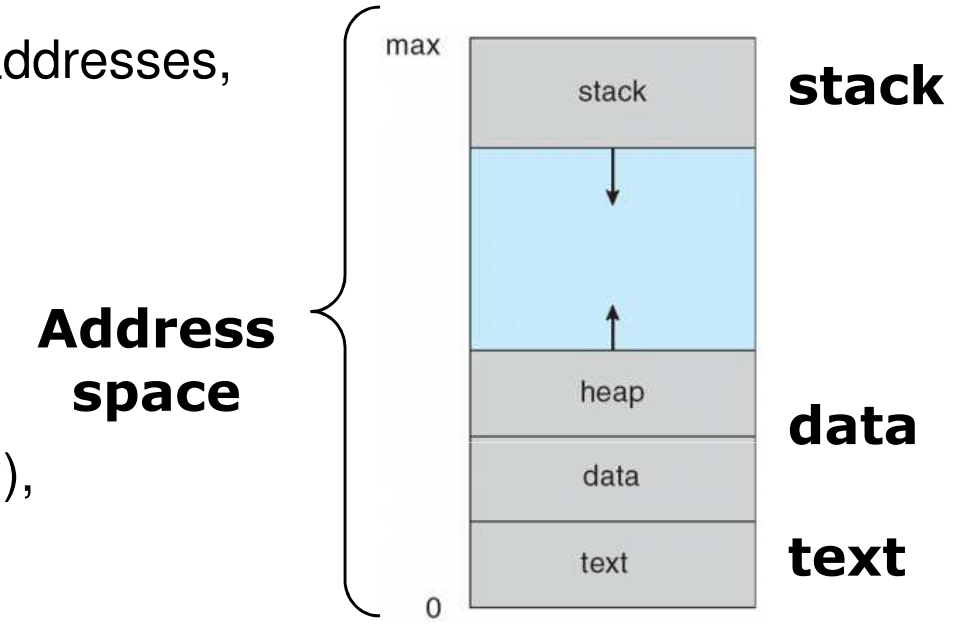
Process Concept

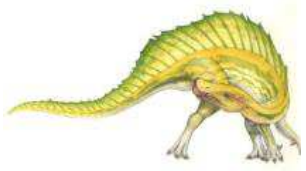
- A process is the basic unit of execution in an operating system
  - Each process has a number, its process identifier (pid)
- Different processes may run different instances of the same program



Process requirements

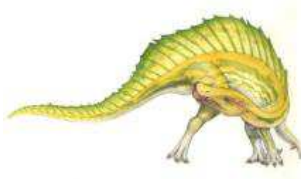
- At a minimum, process execution requires following resources:
  - Memory to contain the program code (**text section**) and data
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - A set of CPU registers to support execution
    - CPU state (registers, Program Counter (PC), Stack Pointer (SP), etc)
    - operating system state (open files, accounting statistics etc)





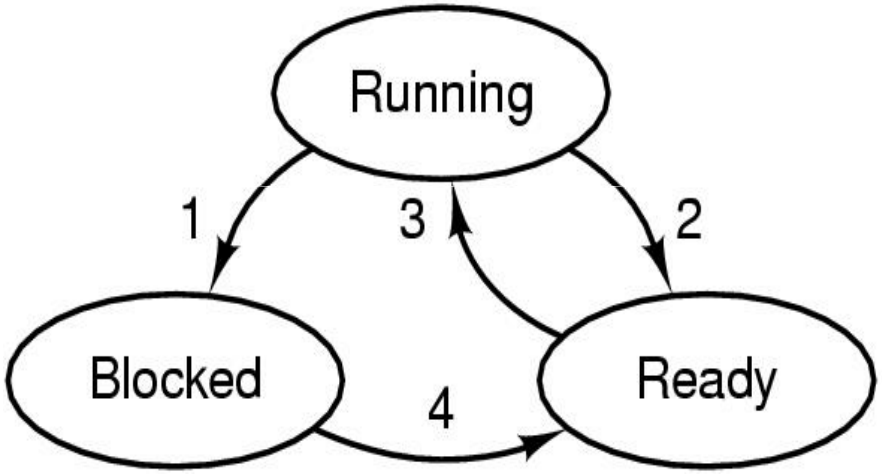
Contents

- **Process**
  - Process concept
  - **Process states**
  - Process Control Block (PCB)
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**

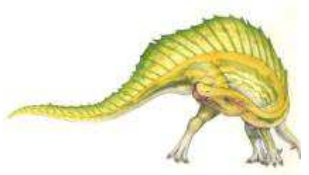


Process states

- As a process executes, it changes **state**
- Possible process states
  - **running**: instructions are being executed
  - **blocked (waiting)**: the process is waiting for some event to occur
  - **ready**: the process is waiting to be assigned to a processor

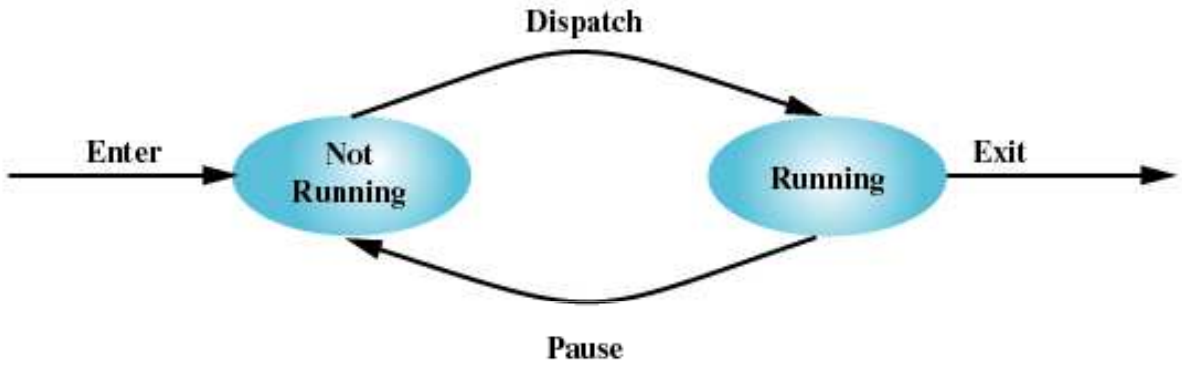


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

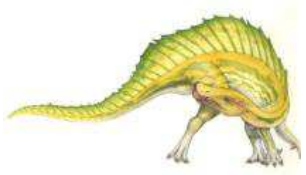


Two-State Process Model

- Process may be in one of two states
  - Running
  - Not-running

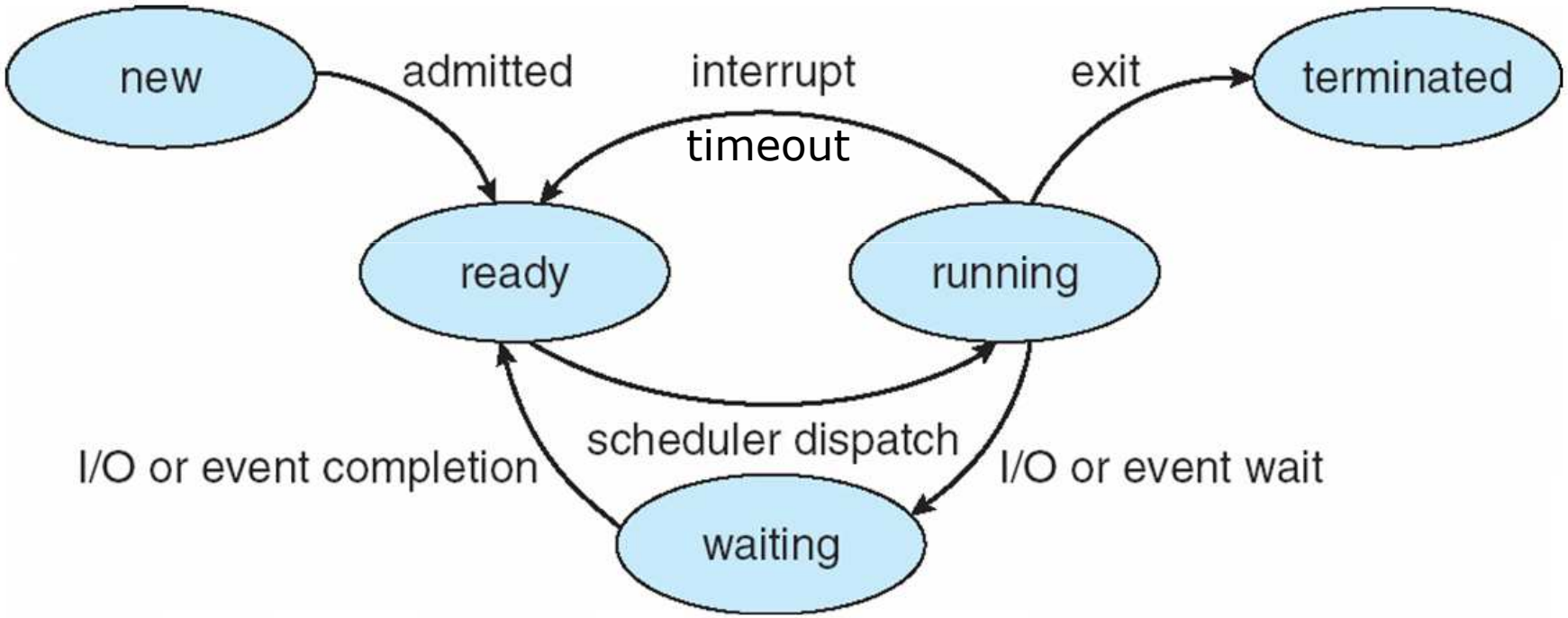


(a) State transition diagram



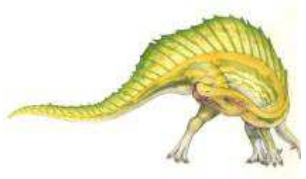
Five-State Process Model

- States:
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting (blocked)**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution



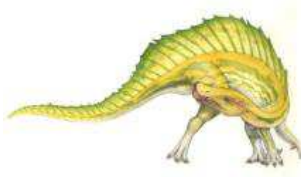
Swapped and Suspended Processes

- Processor is faster than I/O so all processes could be waiting for I/O
  - Swap these processes to disk to free up more memory
    - More processes can be executed
- System overloaded
  - blocked state becomes suspend state when swapped to disk
- Two new states
  - Blocked/Suspend
  - Ready/Suspend



Reasons for Process Suspension

- Swapping: OS needs to release memory to bring in a ready process
- Protection: OS may suspend a process suspected of causing a problem
- User request: may whish to suspend execution for debugging reasons
- Timing: process executed periodically, and suspended while waiting the next round
- Others

