

# Chapter 2

## Processes and Threads

2.1 进程

2.2 线程

2.3 进程通信 (IPC)

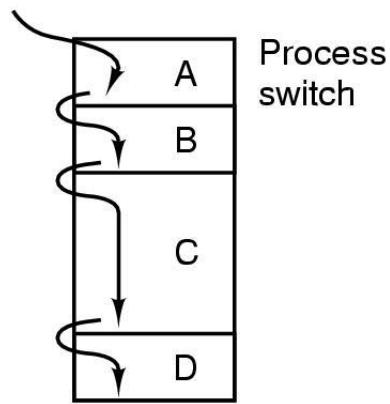
2.4 经典IPC问题

2.5 调度

# Processes

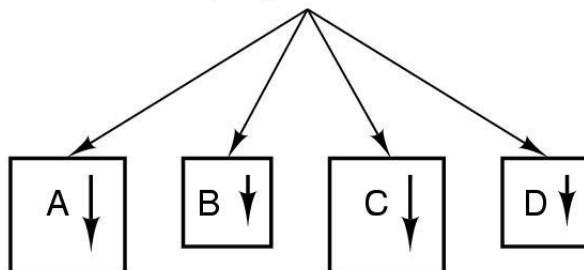
## The Process Model

One program counter

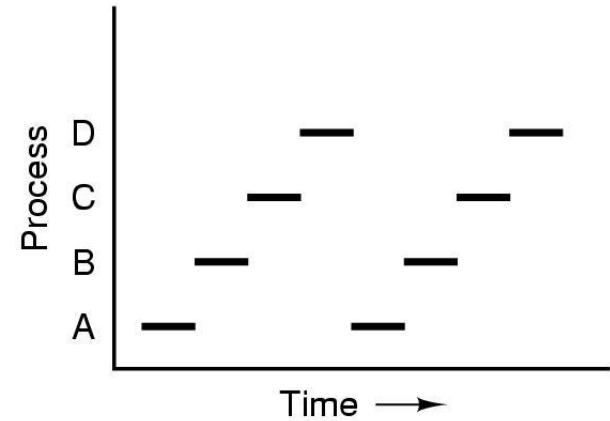


(a)

Four program counters



(b)



(c)

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes
- Only one program active at any instant

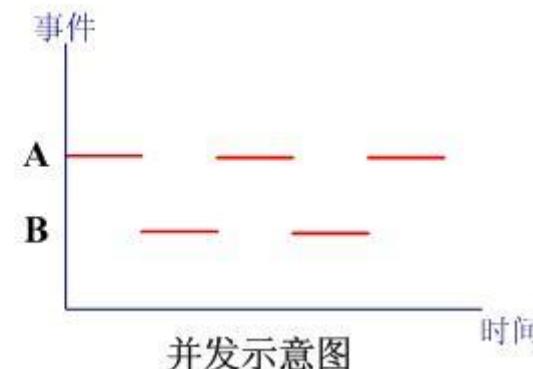
为了描述程序在并发执行时对系统资源的共享，需要一个描述程序执行时动态特征的概念，这就是进程。

# 多道程序设计与进程管理

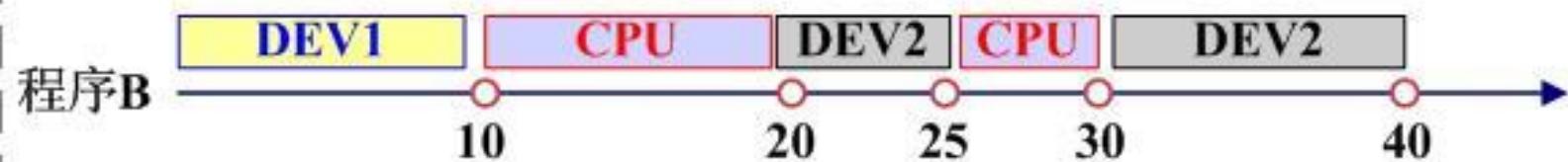
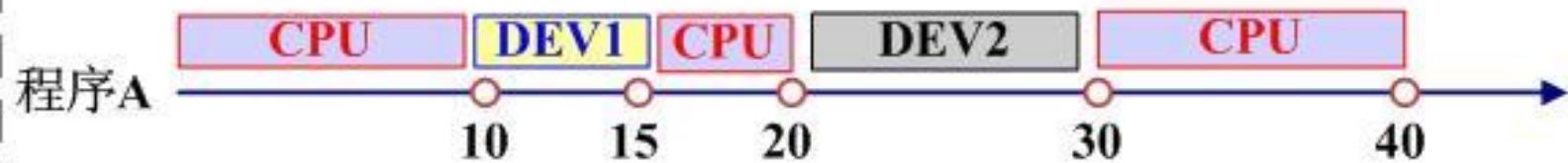
- 计算机中程序运行模式的发展历程
  - 顺序执行模式：单道程序独占CPU和其他资源
  - 并发执行模式：两个/多个程序共享CPU和其他资源
  - 多道程序设计：并发模式下的OS设计与实现
- 程序运行模式的特征
  - 顺序执行模式
    - 顺序性、封闭性、独占性、确定性（结果可再现性）
  - 并发执行模式
    - 并发性、间断性、共享性、不确定性、独立性和制约性
  - 问题1：并发与并行的区别是什么？
  - 问题2：如何计算并发环境下的计算机工作效率？

# 并行与并发的概念差别

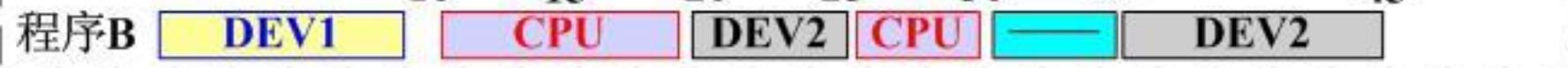
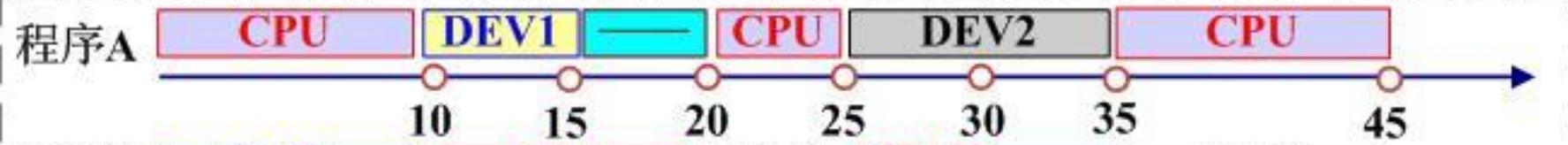
- 并行 (Parallel)
  - 同一时刻，两个事物均处于活动状态
  - 示例：CPU中的超流水线设计和超标量设计
- 并发 (Concurrency)
  - 宏观上存在并行特征，微观上存在顺序性
    - 同一时刻，只有一个事物处于活动状态
  - 示例：分时操作系统中多个程序的同时运行



