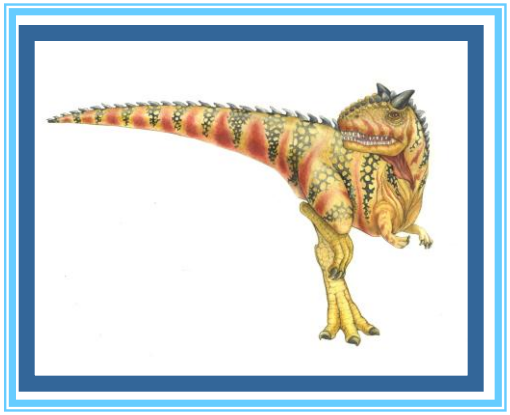
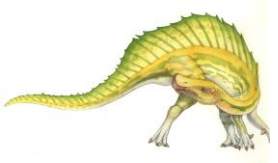
Chapter 3: Processes

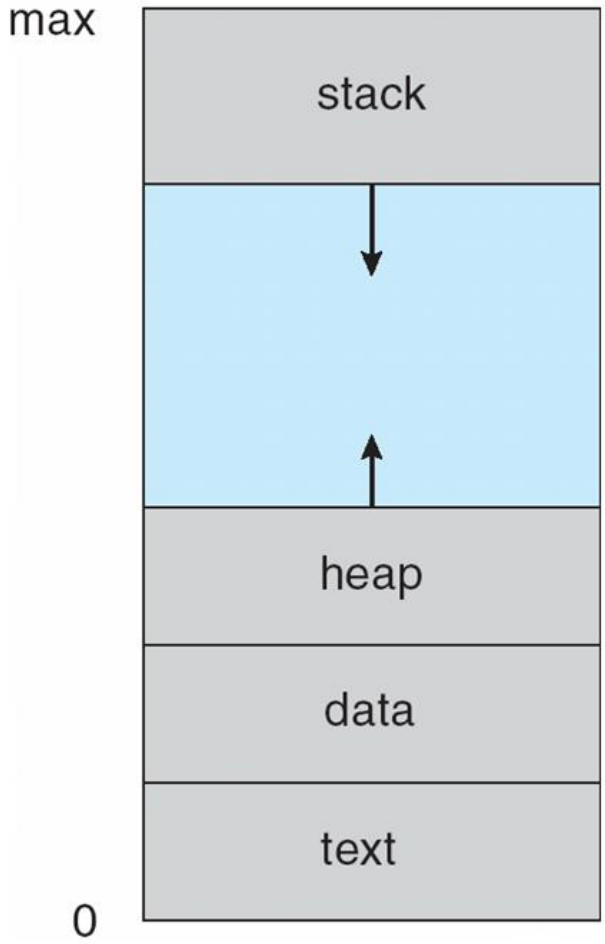


Process Concept

- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section



