MODERN OPERATING SYSTEMS
Third Edition

ANDREW S. TANENBAUM

Chapter 2
Processes and Threads

What Is a Process?

- A program is an executable file.
- A process is a program that is being executed.
- Each process has its own environment:

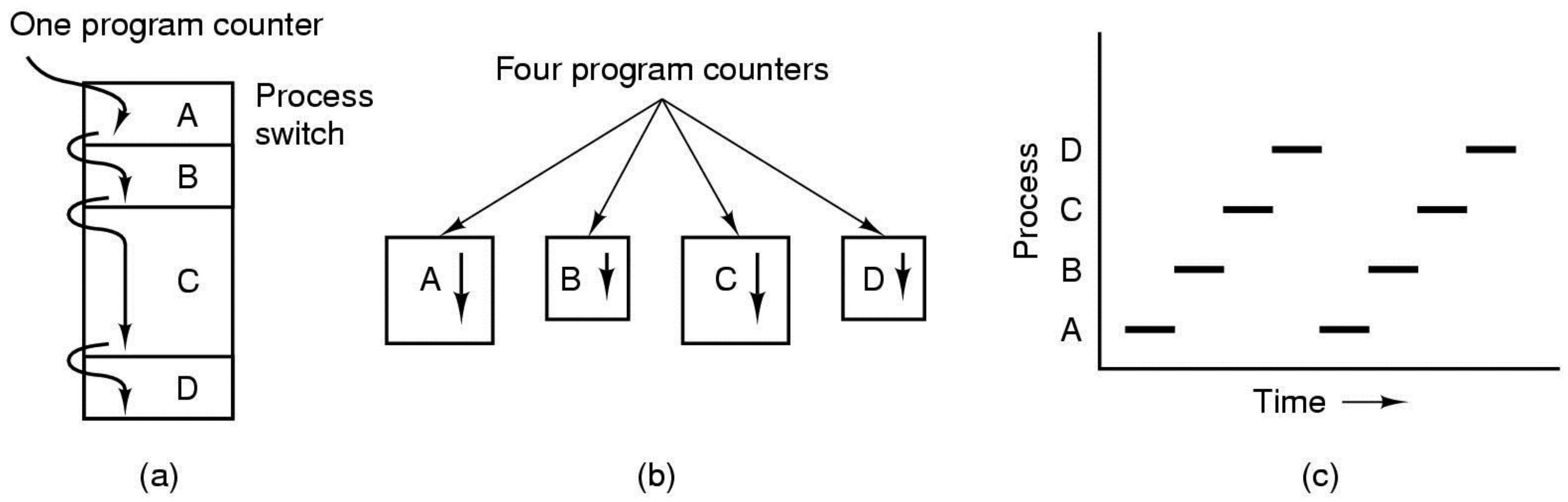
process environment

Program name
Internal data
Open Files
Current Directory
additional parameters

User and Group ID
Process ID (PID)
Parent PID (PPID)
Program variables

- To see the PID of your current shell process, type:
  $ echo $$

The Process Model



(a) Multiprogramming of four programs.
(b) Conceptual model of four independent, sequential processes.
(c) Only one program is active at once.