# 并发所带来的效率提升



顺序执行模式



并发执行模式

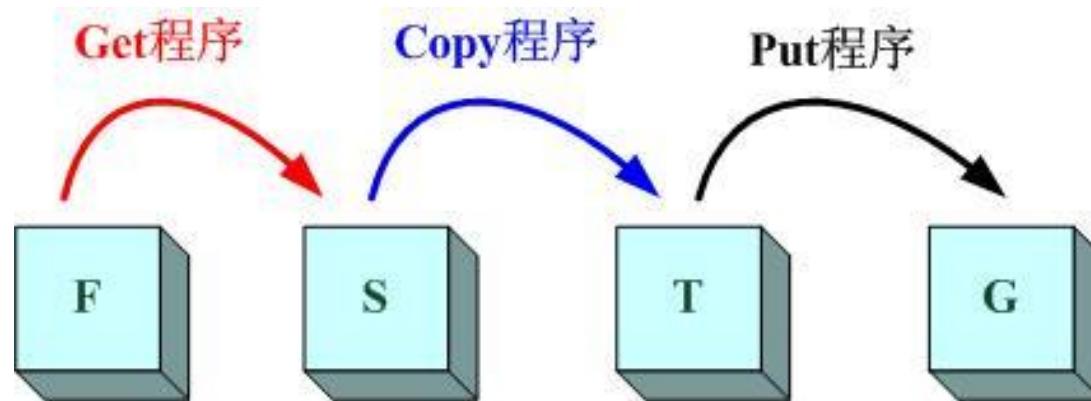
# 并发所带来的效率提升

- 顺序执行模式下的系统工作效率
  - 系统总运行时间: 80
  - CPU使用效率:  $\text{CPU占用时间} / \text{总时间} = 40/80 = 50\%$
  - DEV1使用效率:  $15 / 80 = 18.75\%$
  - DEV2使用效率:  $25 / 80 = 31.25\%$
- 并发执行模式下的系统工作效率
  - 系统总运行时间: 45
  - CPU使用效率:  $40 / 45 = 89\%$
  - DEV1使用效率:  $15 / 45 = 33\%$
  - DEV2使用效率:  $25 / 45 = 55.6\%$

# 并发所带来的设计难题

- 共享性带来的问题：如何调度或分配资源？
  - 最大限度的保证系统运行效率：谁首先获得资源？
  - 最大限度的保证系统运行安全：死锁情况的预防
- 间断性和不确定性带来的问题：如何保证互斥？
  - 互斥问题示例：见下页图
  - 经典IPC问题：进程管理的重点
- 间断性和独立性带来的问题：如何调度和保护？
  - 每个程序运行时都感觉不到间断：上下文切换
- 复杂应用带来的问题：如何进行通信？
  - 多个程序间可动态交换信息：进程通信机制

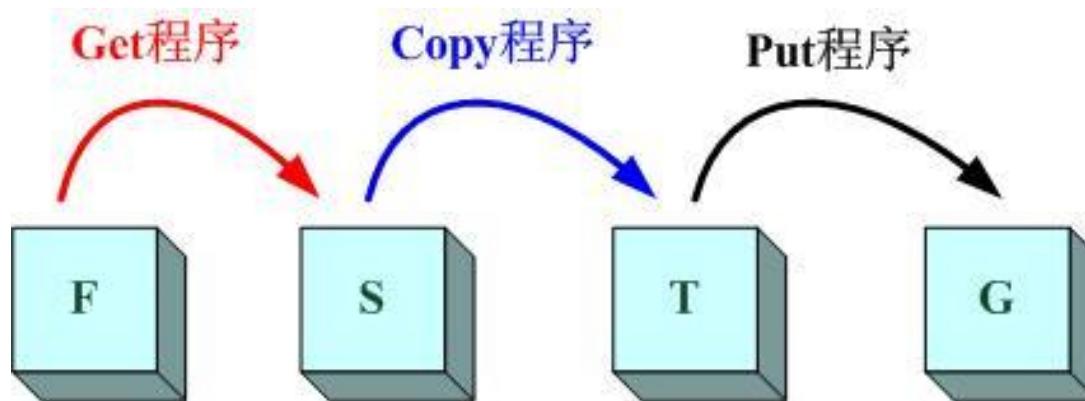
# 互斥问题



- Get、Copy、Put为三个并发的程序
- F、G为保存多条数据的缓冲区，S、T为临时缓冲区
- 三个程序顺序执行，可将F的值经过S、T保存到G中
- 正常情况下，不存在任何问题
- 但是程序运行顺序发生变化时，结果可能出错

# 互斥问题con.

初始状态	F	S	T	G	分析
程序运行顺序	3, 4, 5	2	2	1, 2	
G、C、P	4, 5	3	3	1, 2, 3	正确
G、P、C	4, 5	3	3	1, 2, 2	错误
C、P、G	4, 5	3	2	1, 2, 2	错误
C、G、P	4, 5	3	2	1, 2, 2	错误
P、C、G					
P、G、C					



# 进程概念

- 进程的核心思想

进程是某个程序在某个数据集合上的运行过程，它有程序、输入、输出和状态。

在分时操作系统中，单个**CPU**被若干进程共享，它使用某种调度算法决定何时停止一个进程的运行，转而为其他进程提供服务。

# 进程的特征

- 动态性：进程具有动态的地址空间（数量和内容），地址空间上包括：
  - 代码（指令执行和CPU状态的改变）
  - 数据（变量的生成和赋值）
  - 系统控制信息（进程控制块的生成和删除）
- 独立性：各进程的地址空间相互独立，除非采用进程间通信手段；
- 并发性、异步性：“虚拟”
- 结构化：代码段、数据段和核心段（在地址空间中）；程序文件中通常也划分了代码段和数据段，而核心段通常就是OS核心（由各个进程共享，包括各进程的PCB）

# 进程与程序的区别

- 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。通常进程不可在计算机之间迁移；而程序通常对应着文件、静态和可以复制。
- 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存。
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

# 进程控制块 (PCB, process control block)

进程控制块是由OS维护的用来记录进程相关信息的一块内存。

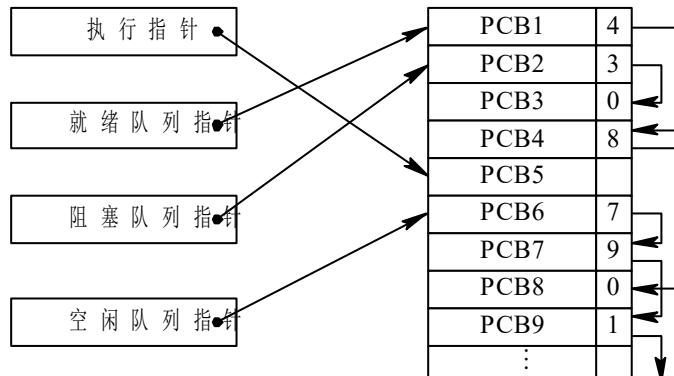
- 每个进程在OS中的登记表项（可能有总数目限制），OS据此对进程进行控制和管理（PCB中的内容会动态改变），不同OS则不同
- 处于核心段，通常不能由应用程序自身的代码来直接访问，而要通过系统调用，或通过UNIX中的进程文件系统(/proc)直接访问进程映象(image)。文件名为进程标识（如：00316），权限为创建者可读写。

# 进程控制块的内容

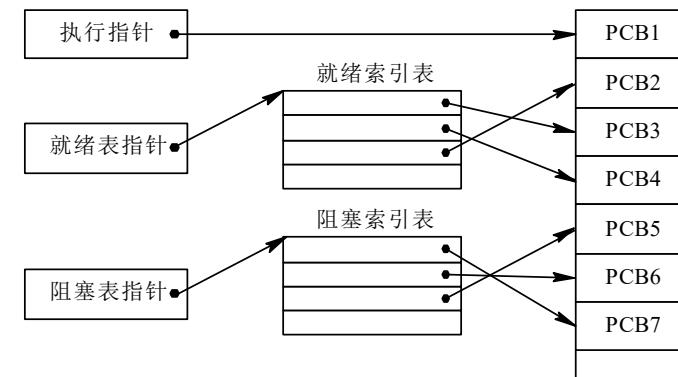
- 进程描述信息：
  - 进程标识符(process ID), 唯一, 通常是一个整数;
  - 进程名, 通常基于可执行文件名 (不唯一) ;
  - 用户标识符(user ID); 进程组关系(process group)
- 进程控制信息：
  - 当前状态;
  - 优先级(priority);
  - 代码执行入口地址;
  - 程序的外存地址;
  - 运行统计信息 (执行时间、页面调度) ;
  - 进程间同步和通信; 阻塞原因
- 资源占用信息：虚拟地址空间的现状、打开文件列表
- CPU现场保护结构：寄存器值（通用、程序计数器PC、状态PSW，地址包括栈指针）