Process in Memory



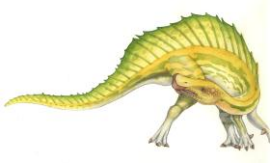
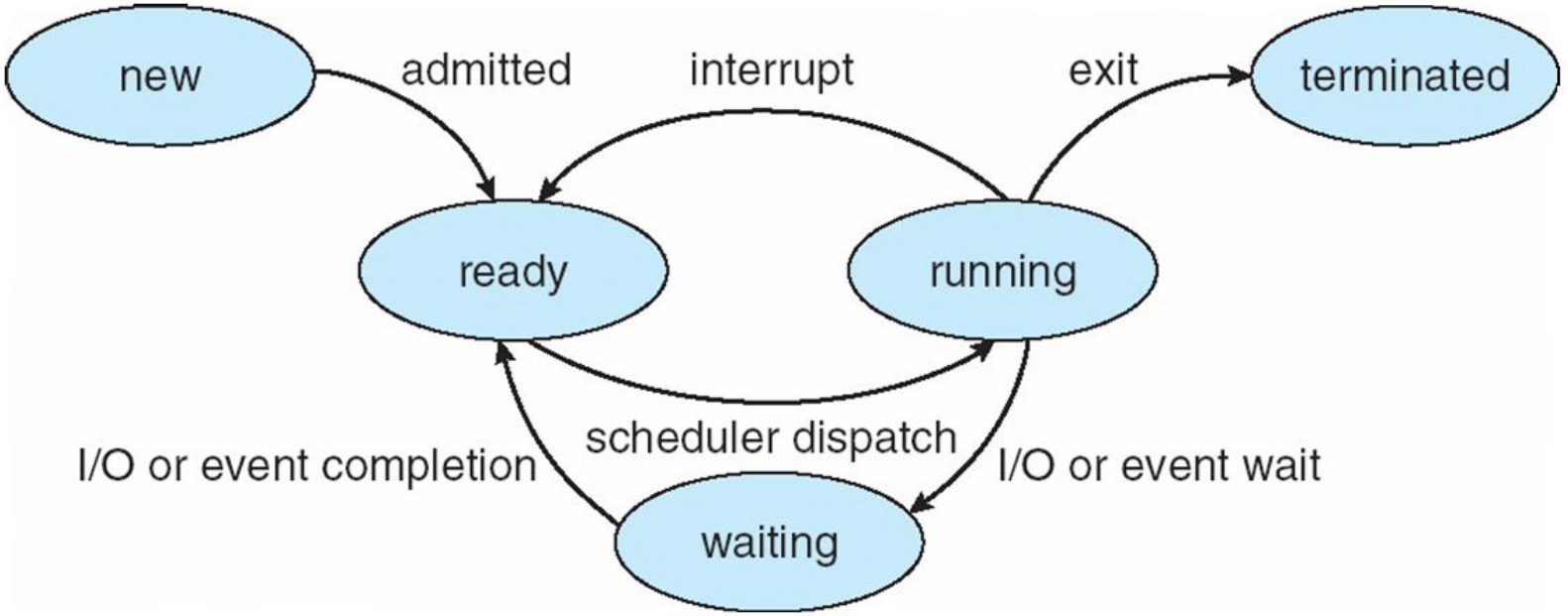
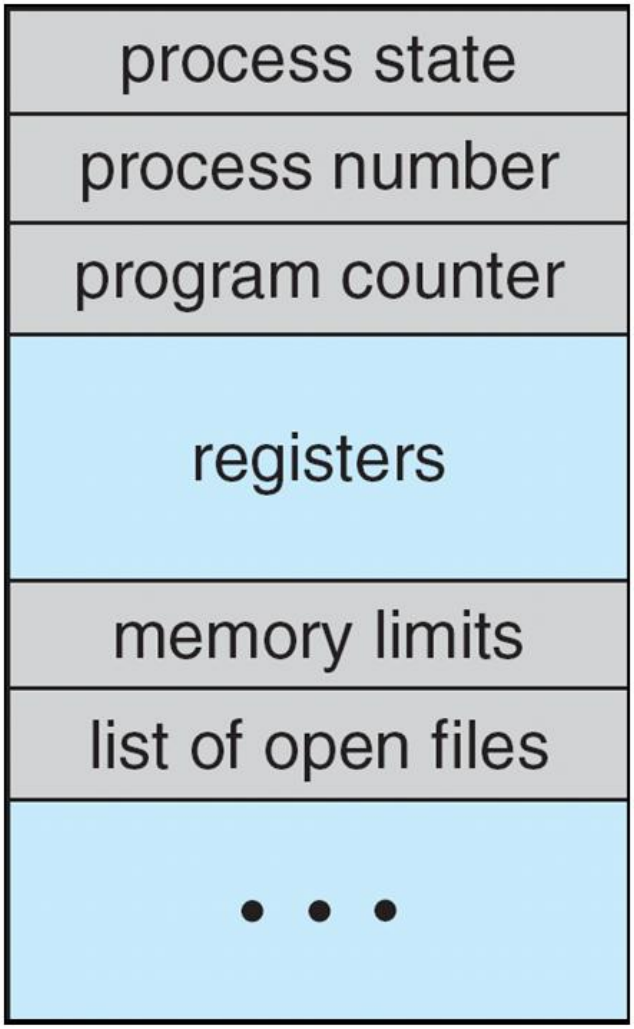
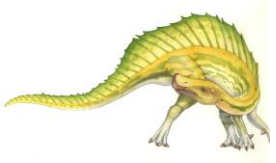