The Process Model

顺序进程-〉进程-〉多道程序设计

程序计数器、寄存器、变量当前值

虚拟CPU（逻辑的）-〉物理CPU（实际的）

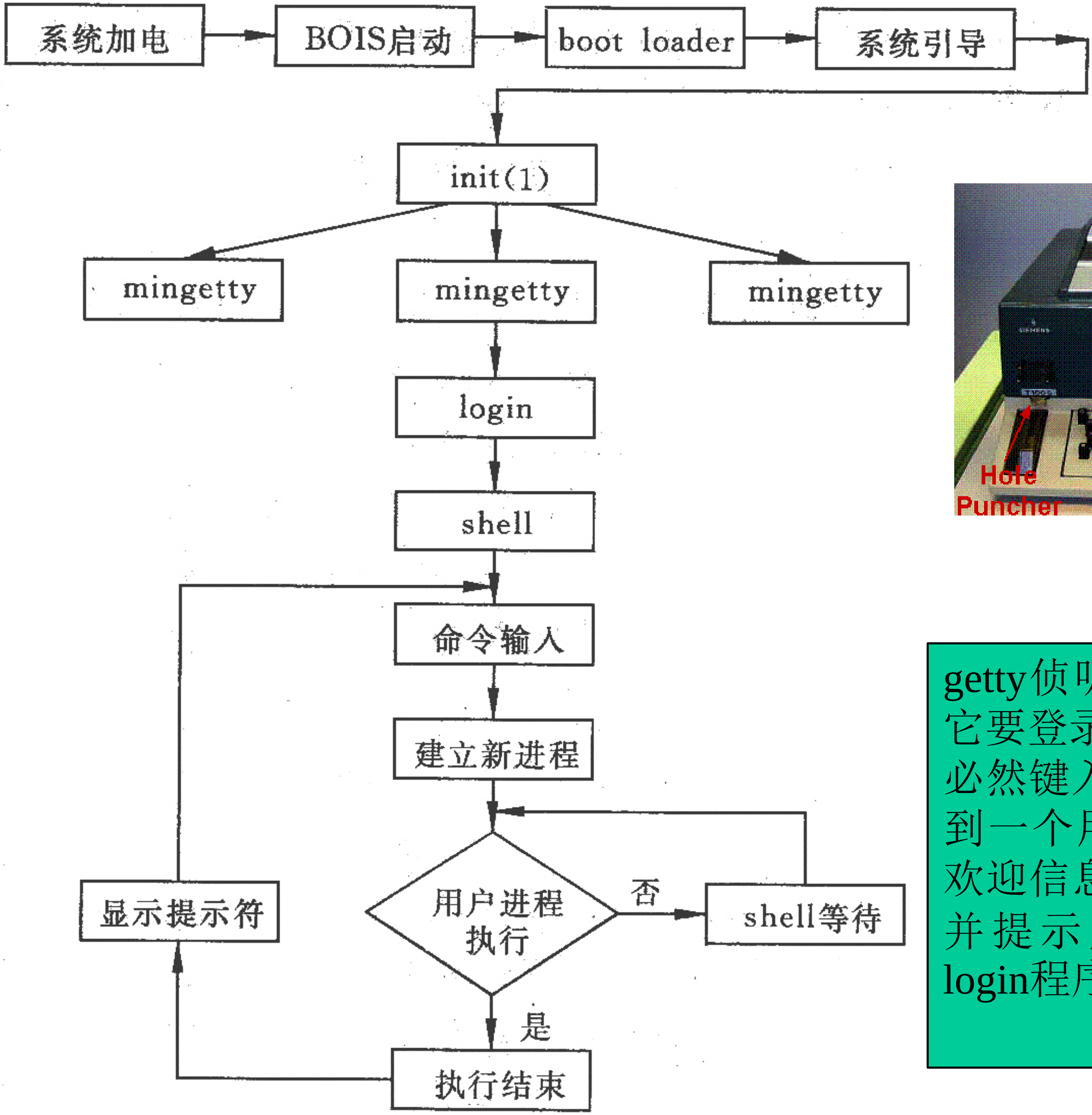
程序切换-〉运算速度不可再现-〉不能对时序做假设

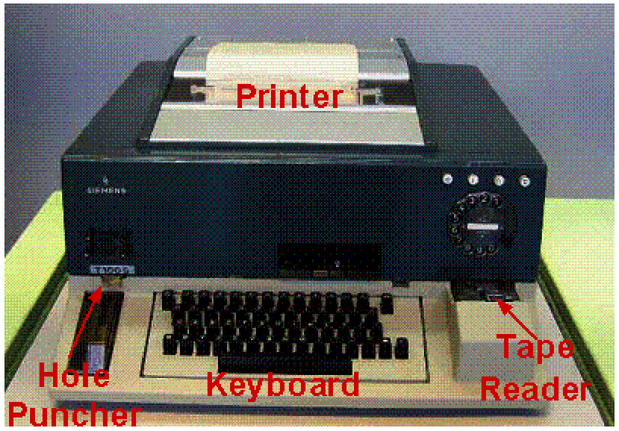
Process Creation

Events which cause process creation:

- System initialization.
- Execution of a process creation system call by a running process.
- A user request to create a new process.
- Initiation of a batch job.