# PCB的组织方式

- **链表:** 同一状态的进程其PCB成一链表，多个状态对应多个不同的链表
  - 各状态的进程形成不同的链表：就绪链表、阻塞链表
- **索引表:** 同一状态的进程归入一个index表（由index指向PCB），多个状态对应多个不同的index表
  - 各状态的进行形成不同的索引表：就绪索引表、阻塞索引表



PCB链接队列示意图



按索引方式组织PCB

# 进程上下文

进程上下文是对进程执行活动全过程的静态描述。进程上下文由进程的用户地址空间内容、硬件寄存器内容及与该进程相关的核心数据结构组成。

- **用户级上下文：** 进程的用户地址空间（包括用户栈各层次），包括用户正文段、用户数据段和用户栈；
- **寄存器级上下文：** 程序寄存器、处理机状态寄存器、栈指针、通用寄存器的值；
- **系统级上下文：**
  - 静态部分（PCB和资源表格）
  - 动态部分：核心栈（核心过程的栈结构，不同进程在调用相同核心过程时有不同核心栈）

# 进程的创建(Process Creation)

引起创建进程的事件

1. System initialization
2. Execution of a process creation system call by a running process
3. User request to create a new process
4. Initiation of a batch job

# Process Creation System Call

- Unix: fork() ---only one system call  
use execve() to change its memory image.
  - fork 系统调用：创建一个子（新）进程。精确复制父进程
  - exec系统调用：在 fork 之后执行，用新的代码替换原父进程的代码。
- Windows: CreateProcess()
- 进程创建要完成的工作
  - 申请空白PCB
  - 为新进程分配资源：为程序、数据、用户栈分配空间
  - 初始化进程控制快：初始化标识信息、处理机状态（PC、SP）、处理机控制信息（进程状态、优先级）
  - 将新进程插入就绪队列

# 进程的终止

## Conditions which terminate processes

### 1. Normal exit (voluntary)

( Unix: exit(), Windows: ExitProcess() )

### 2. Error exit (voluntary)

### 3. Fatal error (involuntary)

### 4. Killed by another process (involuntary)

( Unix: kill(), Windows: TerminateProcess() )

- 越界错误
- 保护错——试图访问不允许访问的资源或文件，或者以不适当方式访问
- 非法指令
- 特权指令错——用户程序试图执行只允许OS执行的指令
- 运行超时
- 等待超时
- 算术运算错——被0除
- I/O故障

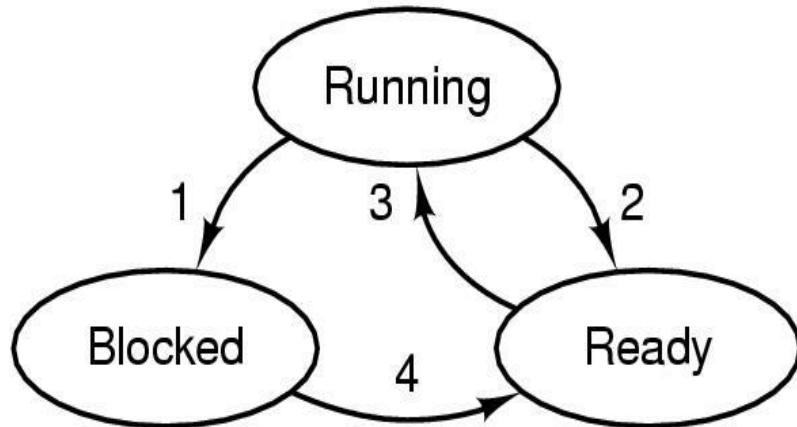
# Process Hierarchies

- 父进程创建子进程， 子进程还可创建子进程，从而形成进程树。
- Forms a hierarchy
  - UNIX calls this a "process group"
- Windows has no concept of process hierarchy
  - all processes are created equal

# 父子进程关系

- 资源继承共享
  - 父子进程共享所有的资源.
  - 子进程共享父进程资源的子集.
  - 父进程和子进程之间不共享资源.
- 执行关系:
  - 父进程和子进程并发执行.
  - 父进程等待子进程执行结束.
- 地址空间
  - 子进程是父进程的复制（克隆）.
  - 子进程将自己代码装入其中。

# Process States (1)



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

- Possible process states
  - running:运行态，正在占用CPU运行程序
  - blocked:阻塞态，等待外部事件发生，无法占用CPU
  - ready:就绪态，可运行，但其他进程正在占用CPU，所有被暂时挂起
- Transitions between states shown

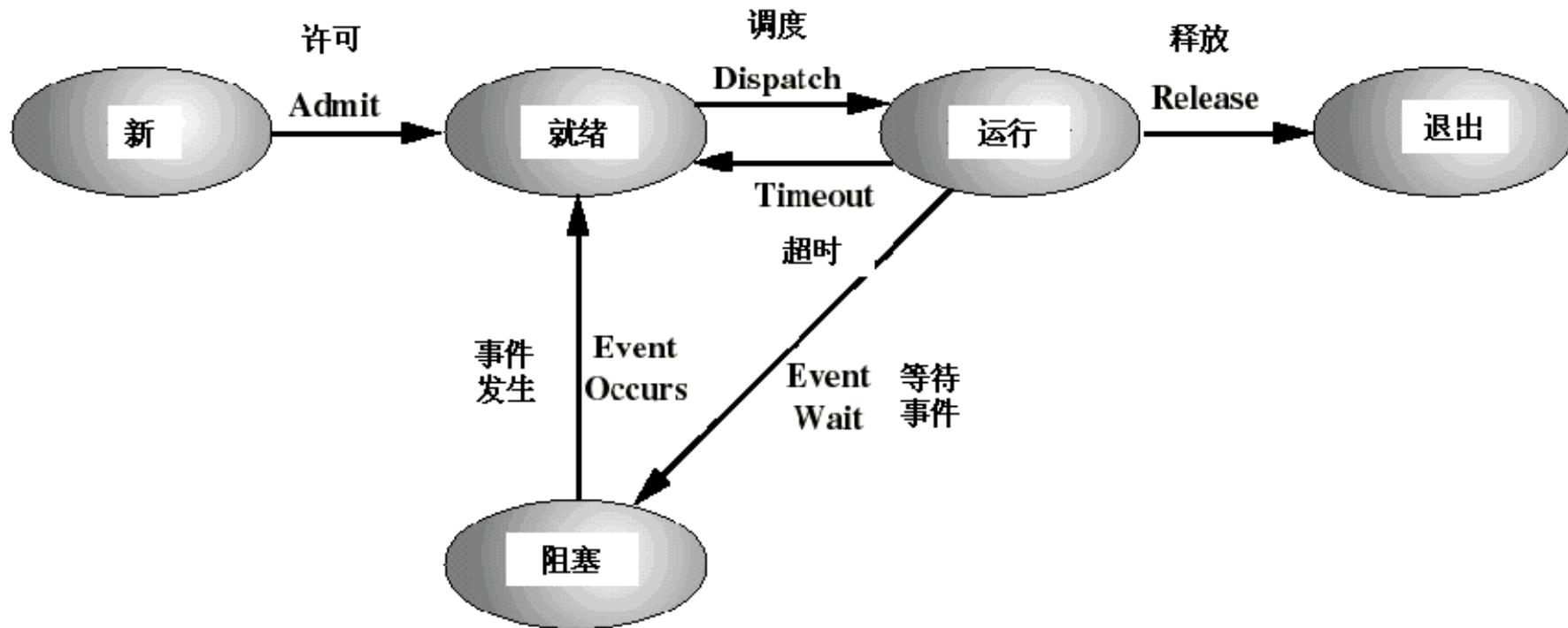
# Problem

- 问题1：为什么不能从阻塞态变为运行态呢？
- 问题2：为什么不能从就绪态变为阻塞态呢？
- 提示：
  - 三种状态的变换体现了OS的资源管理作用
  - 进程的核心思想在于运行——CPU资源的分配