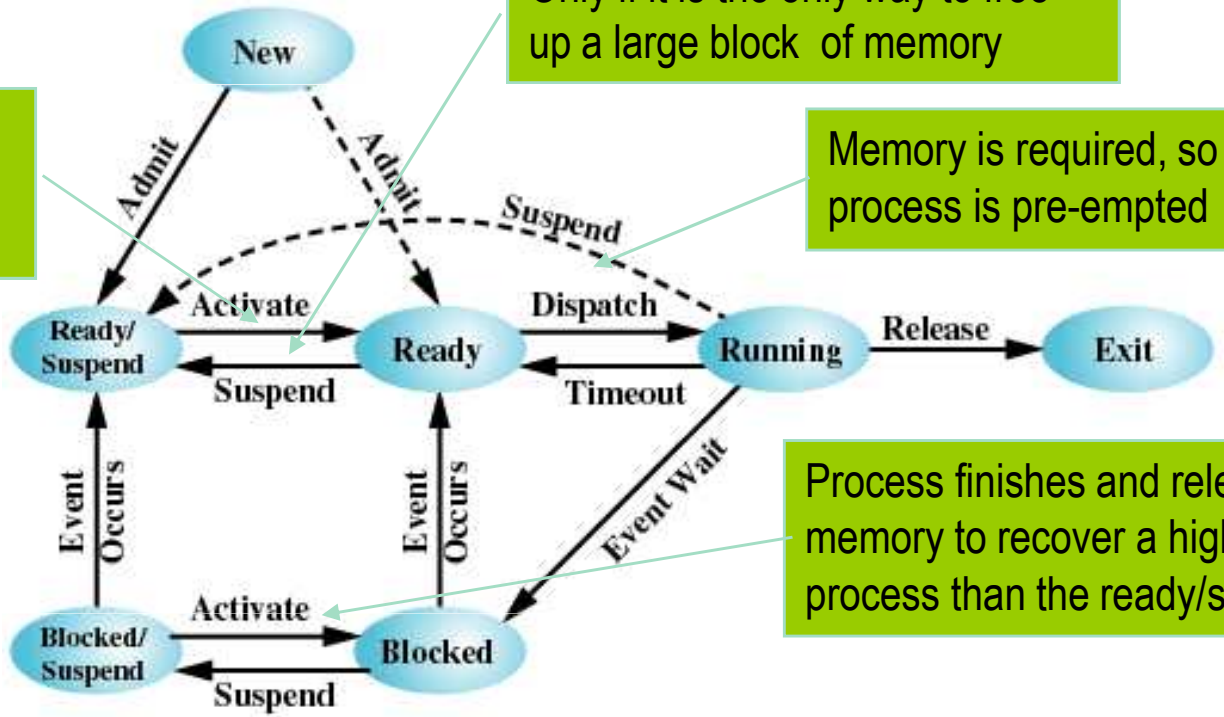


Suspended Processes

Only if it is the only way to free up a large block of memory

No ready processes or ready processes with less priority

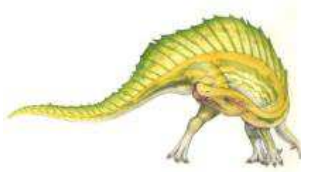
Memory is required, so process is pre-empted



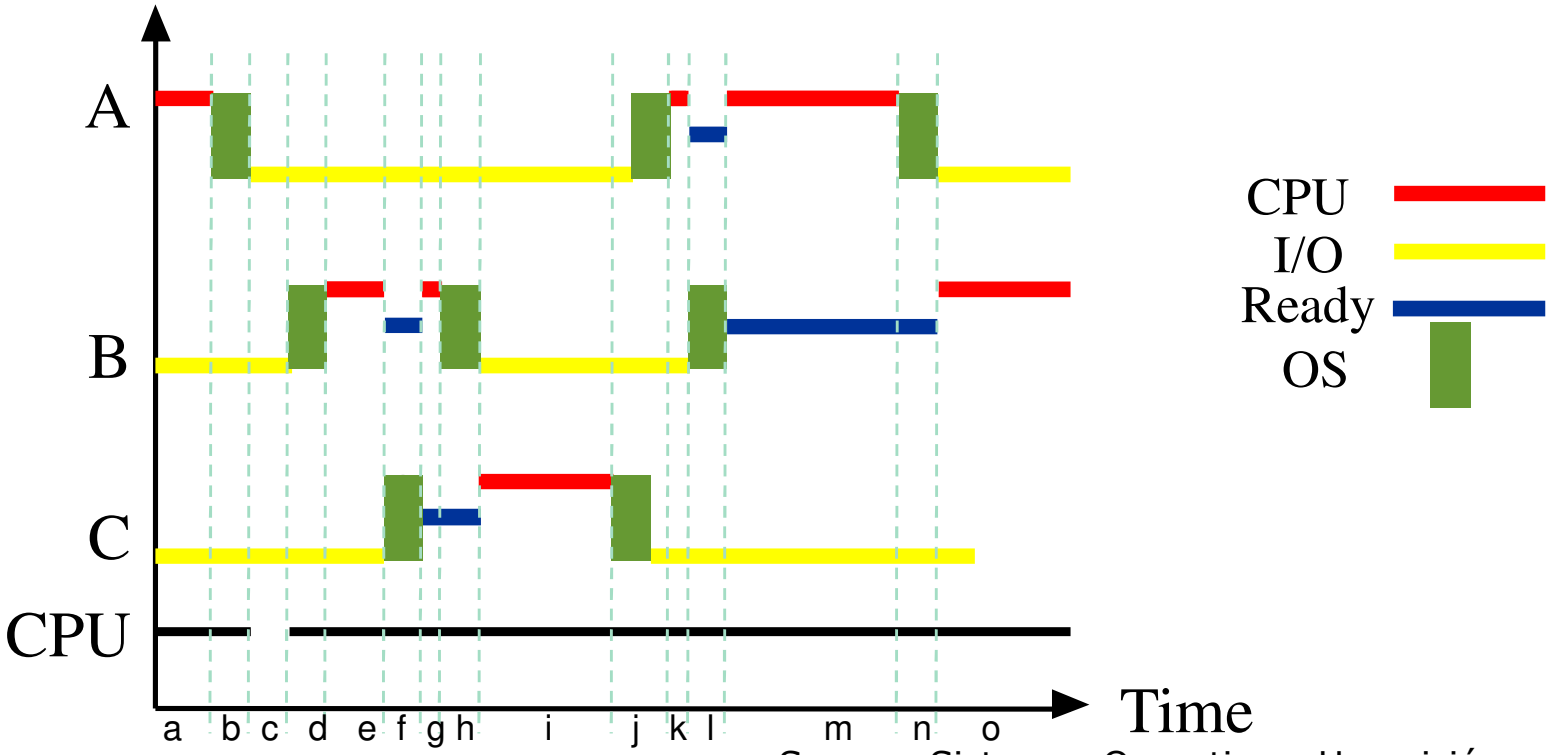
(b) With Two Suspend States

Process finishes and releases enough memory to recover a higher priority process than the ready/suspended ones

Figure 3.9 Process State Transition Diagram with Suspend States



Process Tracking



Source: Sistemas Operativos. Una visión aplicada.

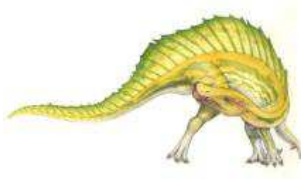
a: A in CPU, B y C blocked
b: A calls the OS for I/O
c: All processes blocked (CPU idle)
d: B I/O finishes (awake and dispatch)
e: B in execution
f: C I/O finishes (awake), B ready
g: B still in CPU and C ready
h: B performs syscall. OS dispatches C and blocks B

i: C in CPU, A y B blocked
j: C calls the OS for I/O and A awakes
k: A in execution
l: I/O interrup. calls OS to wake up B
m: A still in Run and B watis for ready
n: A blocks
o: B starts to run

CPU always busy except in c

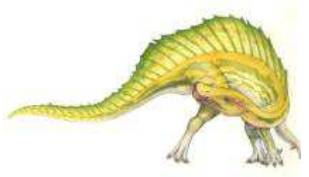
Contents

- **Process**
  - Process concept
  - Process states
  - **Process Control Block (PCB)**
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**



Process Control Block

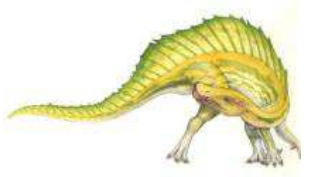
- What happens if the processor switches from process A to process B?
  - The processor needs information to get back to execute process A
  - Operating systems need a information structure



Process Control Block

- What happens if the processor switches from process A to process B?
  - The processor needs information to get back to execute process A
  - Operating systems need a information structure

Process Control Block (PCB)

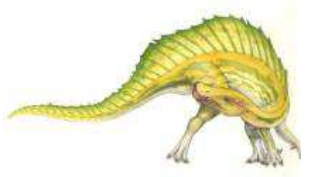


Process Control Block

- What happens if the processor switches from process A to process B?
  - The processor needs information to ge back to execute process A
  - Operating systems need a information tructure

Process Control Block (PCB)

Context Switch



Process Control Block (PCB)

Information associated with **each process**

(also called **task control block**)

- **Process state** – running, waiting, etc
- **Program counter** – location of instruction to next execute
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling information-** priorities, scheduling queue pointers
- **Memory**-management information – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files