Diagram of Process State

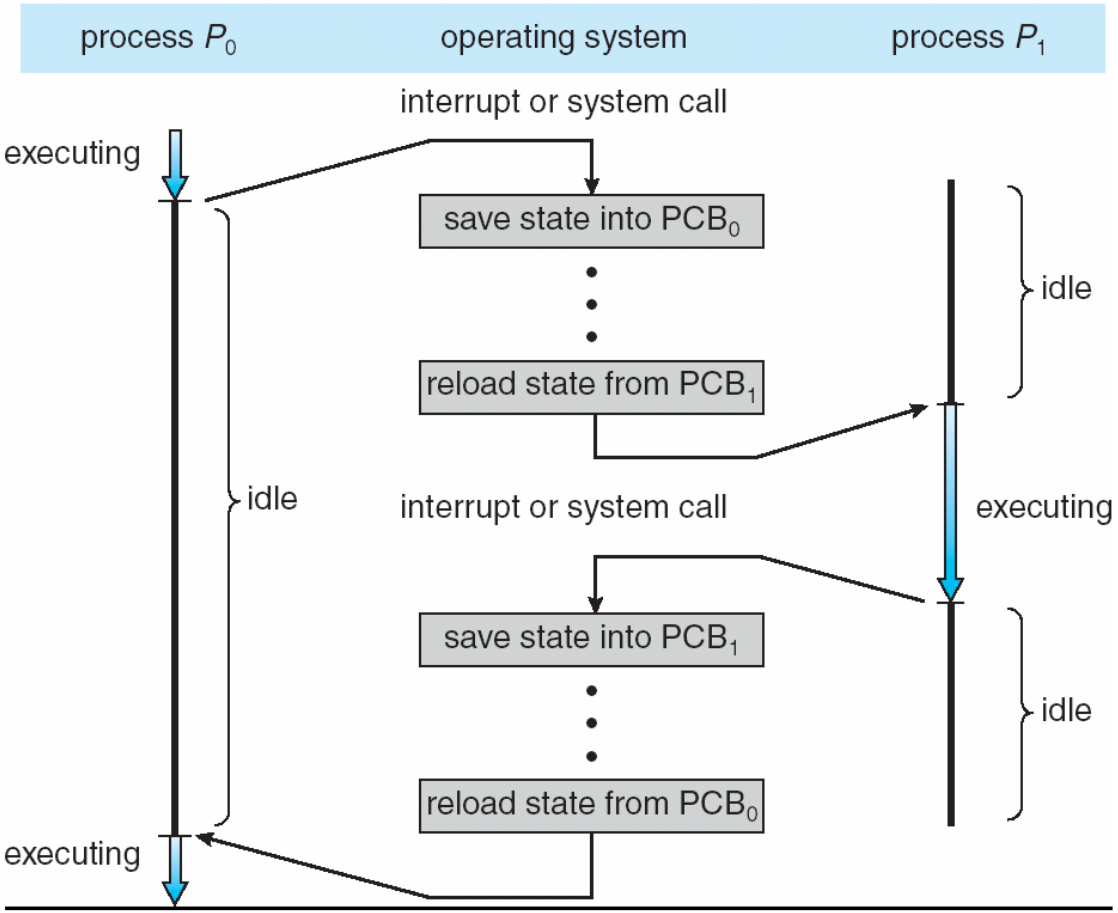


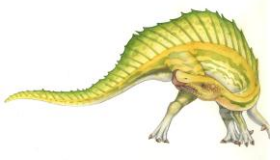
Process Control Block (PCB)