相关的但是没有相互作用的进程，创建新进程是有效的。

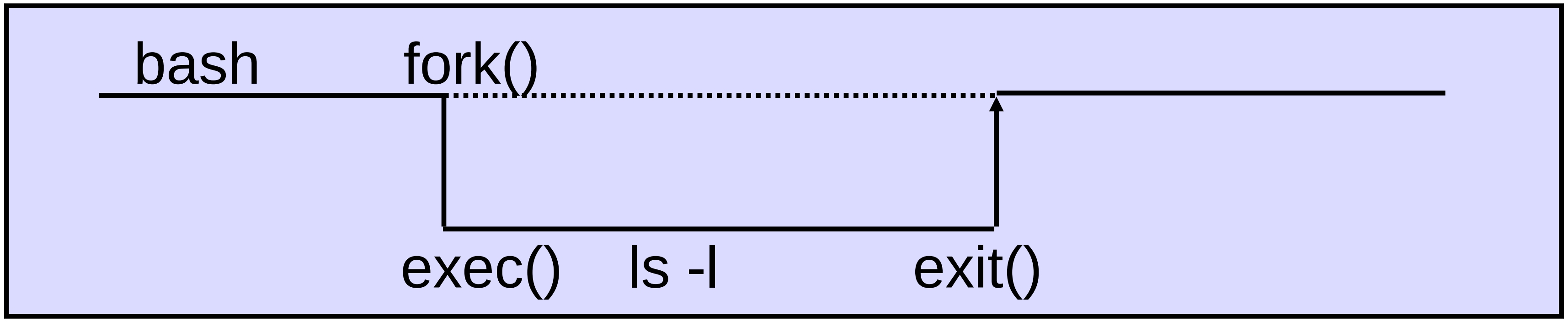




getty侦听终端等候用户告知它要登录 (这通常意味着用户必然键入些什么)。当它注意到一个用户，getty输出一个欢迎信息(存在/etc/issue中)，并提示用户名，最后运行login程序。

Starting a Process

All processes are started by other processes Parent/Child relationship.



$ ls –l

常用进程控制系统调用

- fork——进程创建
- exec——进程执行
- exit——进程终止
- wait——进程同步
- sleep——进程挂起
- system——执行shell命令
- getpid——返回进程PID
- getppid——返回父进程PID

进程的建立——fork系统调用

- 一个现存进程调用fork()函数是Linux创建一个新进程的手段

- fork()调用执行后，两个代码相同的父、子进程并发执行
  - 即子进程也是从fork()调用后的语句开始执行的
  - 如果调用成功，对父进程返回子进程的PID，对子进程返回0
  - 如果调用失败，返回-1
    - 失败主要原因：超出系统或者用户最大进程数的限制