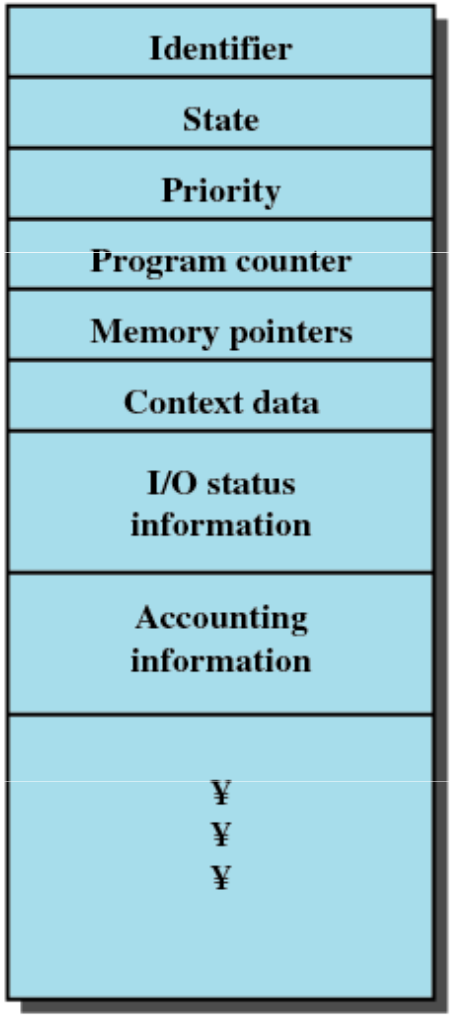
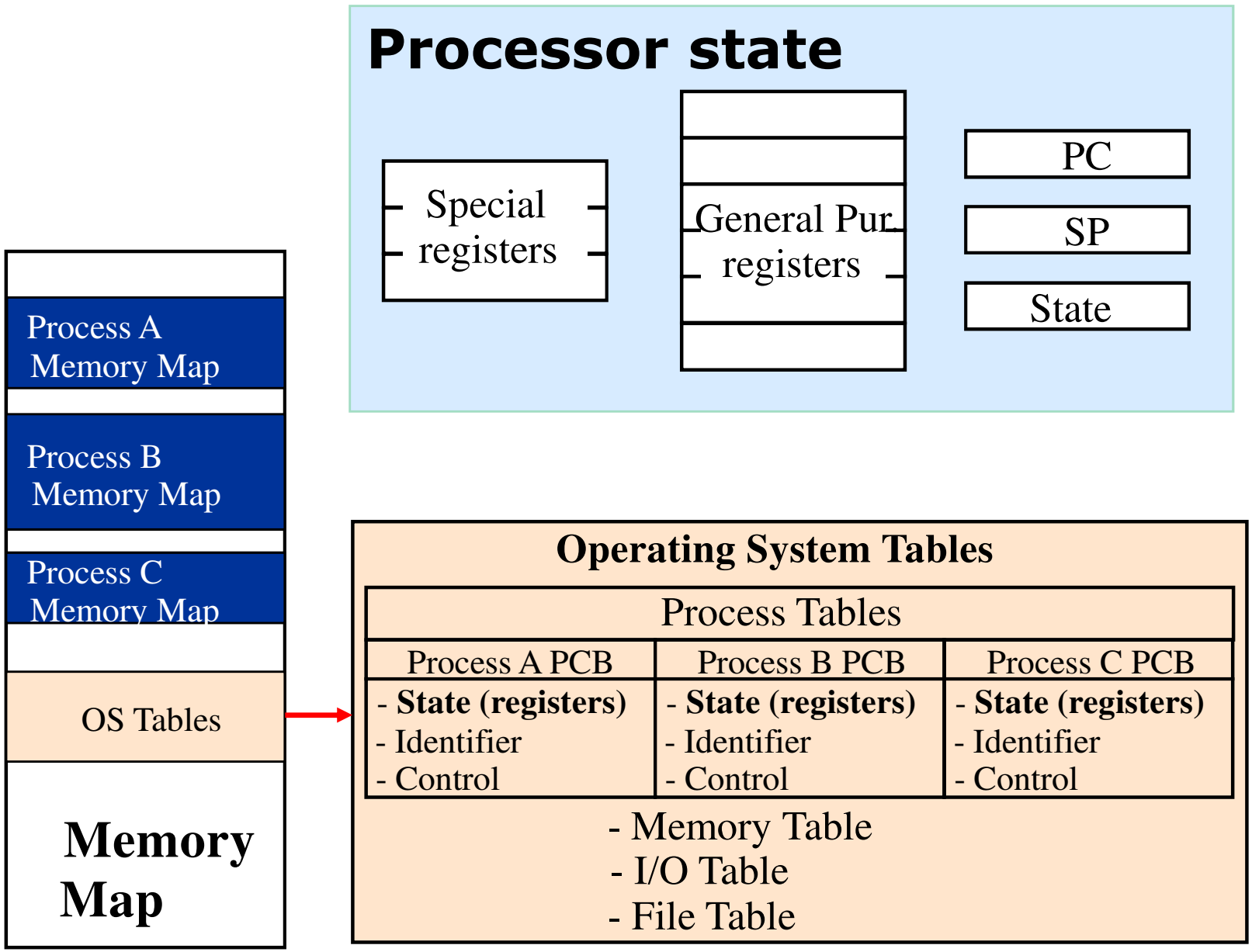
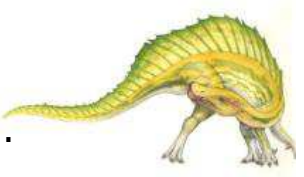


Figure 3.1 Simplified Process Control Block

Process Information



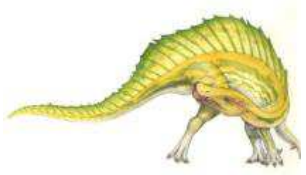
Source: Sistemas Operativos. Una visión aplicada.



What does define a process?

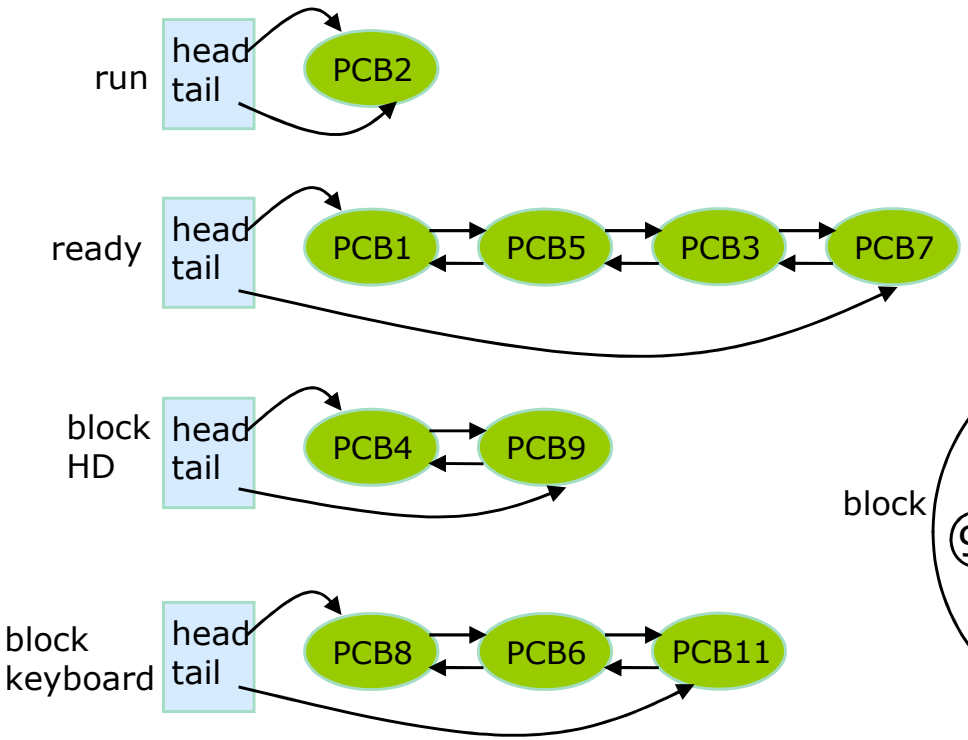
- State of the *processor* 
  - Data of the registers of the processor
- Core image
  - Data of the memory segments (code, data and stack)
- Information/status of each process
  - Process Control Block (PCB)

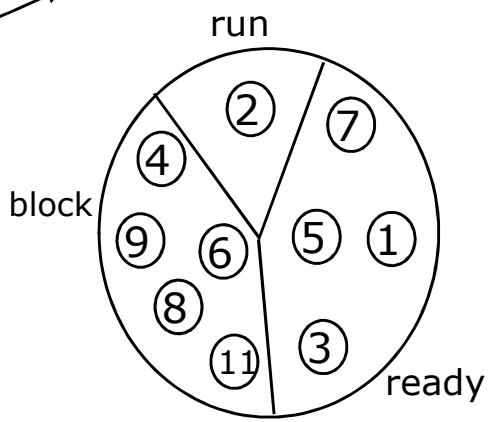
Process Image

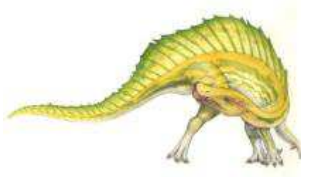


Process organization

- Processes organization
  - Process lists of a certain type
    - Run: as many processes as processors
    - Ready: sorted list by the scheduler
    - Block: several non sorted lists
      - It speeds up the search of the process to wake up