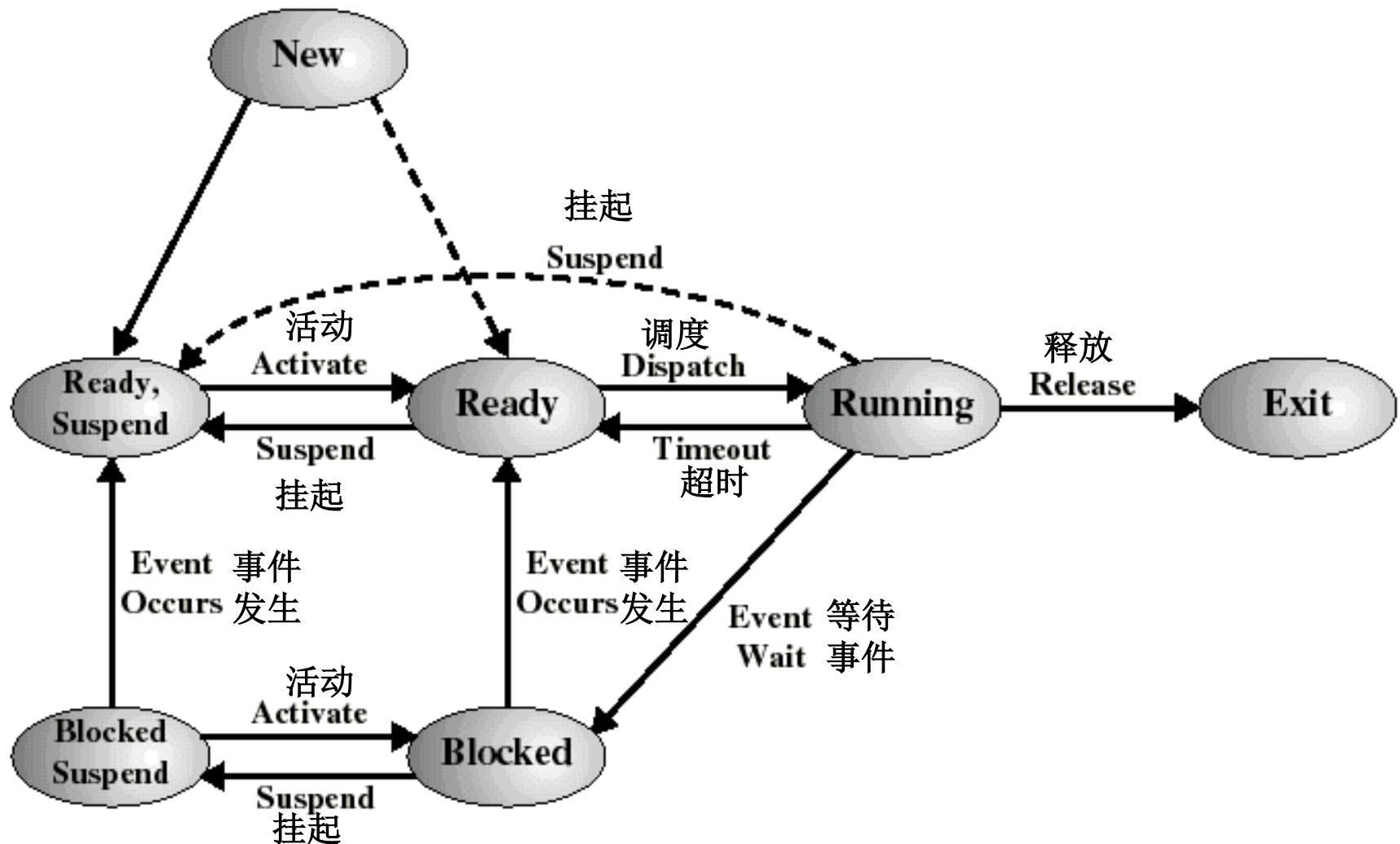
# 进程的复杂状态

- 其他复杂的进程状态
  - 创建、中止、挂起三种状态
- 五种状态模型
  - 增加了“创建”、“退出”两种状态
- 七种状态模型
  - 针对进程的存在位置（内存或者外存），进一步增加“阻塞”  
“挂起”和“就绪挂起”两种状态
- 进程状态模型的重要性
  - OS管理CPU资源的核心运行机制
  - 直接关系到进程数据结构与相关算法的设计

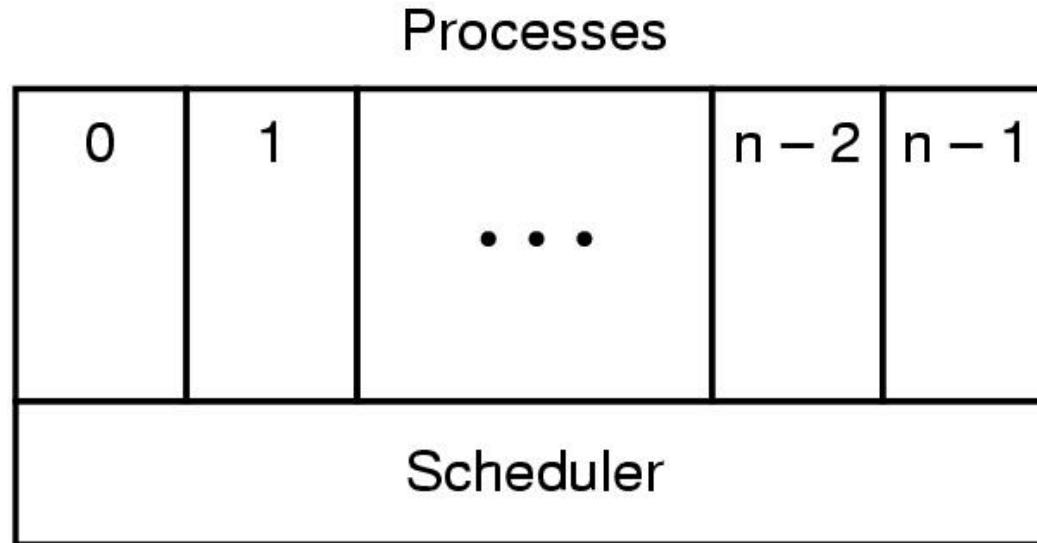
# 进程的复杂状态con.



# 进程的复杂状态con.



# Process States (2)



- Lowest layer of process-structured OS
  - handles interrupts, scheduling
- Above that layer are sequential processes

# 进程管理

- 单个进程的管理
  - 进程的创建、删除：静态与动态概念的差别。
- 多个进程的管理
  - 树状进程集合：父进程和子进程
  - 进程的调度：保持CPU效率的关键
- 进程之间的通信
  - 最简单：两个进程之间如何传递信息？
  - 中级：如何保持进程之间的互斥？
  - 高级：如何维系进程之间的依赖关系？

# Implementation of Processes (1)

<b>Process management</b>	<b>Memory management</b>	<b>File management</b>
Registers Program counter Program status word Stack pointer Process state 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	Pointer to text segment Pointer to data segment Pointer to stack segment	Root directory Working directory File descriptors User ID Group ID

Fields of a process table entry

# 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.

Skeleton of what lowest level of OS does when an interrupt occurs

# 进程小结

- 概念
  - CPU资源的管理载体—OS通过进程实现对CPU的管理
  - 核心思想—动态概念—进程与程序的区别
  - 进程实现—生存周期—状态的变迁
- 进程管理
  - 单个进程—进程调度—进程通信
  - 进程序列—进程树—线程
  - 数据传递—共享与互斥—依赖关系维护
- 术语
  - Cocurrency & Parallel & MultiProgramming
  - Program & Process & Thread
  - Program & Process: 静态和动态的思想
  - MultiProgram ming: 并发与互斥的思想