- 返回值类型pid_t / int

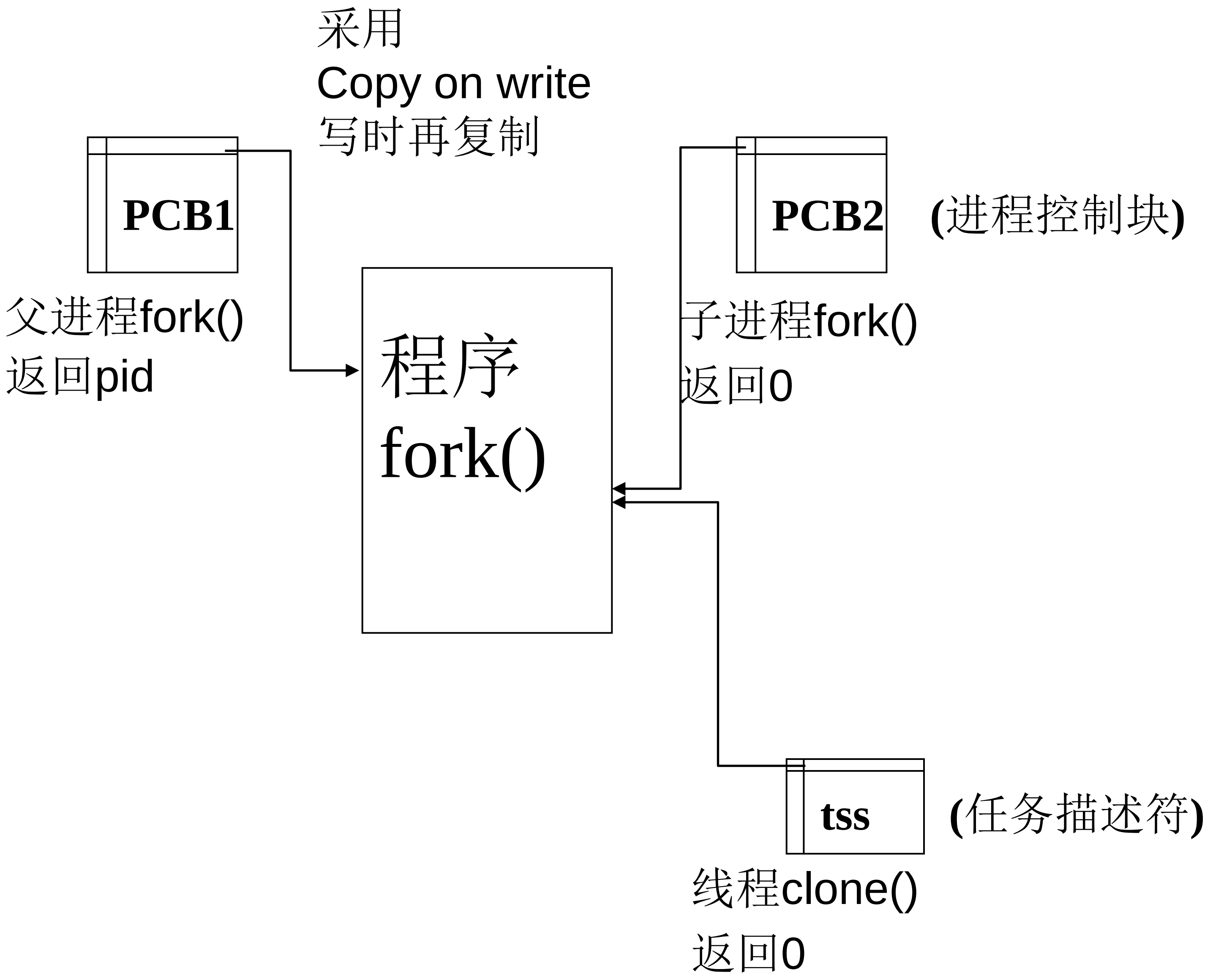
fork()的用处

- 一般来说fork()有两种用处：
  - 一个父进程希望复制自己，使父、子进程同时执行不同的代码段
    - 这在网络服务进程中是最常见的——父进程等待委托者的服务请求
      - 请求到达时，fork一个子进程处理请求，父进程继续等待下一个服务请求
  - 一个进程要执行一个不同的程序
    - 子进程从fork()返回后立即调用exec()

执行新程序——exec系统调用

- exec在这里指所有exec()一族的函数
  - execve()、 execl() 、execlp()、execle() 、execv()、execvp()等各种此类系统调用
- exec()系列负责读取可执行文件并将其载入执行——免于共享资源的复制（写时拷贝机制）
- 当exec系统调用成功时，不会将控制权返回给调用进程；失败时返回-1

进程的创建


采用
Copy on write
写时再复制
PCB1
父进程fork()
返回pid
程序
fork()
PCB2
(进程控制块)
子进程fork()
返回0
tss
(任务描述符)
线程clone()
返回0