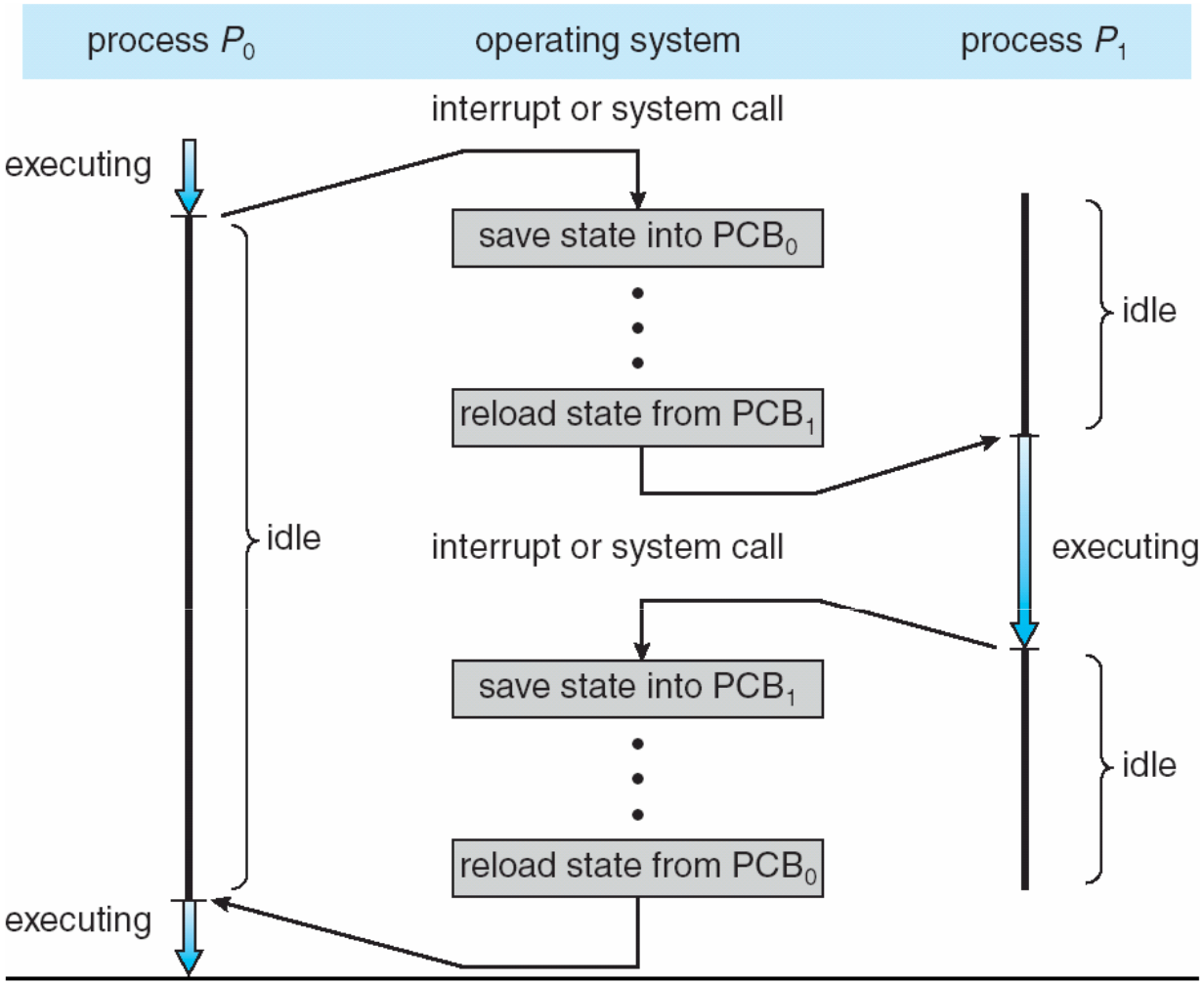






CPU Switch From Process to Process

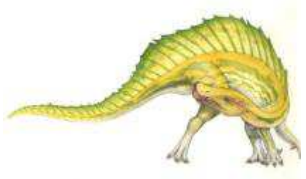
- The PCB is saved when a process is removed from the CPU and another process takes its place (context switch).



Source: Operating System Concepts. A. Silberschatz. Fig. 3.4

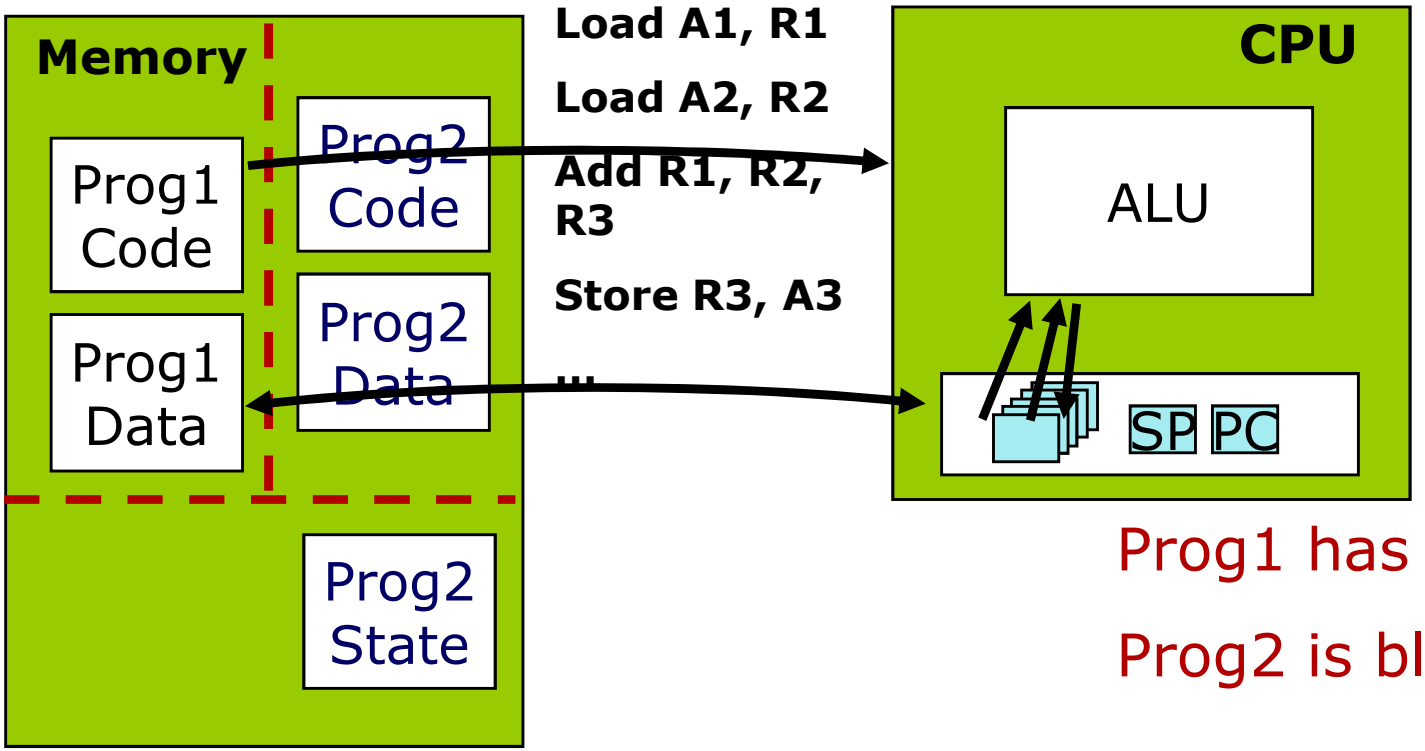
Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB ➔ the longer the context switch
- Time dependent on hardware support.
  - Varies from 1 to 1000 microseconds