# 线程

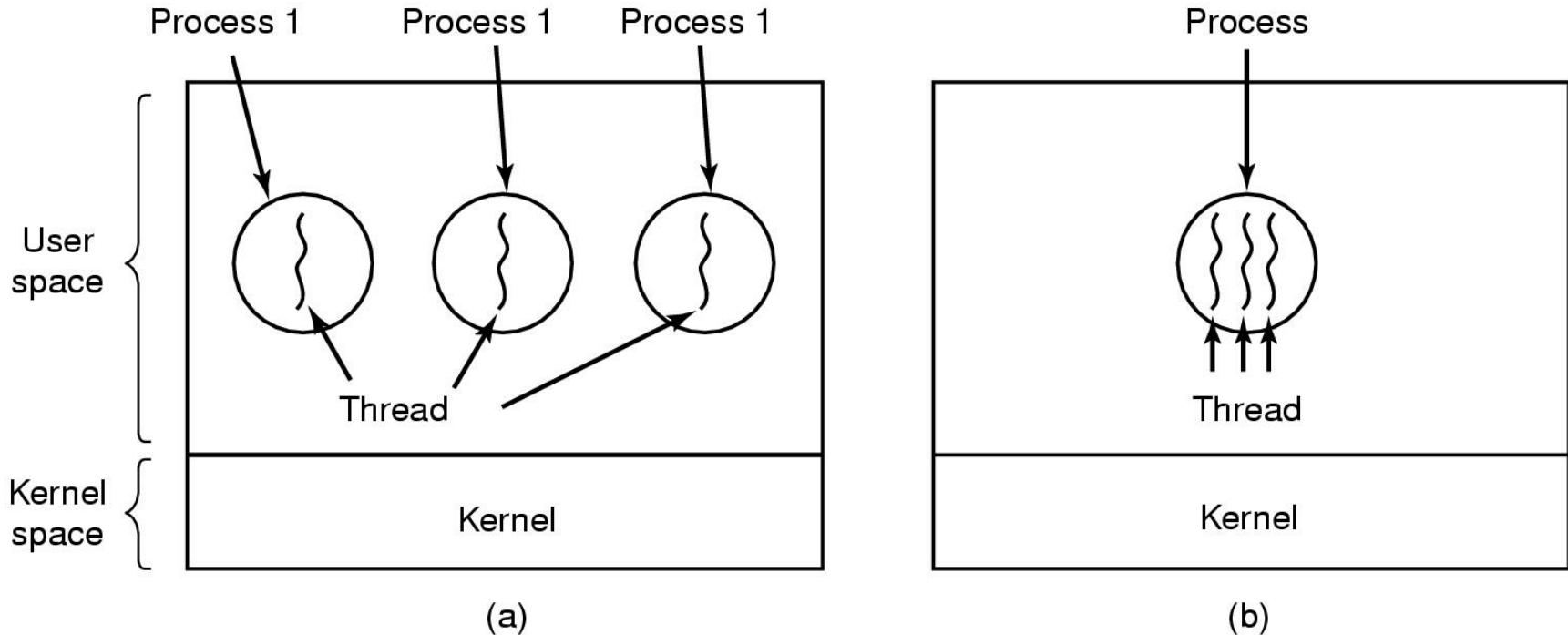
- 在OS中，进程的引入提高了计算机资源的利用率。但在进一步提高进程的并发性时，人们发现进程切换开销的比重越来越大，同时进程通信的效率也受到限制。
- 引入线程的目的是用它来提高系统内程序的并发程度，提高系统效率，增大系统作业的吞吐量。
- 进程：资源分配单位（存储器、文件）和CPU调度（分派）单位。又称为"任务(task)"
- 线程：作为CPU调度单位，而进程只作为其他资源分配单位。
  - 只拥有必不可少的资源，如：线程状态、寄存器上下文和栈
  - 同样具有就绪、阻塞和执行三种基本状态

# 线程

- 引入线程的目的是简化线程间的通信，以小的开销来提高进程内的并发程度。
- 线程的优点：减小并发执行的时间和空间开销（线程的创建、退出和调度），因此容许在系统中建立更多的线程来提高并发程度。
  - 线程的创建时间比进程短；
  - 线程的终止时间比进程短；
  - 同进程内的线程切换时间比进程短；
  - 由于同进程内线程间共享内存和文件资源，可直接进行不通过内核的通信

# Threads

## The Thread Model (1)



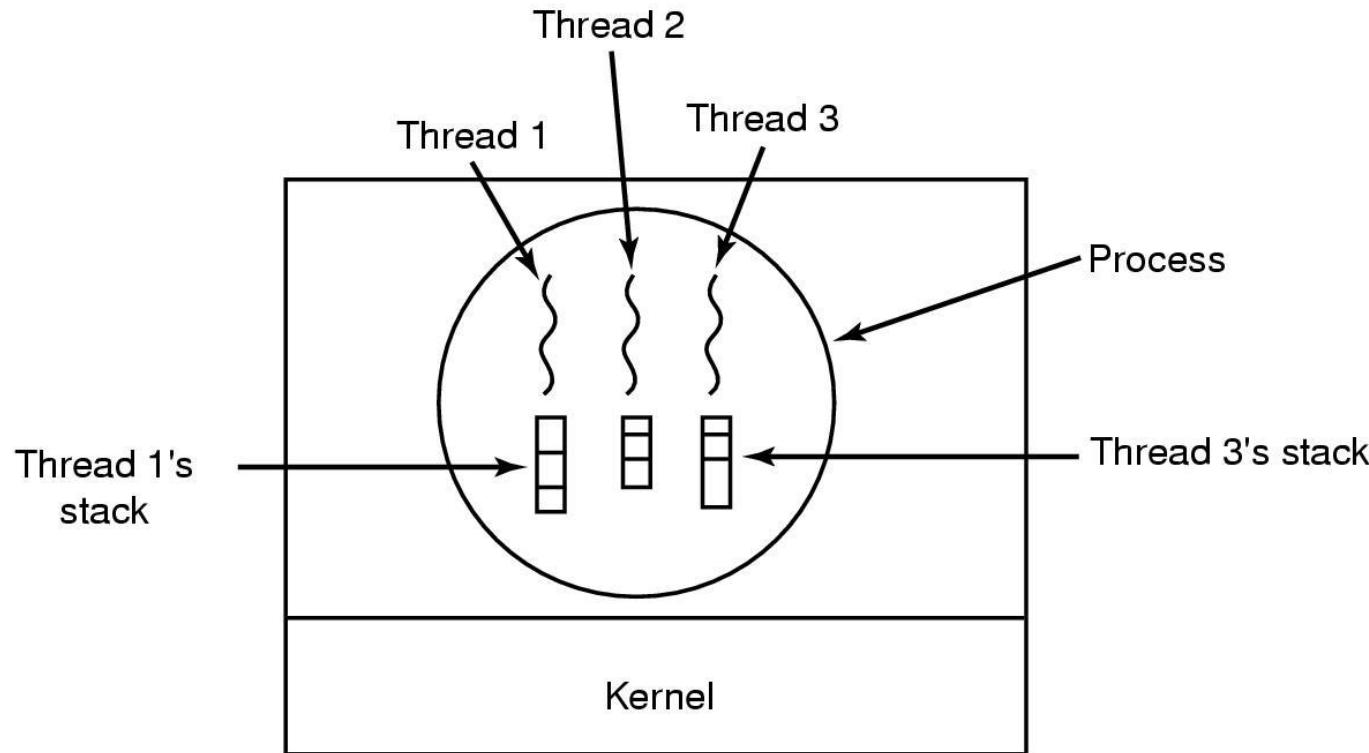
- (a) Three processes each with one thread
- (b) One process with three threads

# The Thread Model (2)

<b>Per process items</b>	<b>Per thread items</b>
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- Items shared by all threads in a process
- Items private to each thread

# The Thread Model (3)



Each thread has its own stack

Every thread can access every memory address within the process address space. There is no protection between threads because 1) it is impossible, and 2) it should not be necessary.