通用寄存器组的定义

```
struct pt_regs {
        long ebx;
        long ecx;
        long edx;
        long esi;
        long edi;
        long ebp;
        long eax;
        int  xds;
        int  xes;
        long orig_eax;
        long eip;
        int  xcs;
        long eflags;
        long esp;
        int  xss;
};
```

当从**fork()**返回时，**eax**寄存器存放返回值

进程的同步——wait系统调用

- 一个进程可以通过系统调用wait，使之与执行的子进程终止实现同步

- wait系统调用将调用进程挂起，直到其任一个子进程暂停或退出，或者收到一个不能忽略的信号为止。

- 常用函数：
  - pid_t wait(0);
  - pid_t wait(int *status);
  - pid_t waitpid(pid_t pid, int *status, int options);
  - 其中status是接收子进程结束时返回的状态信息的地址

Monitoring Processes

- The ps command displays process status information

```
$ ps aux
USER    PID %CPU %MEM  VSZ RSS   TTY  STAT START TIME COMMAND
root      1  0.0  0.0 1336 436     ?  S    Jan1  0:05 init
root      2  0.0  0.0    0   0     ?  SW   Jan1  0:00 [keventd]
root      3  0.0  0.0    0   0     ?  SW   Jan1  0:05 [kapmd]
root      4  0.0  0.0    0   0     ?  SW   Jan1  0:05 [kswapd]
...
root 10248  0.0  0.1 2852 884 pts/2  R    13:47 0:00 ps aux
```

- **ps** supports a large number of options;
- you typically use **ps aux**:
  - **a** all processes attached to a terminal
  - **x** all other processes
  - **u** provides more columns

D 无法中断的休眠状态（ IO ）；
R 正在运行可中在队列中可过行的；
S 处于休眠状态；
T 停止或被追踪；
W 进入内存交换 （2.6后无效）；
X 死掉的进程 （基本很少見）；

A Simple Shell

```
#define TRUE 1

while (TRUE) {                                        /* repeat forever */
     type_prompt( );                                  /* display prompt on the screen */
     read_command(command, parameters);               /* read input from terminal */

     if (fork( ) != 0) {                              /* fork off child process */
         /* Parent code. */
         waitpid(−1, &status, 0);                     /* wait for child to exit */
     } else {
         /* Child code. */
         execve(command, parameters, 0);              /* execute command */
     }
}
```

Figure 1-19. A stripped-down shell.

Process Termination

Events which cause process termination:

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

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

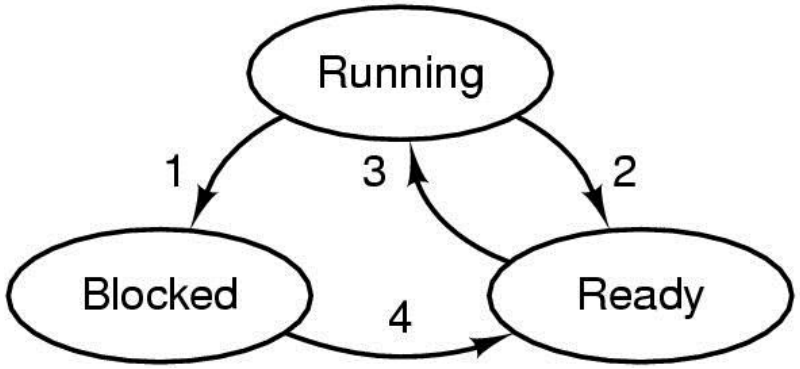
Viewing Process Hierarchy

- **pstree** shows process hierarchy

```
$ pstree
init-+-apmd
     |-atd
     |-crond
     |-gpm
     |-httpd---10*[httpd]
     |-inetd
     |-kattraction.kss
     |-kdm-+-X
     |     `-kdm---kwm-+-kbgndwm
     |                 |-kfm
     |                 |-kpanel
     |                 |-krootwm
     |                 |-kvt---bash---man---sh-+-gunzip
     |                 |                       `-less
     |                 `-startkde---autorun
     |-kflushd
```

Process States

例如：cat chapter1 chapter2 | grep tree



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

Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

Process States (2)