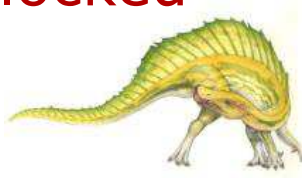
Switching among multiple processes

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



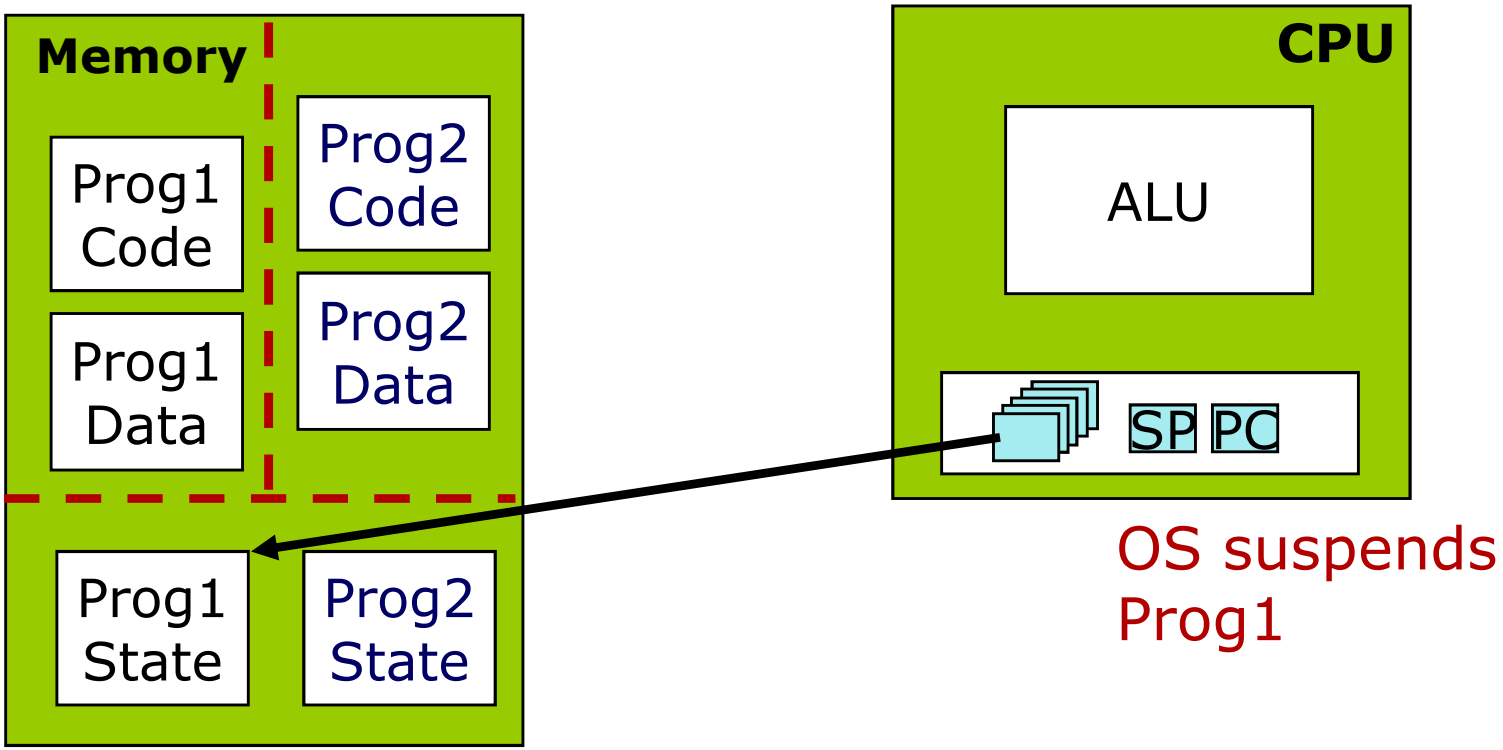
Prog1 has CPU

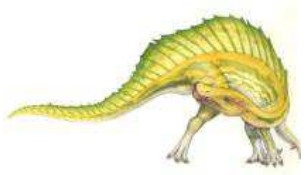
Prog2 is blocked



Switching among multiple processes

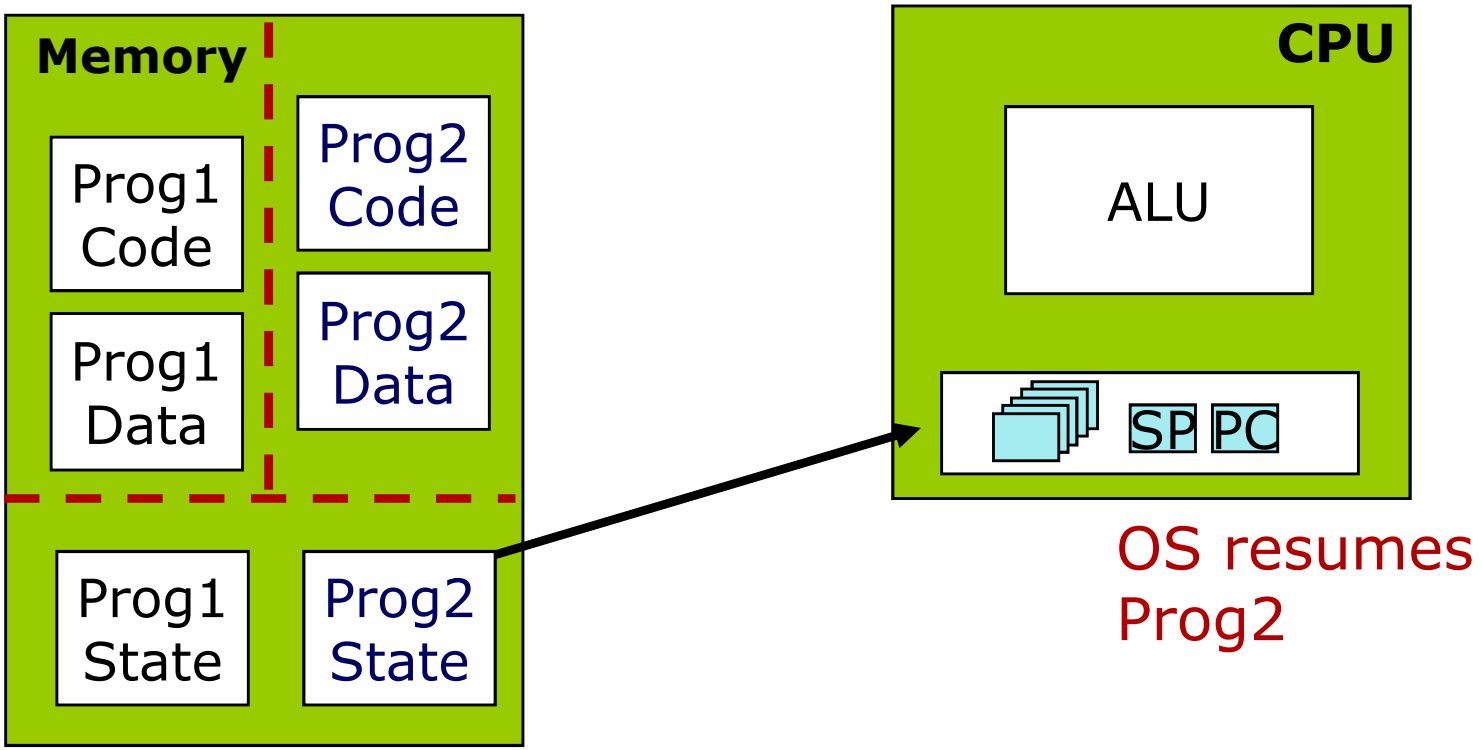
- Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed* from the same point

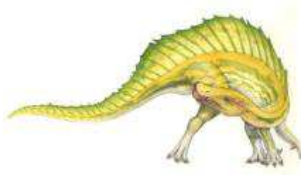




Switching among multiple processes

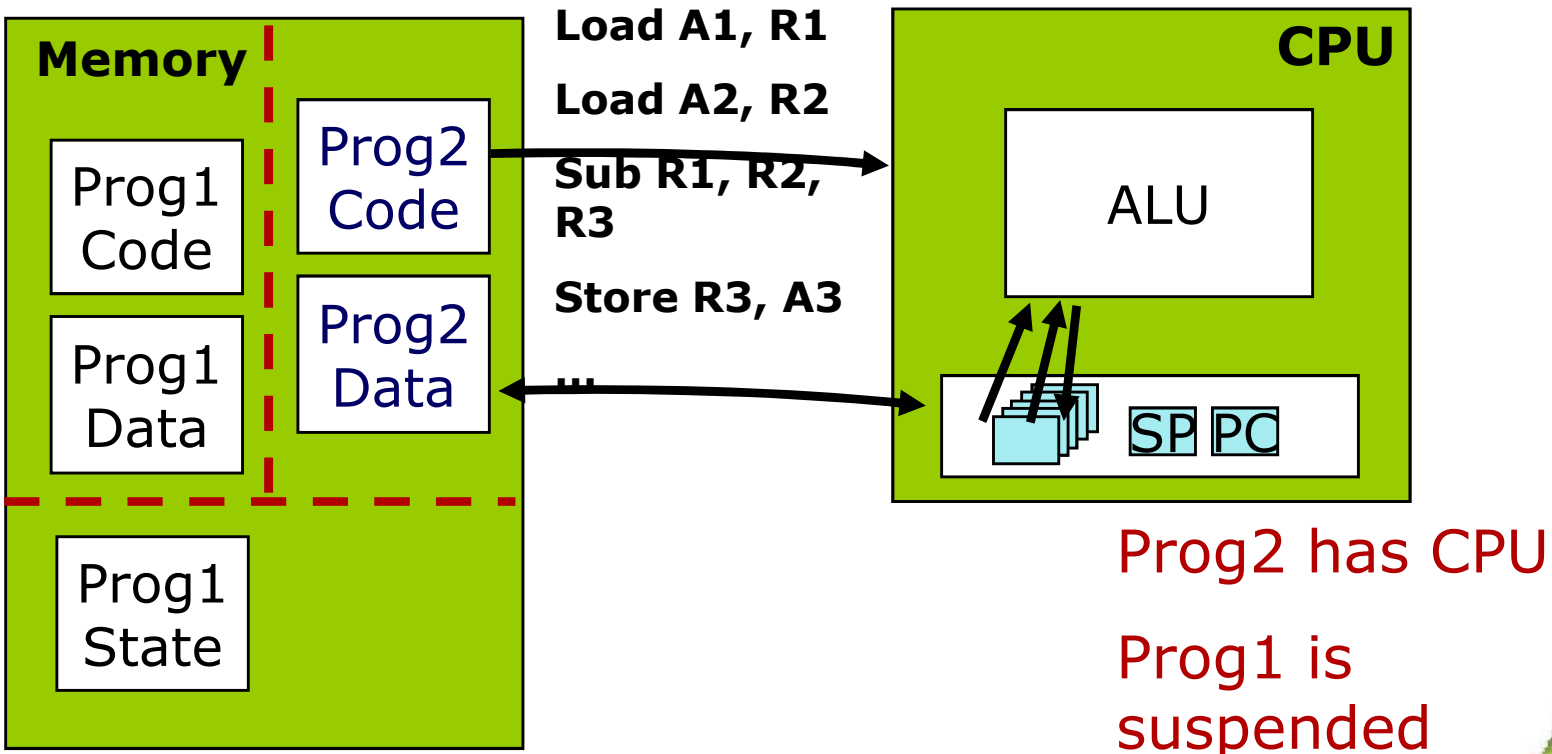
- Saving all the information about a process allows a process to be *temporarily suspended* and later *resumed*





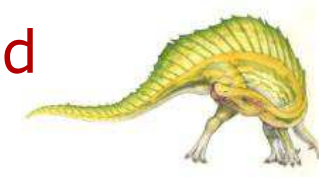
Switching among multiple processes

- Program instructions operate on operands in memory and in registers



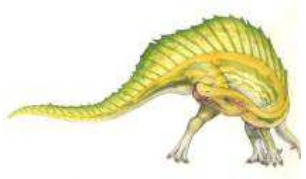
Prog2 has CPU

Prog1 is suspended



Contents

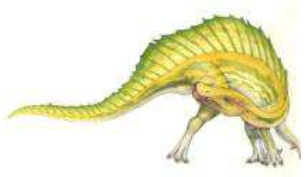
- **Process**
  - Process concept
  - Process states
  - Process Control Block (PCB)
  - Operations on Processes
- **Threads**
  - Thread concept
  - Libraries to create threads
- **Operating System Examples**
- **CPU scheduling**



How do processes get created?

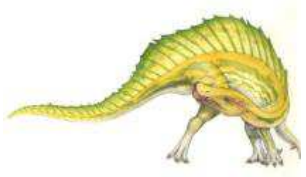
Principal events that cause process creation

- System initialization
- Initiation of a batch job
- User request to create a new process
- Execution of a process creation system call from another process



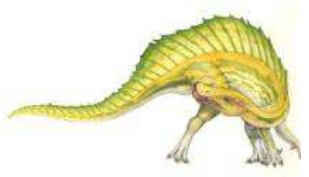
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
  - special system calls for communicating with and waiting for child processes
  - each process is assigned a unique identifying number or **process ID** (**PID**)
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate



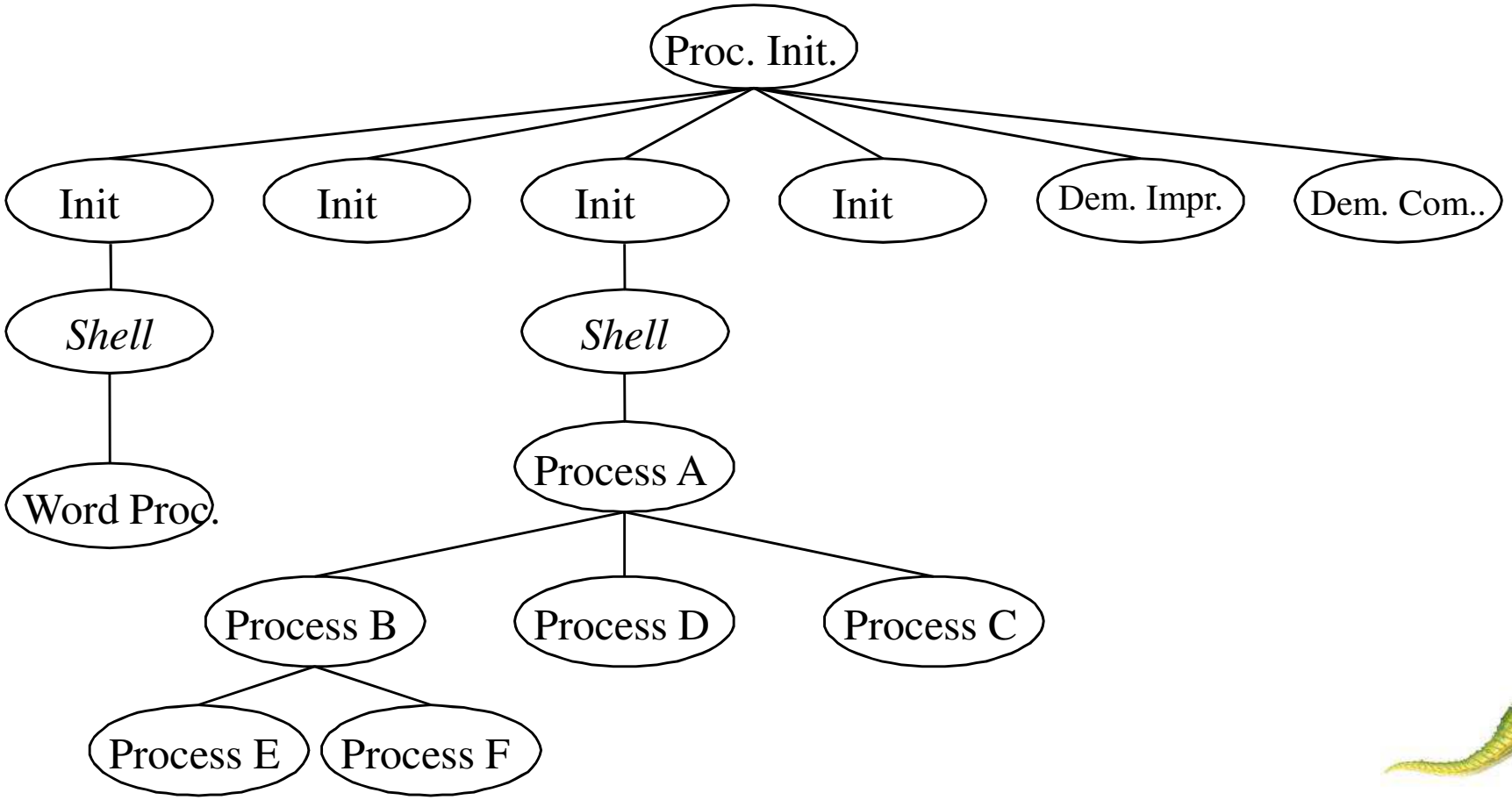
Process Creation (Cont)