CPU Switch From Process to Process

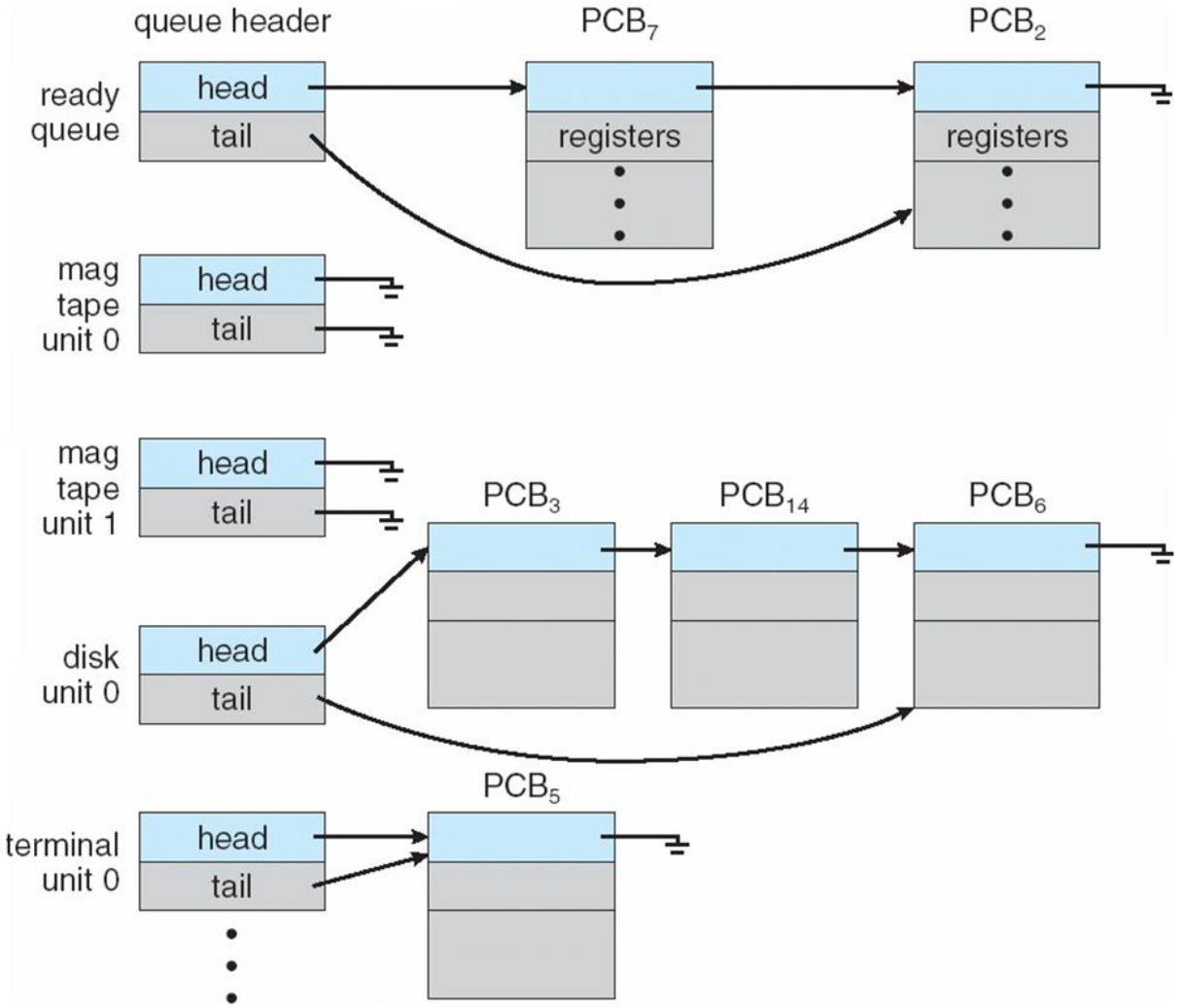




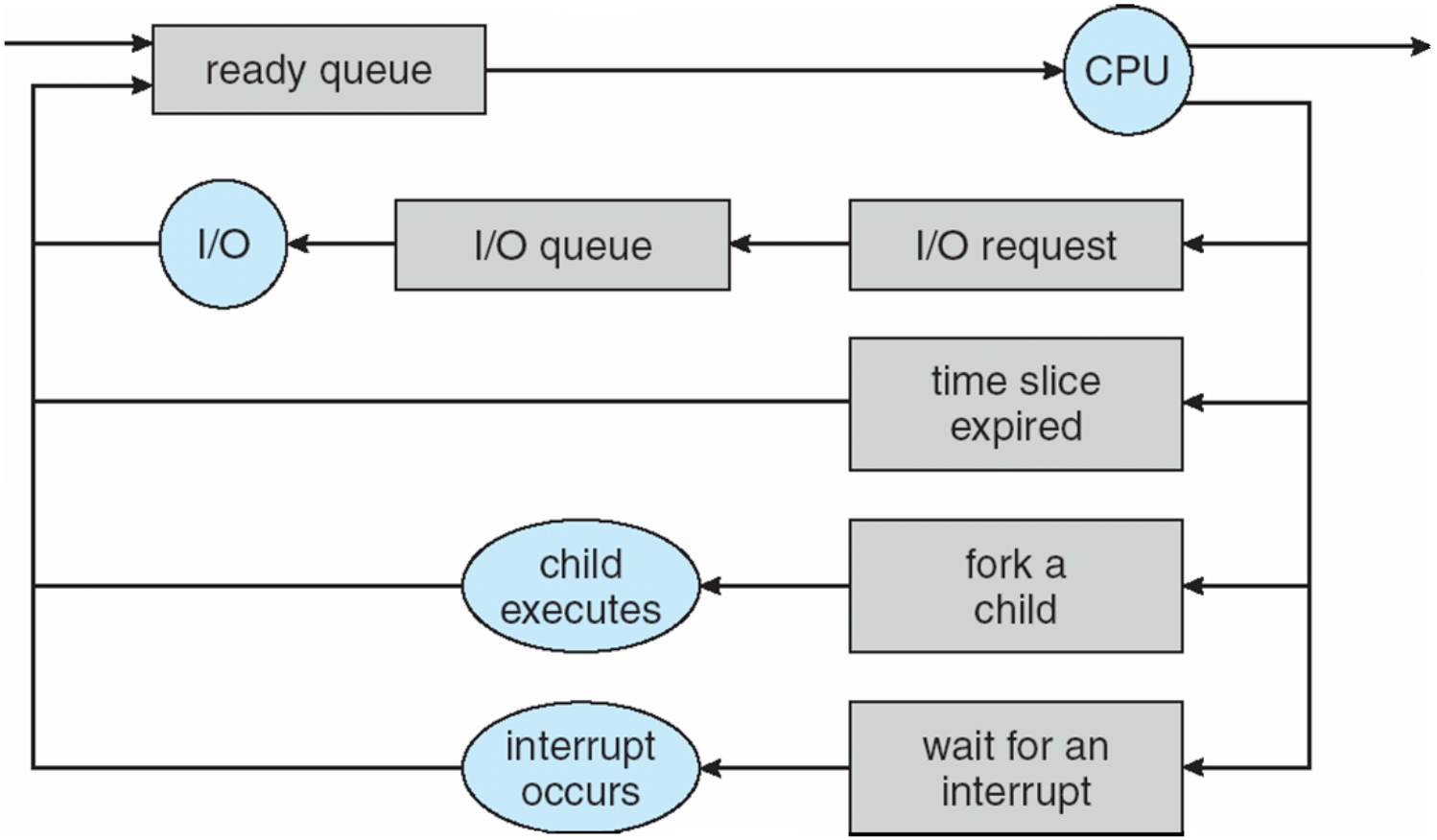
Process Scheduling Queues

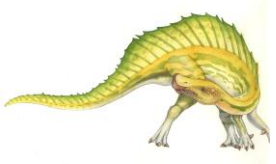
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



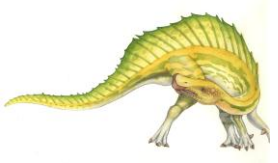
Representation of Process Scheduling





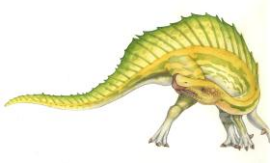
Context Switch

- 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



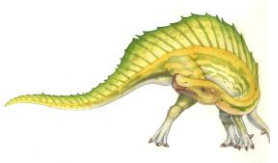
Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**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