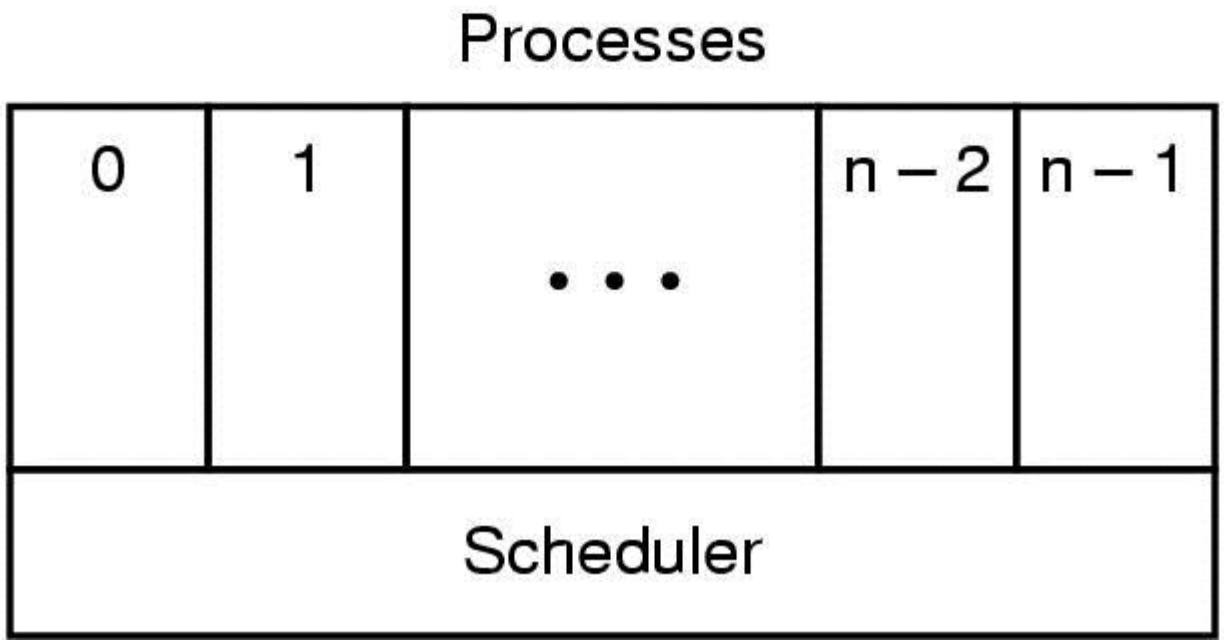
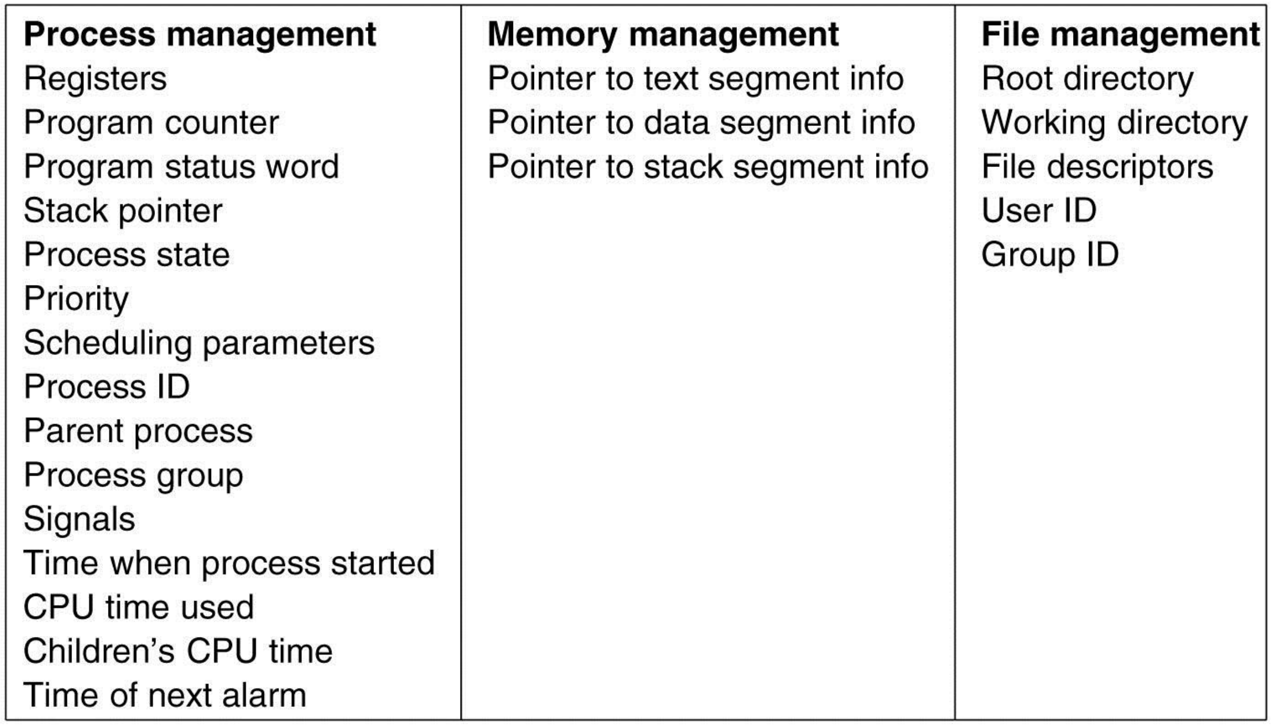