- Address space
  - Child duplicate of parent (UNIX)
    - Child has copy of parent's address space
      - Enables easy communication between the two
      - The child process' memory space is replaced with a new program which is then executed. Parent can wait for child to complete or create more processes
  - Child has a program loaded into it directly (Windows or DEC VMS)
- Child processes can create their own child processes
  - Forms a hierarchy
  - UNIX calls this a "process group"
  - Windows has no concept of process hierarchy
    - all processes are created equal

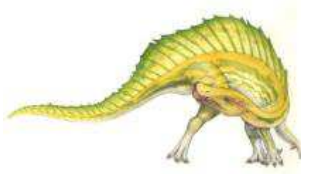


A tree of processes in Unix

- The first process is Init ()
- Init creates login daemons
- Login becomes a shell
- Using the shell user spawns new processes

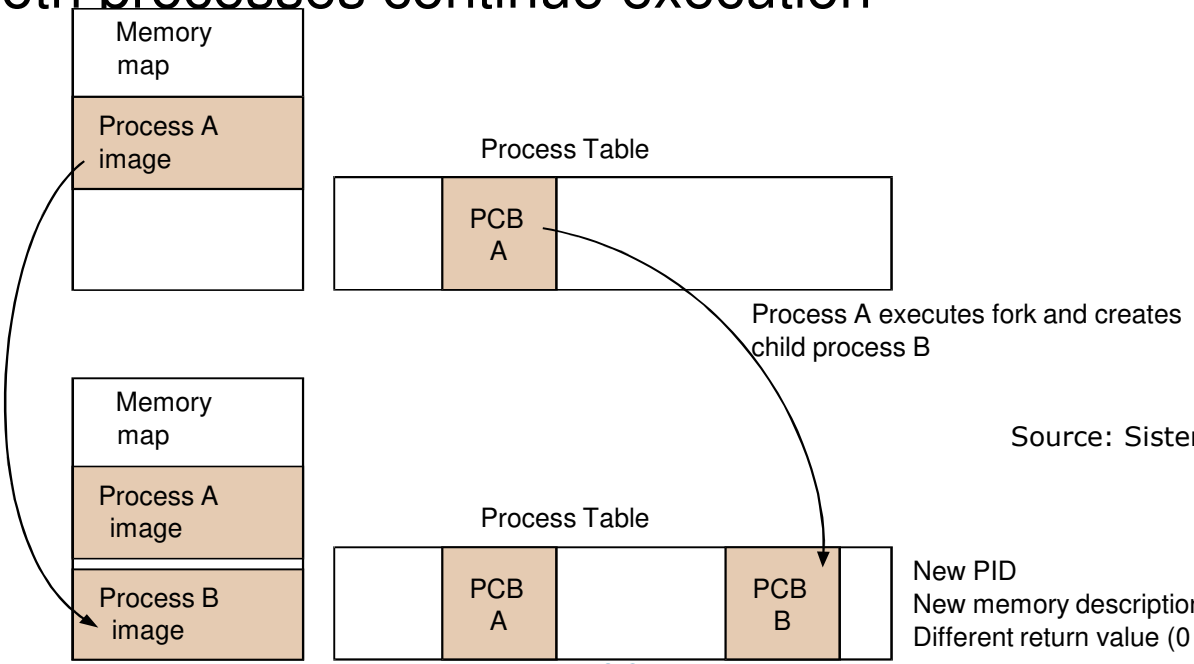


Source: Sistemas Operativos. Una visión aplicada.

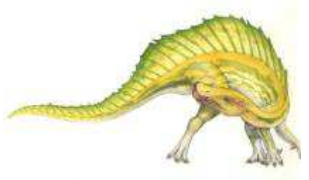


Process creation in UNIX (POSIX)

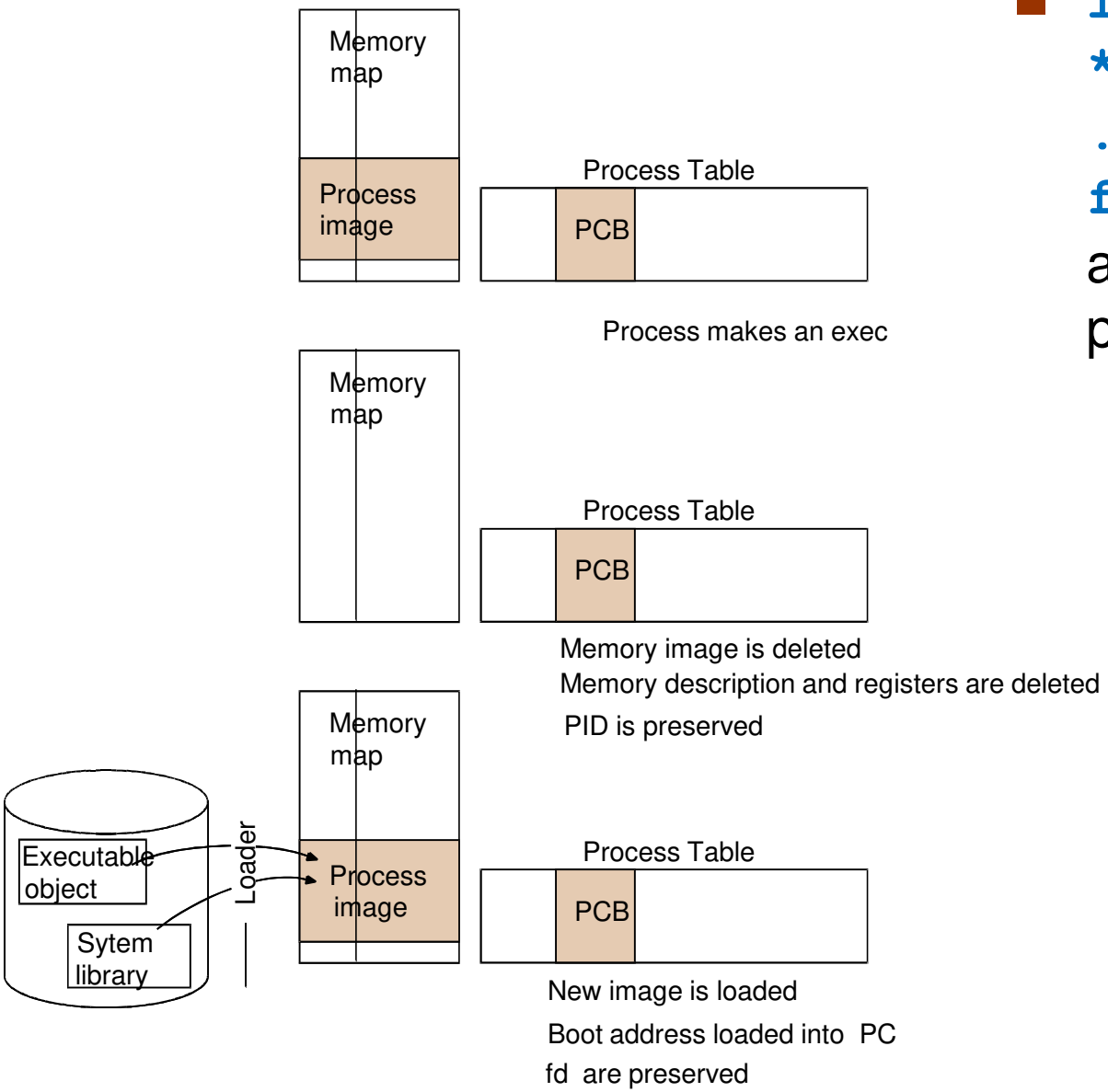
- Process creation: **`fork`** and **`exec`**
  - **`pid_t fork(void)`** system call
    - creates new process which has a copy of the address space of the original process and returns in both processes (parent and child), but with a different return value:
      - **0 to child, pid's child to parent and -1 in error case**
    - Simplifies parent-child communication
    - Both processes continue execution



Source: Sistemas Operativos. Una visión aplicada.

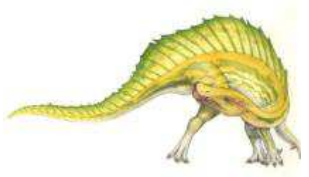


Process creation in Unix



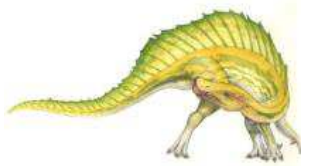
Source: Sistemas Operativos. Una visión aplicada.