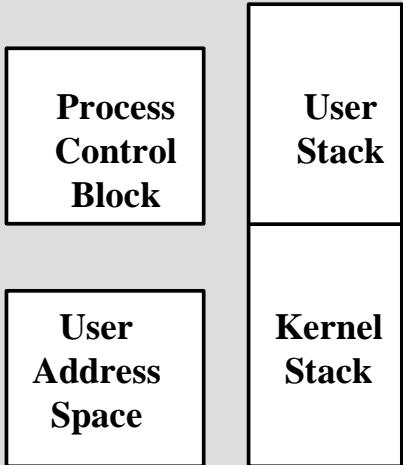
# Thread Call

- create new threads: `thread_create()`
- exit thread: `thread_exit()`
- one thread wait for a specific thread to exit: `thread_wait()`
- allow a thread to voluntarily give up the CPU to let another thread run: `thread_yield()`

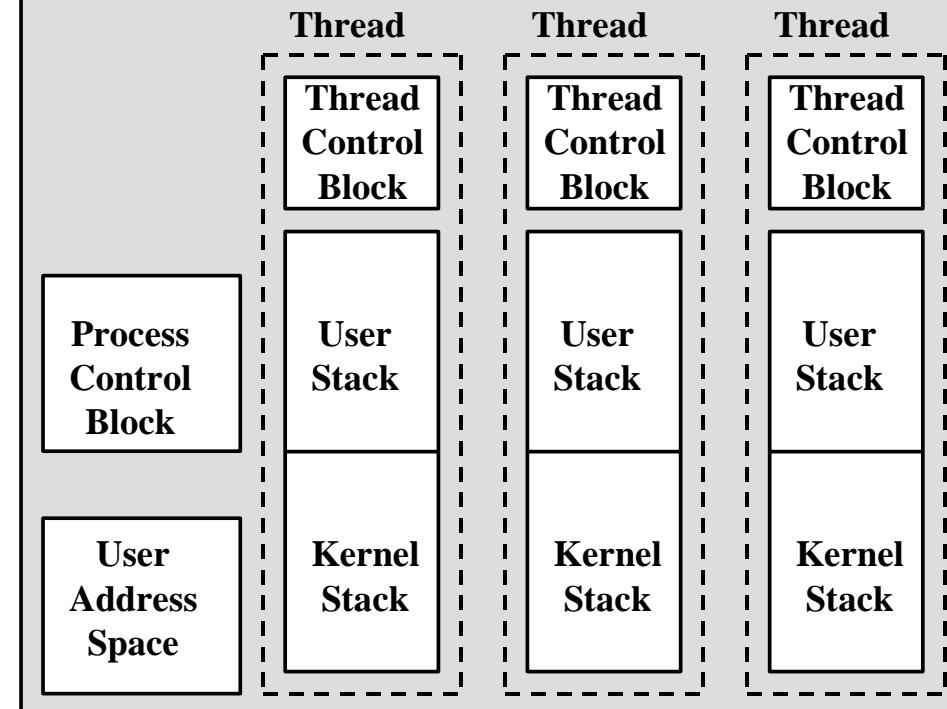
# 线程与进程的比较

- 1. 在支持多线程处理中，进程是资源分配的单位（如存储器、打开文件等，不含处理机）。线程是处理机调度单位，但不是资源的分配单位。
- 2. 线程只拥有必不可少的资源，如：线程状态、寄存器上下文和栈
- 3. 线程同样具有就绪、阻塞和执行三种基本状态
- 4. 线程减少了并发执行的时间和空间开销（线程的创建、退出和调度），因此容许在系统中建立更多的线程来提高并发程度。
- 5. 进程间的地址空间和其他资源（如打开文件）相互独立，同一进程的各线程间共享进程的所有资源——某进程内的线程在其他进程不可见
- 6. 通信：进程间通信复杂。线程间可以直接读写进程数据段（如全局变量）来进行通信，但需要进程同步和互斥手段的辅助，以保证数据的一致性。