Figure 2-3. The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes.

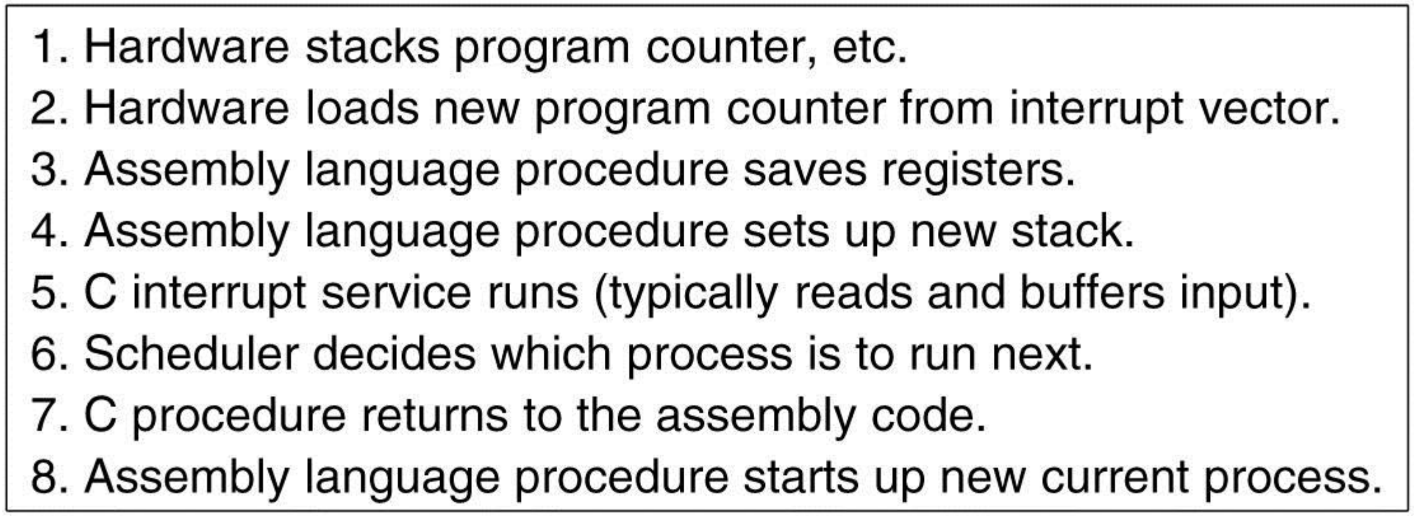
Implementation of Processes (1)



Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure 2-4. Some of the fields of a typical process table entry. (PCB)

Implementation of Processes (2)



1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.