- `int execlp(const char *file, const char *arg, ...)` system call used after a `fork()` to replace the process' address space with a new program
  - Loads a binary file into memory and starts execution
  - Parent can then create more children processes or issue a `wait()` system call to move itself off the ready queue until the child completes



C Program Forking Separate Process

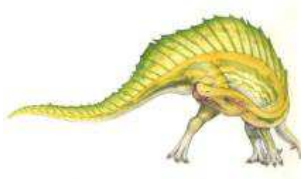
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```



Example: process creation in UNIX

csh (pid = 22)

```
…

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



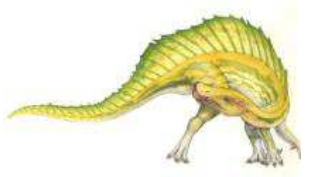
Process creation in UNIX example

csh (pid = 22)

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

csh (pid = 24)

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



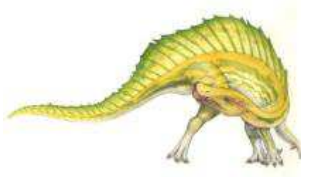
Process creation in UNIX example

csh (pid = 22)

```
…

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

csh (pid = 24)

```
…

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



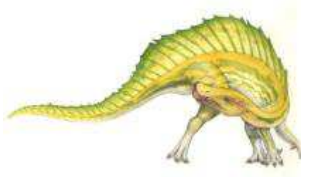
Process creation in UNIX example

csh (pid = 22)

```
…

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

csh (pid = 24)

```
…

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



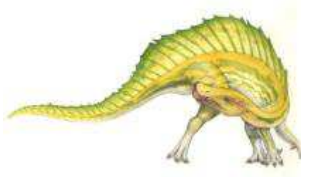
Process creation in UNIX example

csh (pid = 22)

```
…

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

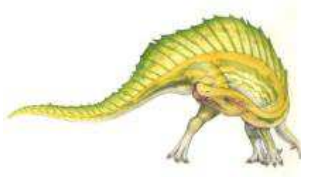
ls (pid = 24)

```
//ls program

main(){

   //look up dir

   …

}
```



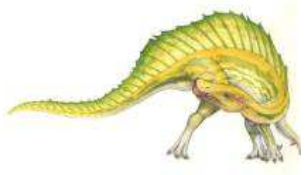
Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```



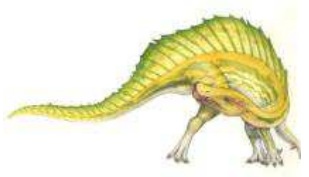
Others POSIX services

- All processes have a unique process id
  - `getpid(), getppid()` system calls allow processes to get their information
    - `pid_t getpid(void)` —returns the process ID of the calling process
    - `pid_t getppid(void)` –returns the parent process ID of the calling process



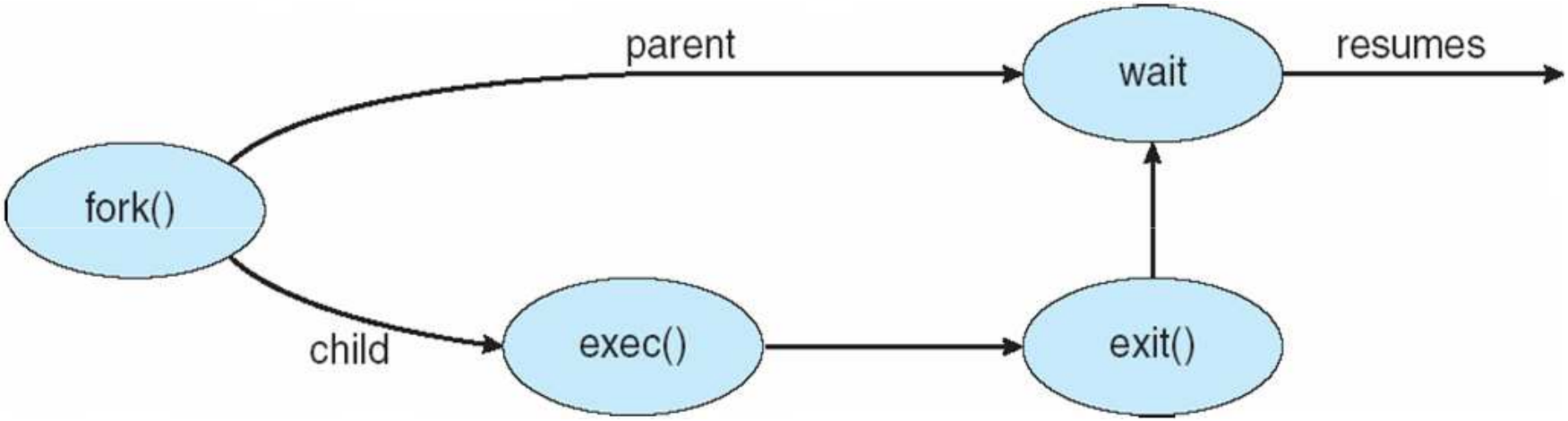
Process Termination

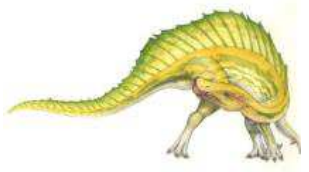
- Process executes last statement and asks the operating system to delete it by using the `exit()` system call
  - Process may return a status value to parent via `wait()`
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes `abort()` in UNIX
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates - `cascading termination`



Process Termination in Unix

- In UNIX, a process can be terminated via the **`exit`** system call.
  - Parent can wait for termination of child by the **`wait`** system call
  - **`wait`** returns the process identifier of a terminated child so that the parent can tell which child has terminated

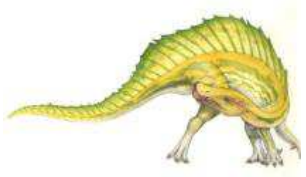




Source: Operating System Concepts. A. Silberschatz. Fig. 3.10

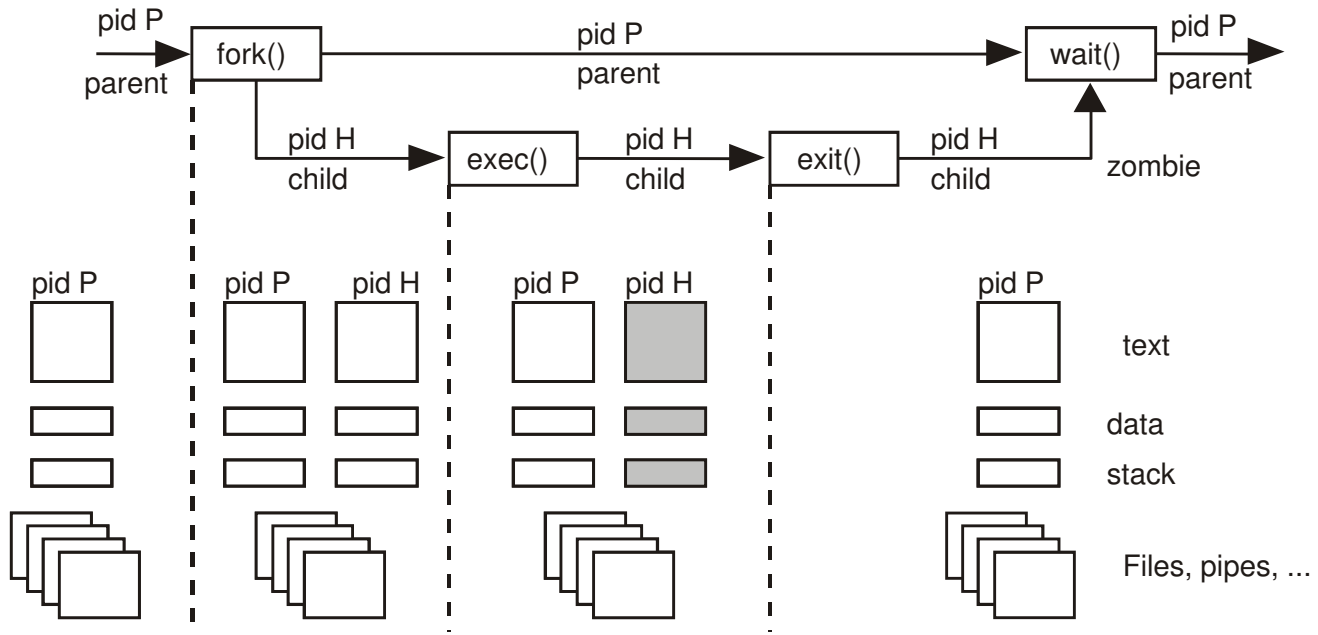
Wait() and Exit() (POSIX services)

- `int exit (int status);`
  - Arguments: code of returning to the parent
  - Description:
    - It finishs process execution
    - It closes all fd
    - It releases all process resources

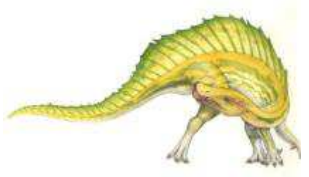


Wait() and Exit() (POSIX services)

- `pid_t wait (int *status);`
  - Arguments: returns the exit status of the child
  - Description:
    - This system call suspends execution of the calling process until one of its children terminates (a parent can wait till its child finishes)
    - It returns pid and exit status of the child (-1 in error case)

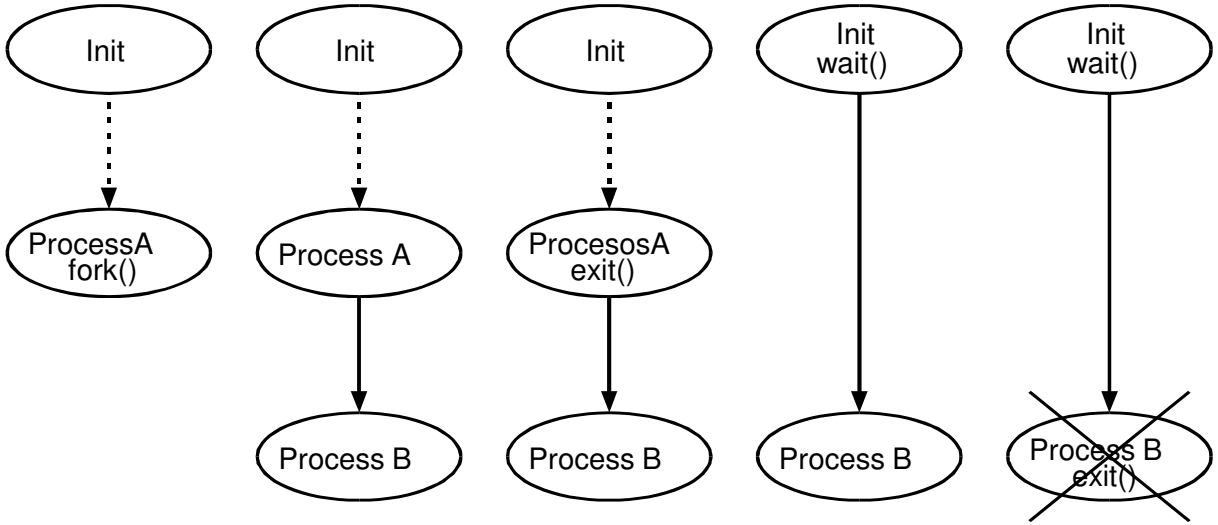


Source: Sistemas Operativos. Una visión aplicada.