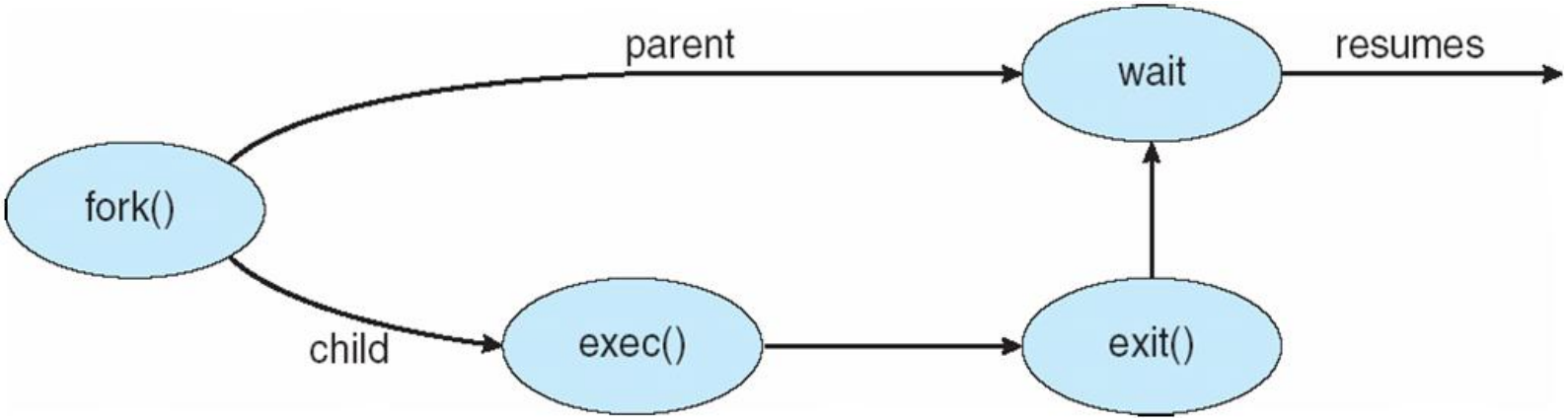


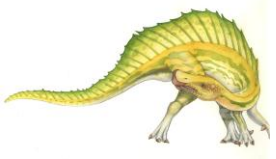
Process Creation (Cont)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program