## Single-Threaded Process Model

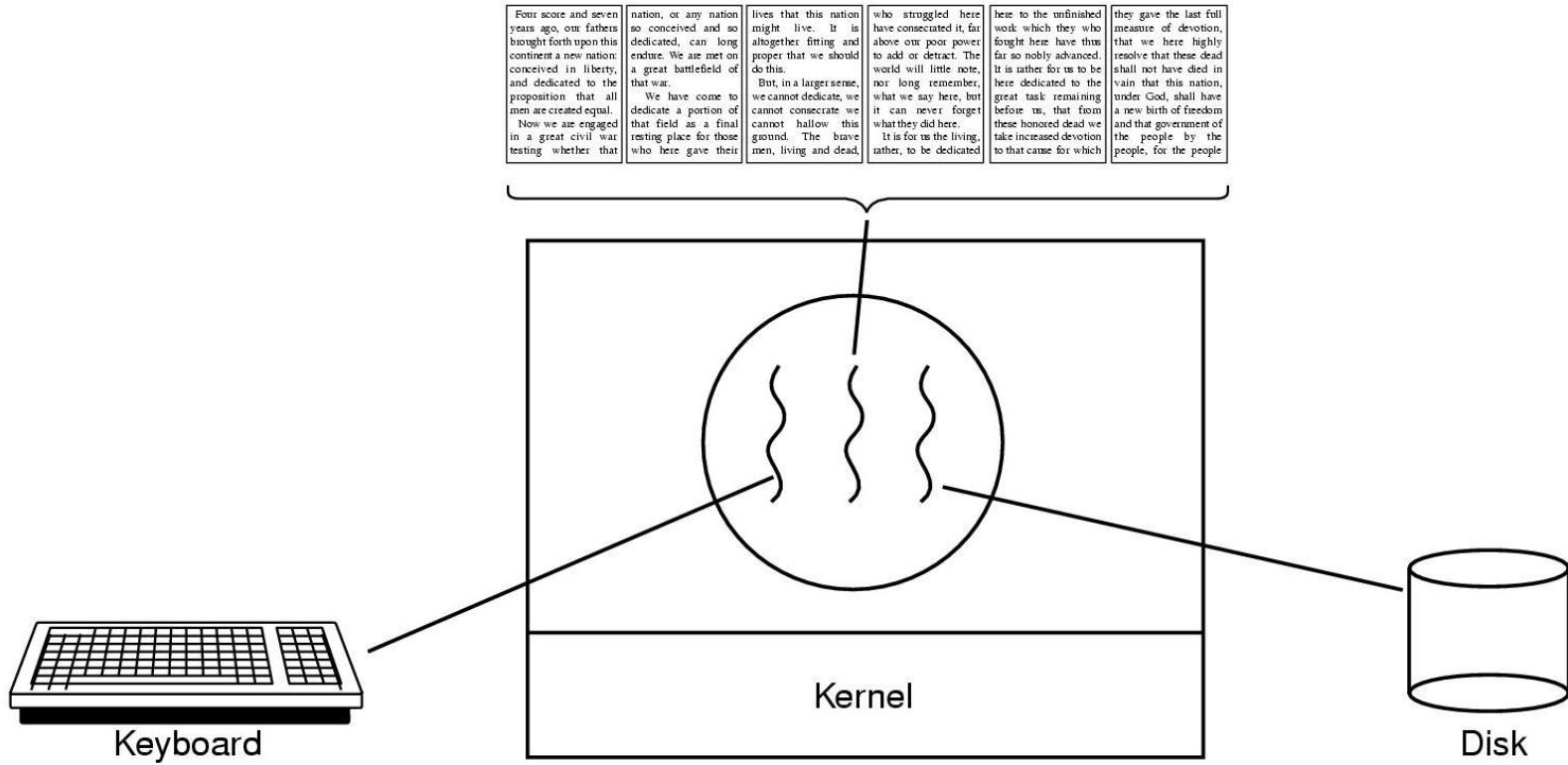


## Multithreaded Process Model



线程切换和进程切换

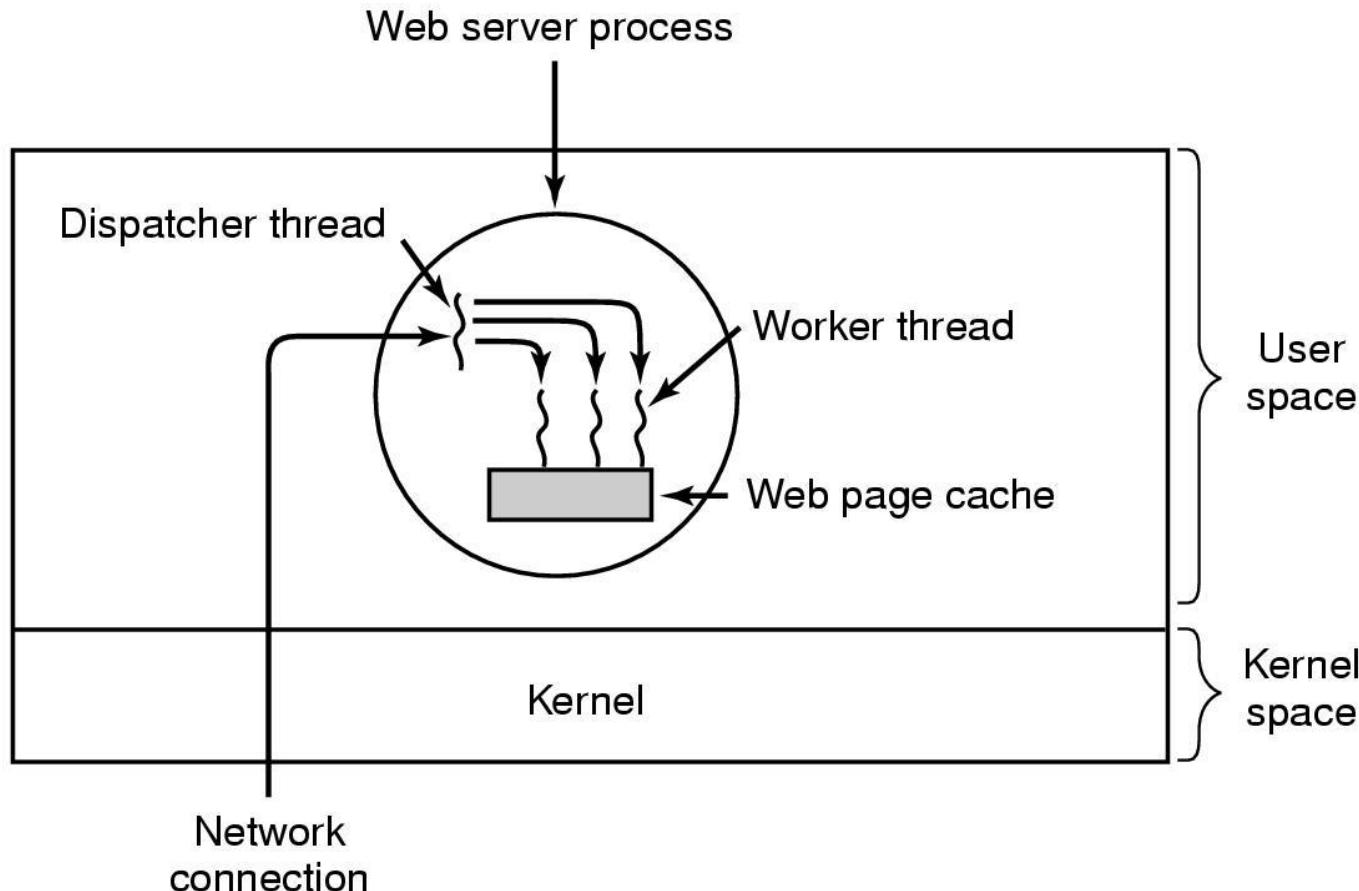
# Thread Usage (1)



## A word processor with three threads

threads: one interacts with user, another handles reformatting in the background, a third thread automatically save the entire file to disk during a specific time (handle the disk backups without interfering with the other two).

# Thread Usage (2)



A multithreaded Web server

# Thread Usage (3)

```
while (TRUE) {  
    get_next_request(&buf);  
    handoff_work(&buf);  
}
```

(a)

```
while (TRUE) {  
    wait_for_work(&buf)  
    look_for_page_in_cache(&buf, &page);  
    if (page_not_in_cache(&page))  
        read_page_from_disk(&buf, &page);  
    return_page(&page);  
}
```

(b)

- Rough outline of code for previous slide
  - (a) Dispatcher thread
  - (b) Worker thread

# Thread Usage (4)

<b>Model</b>	<b>Characteristics</b>
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

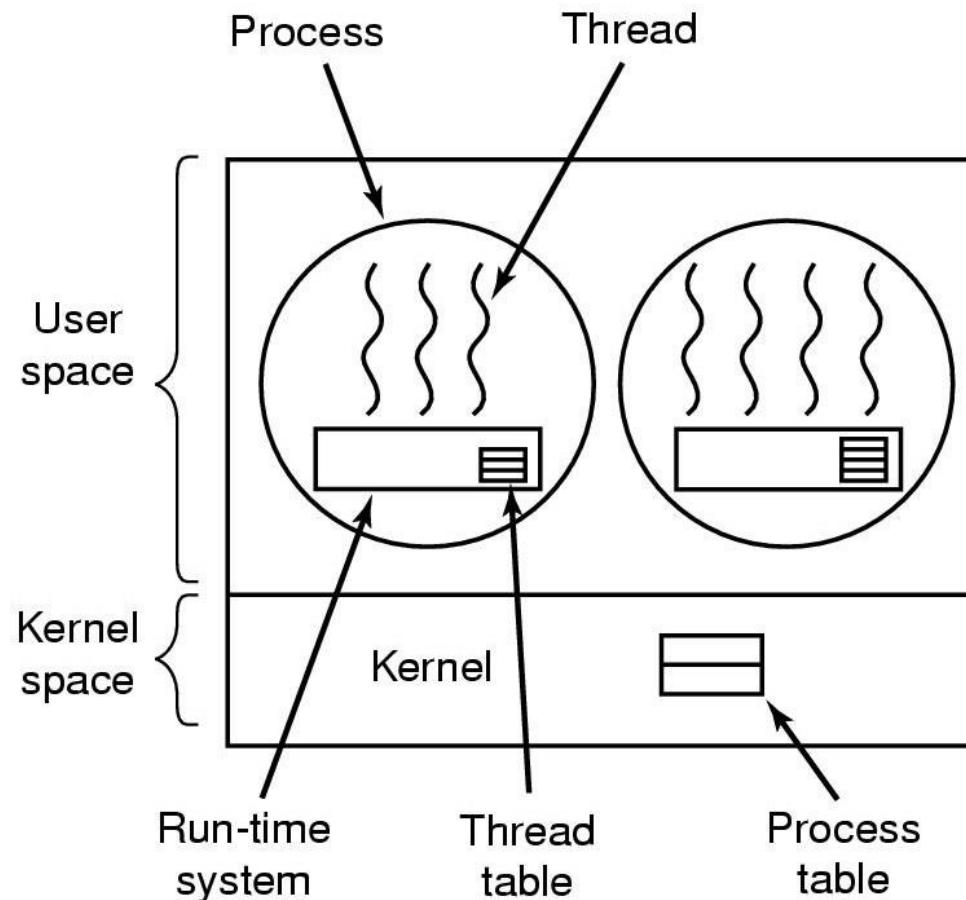
Three ways to construct a server

# 用户线程(user-level thread)

不依赖于OS核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。如：数据库系统informix，图形处理Aldus PageMaker。调度由应用软件内部进行，通常采用非抢先式和更简单的规则，也无需用户态/核心态切换，所以速度特别快。一个线程发起系统调用而阻塞，则整个进程在等待。时间片分配给进程，多线程则每个线程就慢。