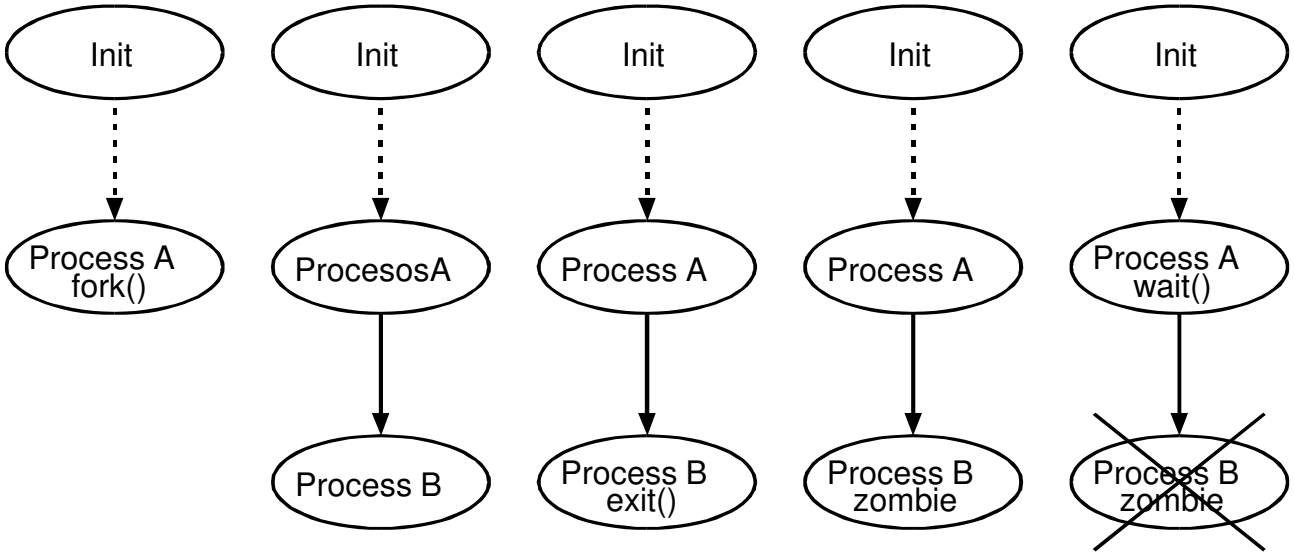


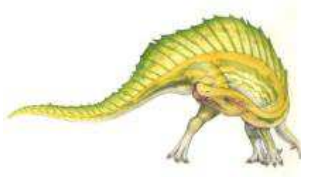
Process Termination in Unix

- If a parent terminates, all children are assigned the **init** process as their new parent

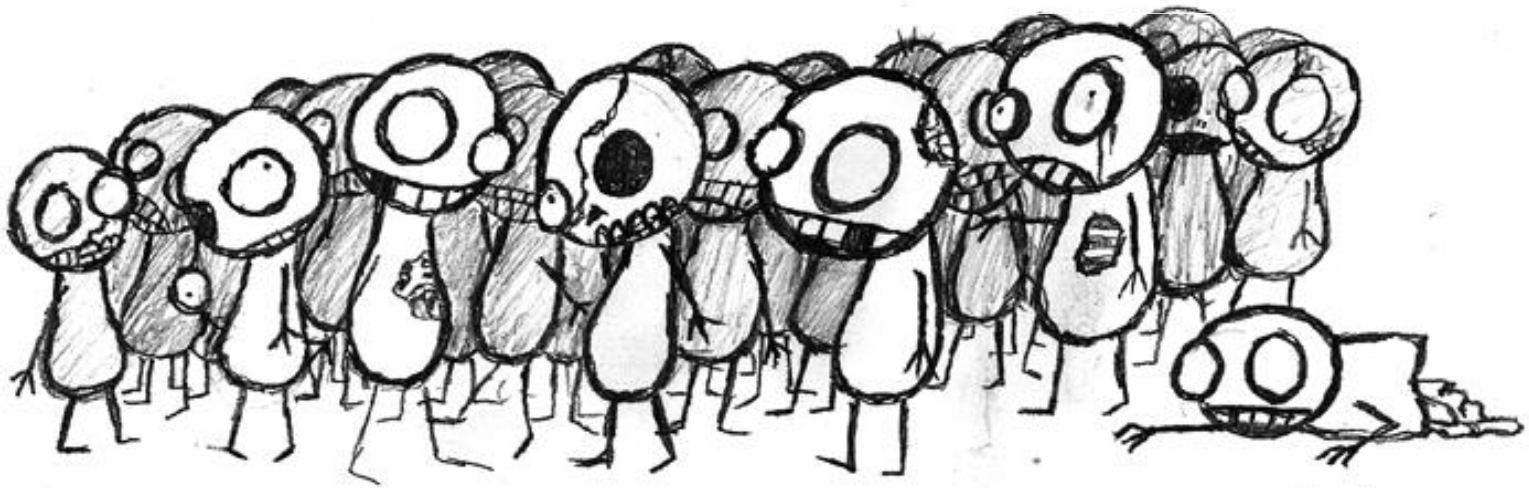


- Zombie: childrens terminate (exit) and father doesn't call to wait





Source: Sistemas Operativos. Una visión aplicada.

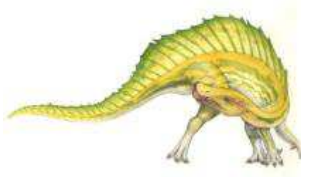


Source: https://ninefold.com/blog/2014/11/25/threads/

Example: process creation in UNIX

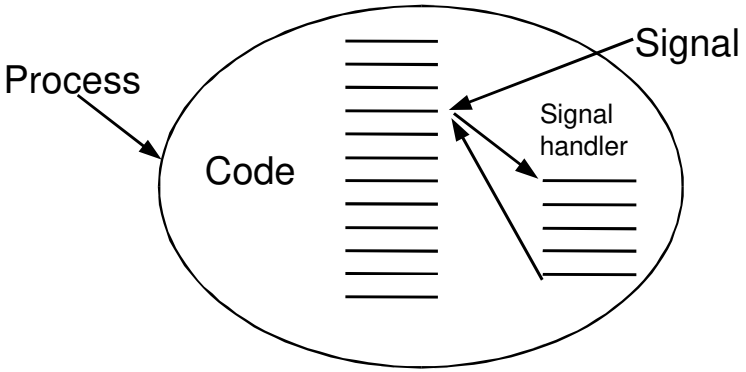
- A process launches command “ls –l”

```
#include <sys/types.h>
#include <stdio.h>
/* program executes command ls -l */
main() {
   pid_t pid;
   int status;
   pid = fork();
   if (pid == 0)    { /* child process */
      execlp("ls","ls","-l",NULL);
      exit(-1);
   }
   else        /* parent process */
     while (pid != wait(&status));
   exit(0);
}
```

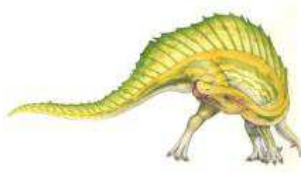


Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- Process termination, signaling
  - *signal(), kill()* system calls allow a process to be terminated or have specific signals sent to it
- The signals are process interrupts
  - Signals are sent:
    - From process to process with kill
    - From OS to a process

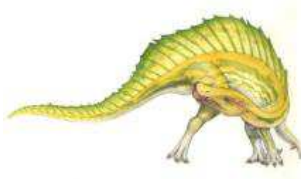


Source: Sistemas Operativos. Una visión aplicada.



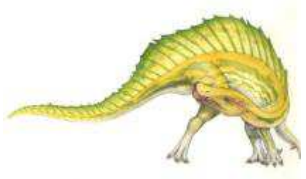
Signal Handling

- Types of signals (kill –l shows the list)
  - SIGKILL -9- (Kill : terminate immediately)
  - SIGTERM -15- (Termination : request to terminate)
  - SICHLD -17- (Child process terminated, stopped or continued)
  - SIGCONT -18- (Continue if stopped)
  - SIGSTOP -19- (Stop executing temporarily)
  - SIGTTIN -21. (Background process attempting to read from tty)
  - SIGTTOU -22- (Background process attempting to write to tty)



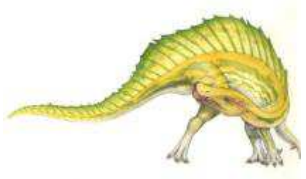
Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default



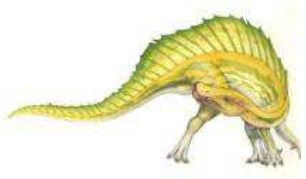
Signal Handling in Unix

- Signal handlers can be installed with the `signal()` system call.
  - When the signal is intercepted, the signal handler is invoked
- If a signal handler is not installed for a particular signal, the default handler is used.
- The process can also specify two default behaviors, without creating a handler:
  - ignore the signal (SIG_IGN)
  - use the default signal handler (SIG_DFL)
- The `sigprocmask()` call can be used to block and unblock delivery of signals.



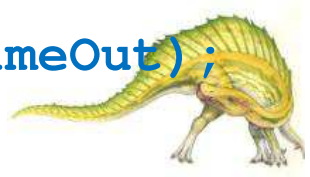
Handling Signals (POSIX)

- `int kill(pid_t pid, int sig)`
  - It sends to the process “pid” the signal “sig”
- `int sigaction(int sig, struct sigaction *act, struct sigaction *oact)`
  - It allows to specify the action to be taken when the signal “sig” is received
- `int pause(void)`
  - It blocks the process until the reception of a signal
- `unsigned int alarm(unsigned int seconds)`
  - It generates the reception of signal SIGALARM after “seconds” seconds
- `sigprocmask(int how, const sigset_t *set, sigset_t *oset)`
  - It is used to explore or modify the signal mask of a process



Win32 services

- To create a process
  - `BOOL CreateProcess(….);`
- To finish the execution of a process
  - `VOID ExitProcess(UINT nExitCode);`
- To obtain the end code of a process
  - `BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpdwExitCode);`
- To finish the execution of another process
  - `BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode);`
- To wait till the end of a process
  - `DWORD WaitForSingleObject(HANDLE hObject, DWORD dwTimeOut);`
  - `DWORD WaitForMultipleObjects(DWORD cObjects, LPHANDLE lphObjects, BOOL fWaitAll, DWORD dwTimeOut);`



Summary (POSIX & Win32 APIs)

- Process identification
  - `getpid, GetCurrentProcessId`
- Process environment variables
  - `getenv, GetEnvironmentStrings`
  - Path, home directory, working directory, personal directory
- Process creation: `fork, CreateProcess`
  - Program transformation: `exec`
- Waiting for termination of a process: `wait, WaitForSingleObject, WaitForMultipleObjects`
- Process termination: `exit, ExitProcess`
- Send signal to process: `kill, TeminateProcess`