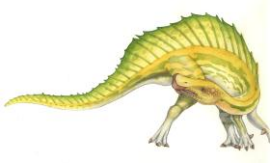
Process Creation



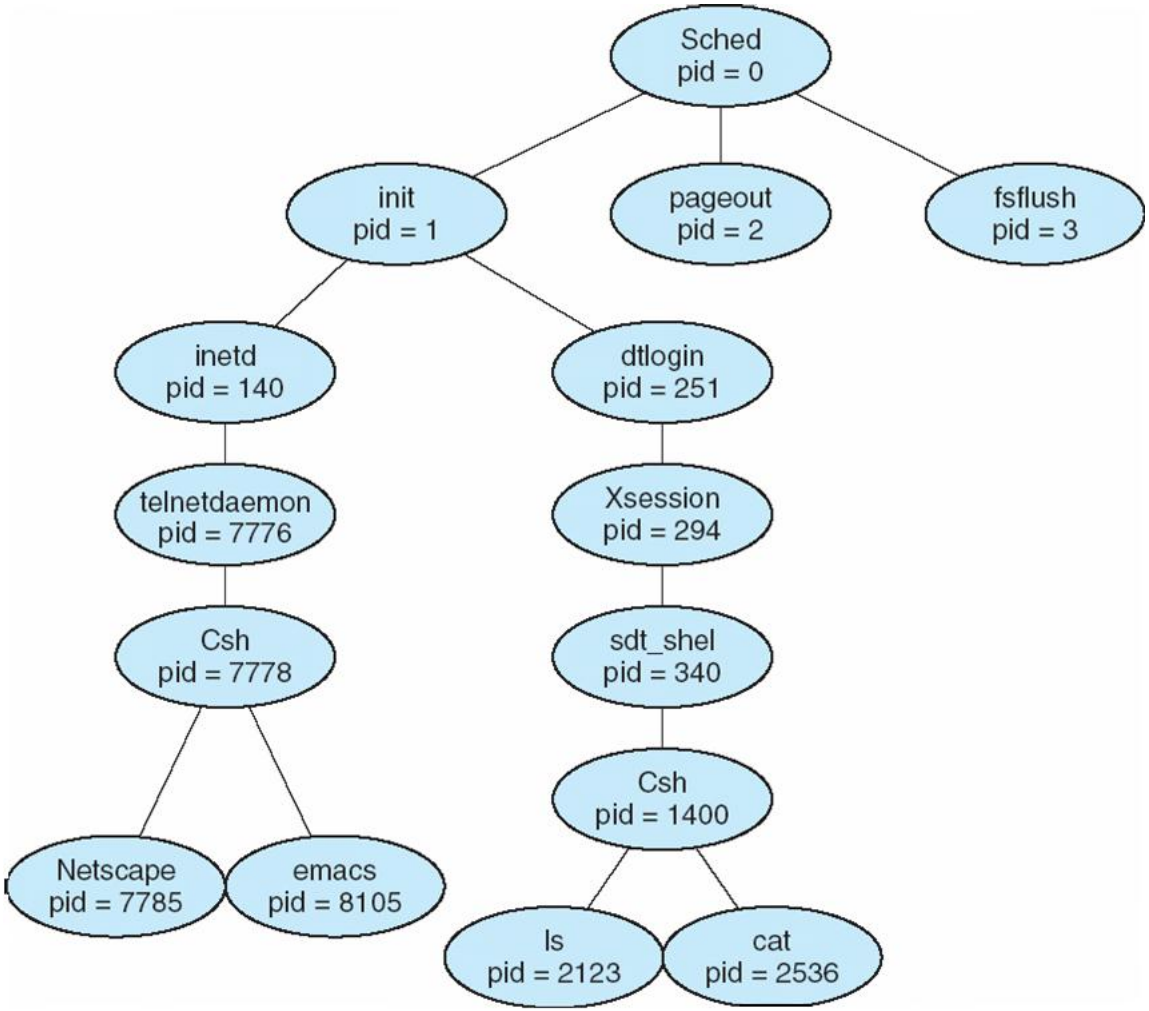


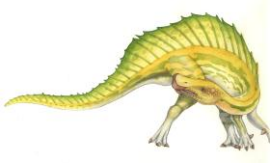
C Program Forking Separate Process

```
int main()
{
pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-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");
                exit(0);
        }
}
```



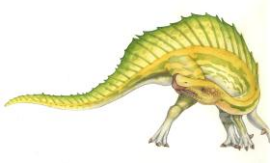
A tree of processes on a typical Solaris





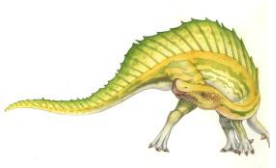
Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**)
  - Output data from child to parent (via **wait**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**



C Program Forking Separate Process

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

int global;

int main()
{
    pid_t child_pid;
    int status;
    int local = 0;
    /* now create new process */
    child_pid = fork();

    if (child_pid >= 0) /* fork succeeded */
    {
        if (child_pid == 0)
        {
            printf("child process!\n");

            // Increment the local and global variables
            local++;
            global=5;
           printf("child PID =  %d, parent pid = %d\n",
getpid(), getppid());
      printf("\n child's local = %d, child's global =
%d\n",local,global);

            char *cmd[] = {"whoami",(char*)0};
            return execv("/usr/bin/",cmd); // call whoami
command

        }
        else /* parent process */
        {
            printf("parent process!\n");
            printf("parent PID =  %d, child pid = %d\n",
getpid(), child_pid);
            wait(&status); /* wait for child to exit, and store
child's exit status */
            printf("Child exit code: %d\n",
WEXITSTATUS(status));

            //The change in local and global variable in child
process should not reflect here in parent process.
            printf("\n Parent'z local = %d, parent's  global =
%d\n",local,global);

            printf("Parent says bye!\n");
            exit(0);  /* parent exits */
        }
    }
    else /* failure */
    {
        perror("fork");
        exit(0);
    }
}
```



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

int global; /* In BSS segement, will automatically be assigned '0'*/

int main()
{
    pid_t child_pid;
    int status;
    int local = 0;
    /* now create new process */
    child_pid = fork();

    if (child_pid >= 0) /* fork succeeded */
    {
        if (child_pid == 0) /* fork() returns 0 for the child process */
```