- 用户线程的维护由应用进程完成；
- 内核不了解用户线程的存在；
- 用户线程切换不需要内核特权；
- 用户线程调度算法可针对应用优化；

# Implementing Threads in User Space



A user-level threads package

# Implementing Threads in User Space: Advantage

- A user-level threads package can be implemented on an operating system that does not support threads.
- Doing thread switching is faster than trapping to the kernel, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.
- Allow each process to have its own customized scheduling algorithm.
- Scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

# Implementing Threads in User Space: Problems

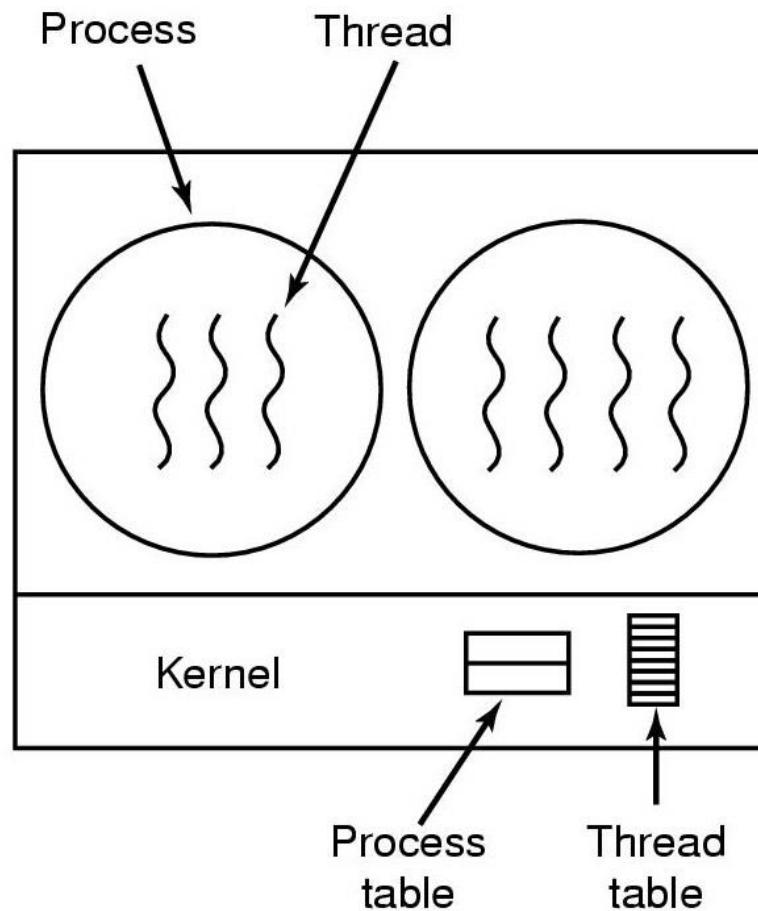
- the problem of how blocking system calls are implemented:  
1) one blocking call may stop all the threads, 2)  
nonblocking system call changing require changes to the  
OS, 3) to tell in advance if a call will block: do the call if it  
is safe, if the call will block, the call is not made. Instead,  
another thread is run.
- the problem of page faults: block even though other  
runnable threads.
- Within a single process, there are no clock interrupts,  
Unless a thread enters the run-time system of its own free  
will, the scheduler will never get a chance.
- Programmers generally want threads in applications where  
the threads block often. Block vs computing.

# 内核线程(kernel-level thread)

依赖于OS核心，由内核的内部需求进行创建和撤销，用来执行一个指定的函数。Windows NT和OS/2支持内核线程；

- 内核维护进程和线程的上下文信息；
- 线程切换由内核完成；
- 一个线程发起系统调用而阻塞，不会影响其他线程的运行。
- 时间片分配给线程，所以多线程的进程获得更多CPU时间。

# Implementing Threads in the Kernel



A threads package managed by the kernel

# Implementing Threads in the Kernel

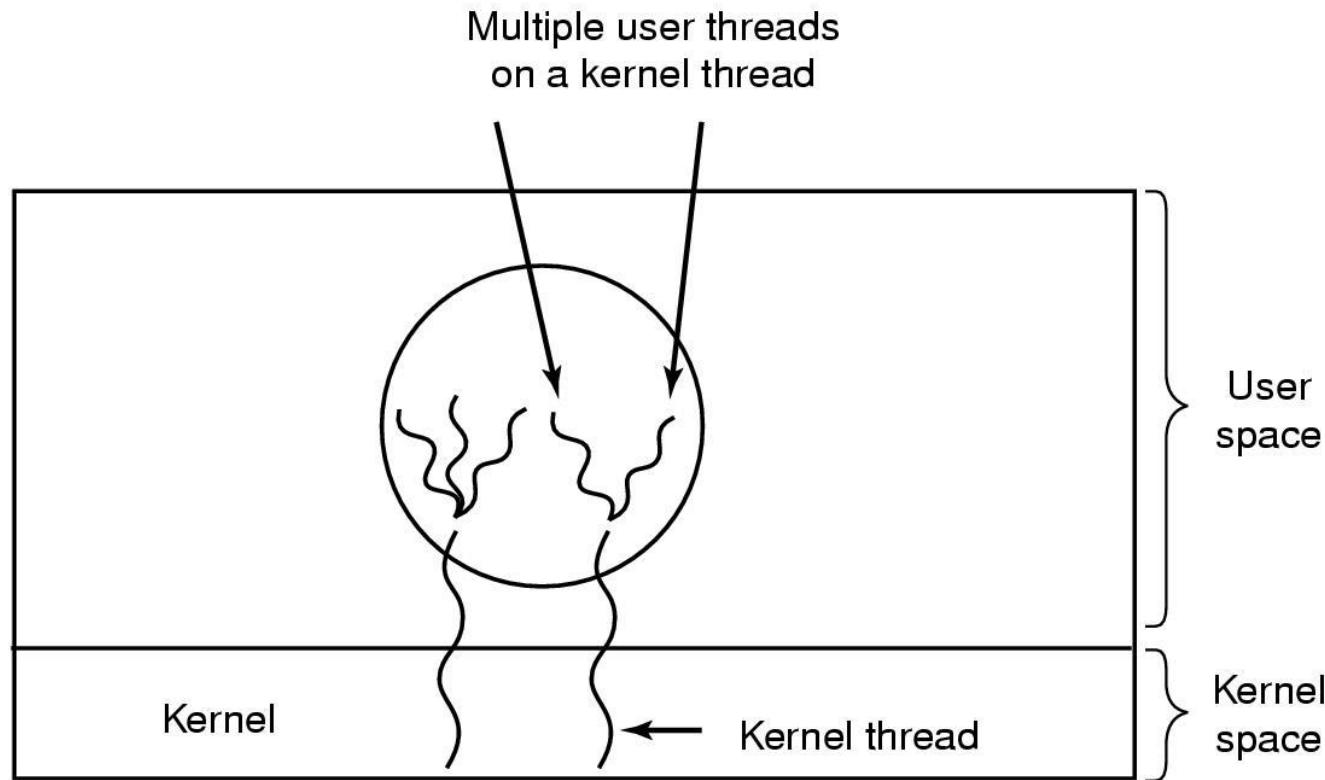
## Advantage

- No run-time system is needed
- Kernel threads do not require any new, nonblocking system calls.
- The kernel can easily check which thread (in the same process) when page fault.

## Disadvantage

- The cost of a system call is substantial, so if thread operations (creation, termination, etc.) are common, much more overhead will be incurred.

# Hybrid Implementations

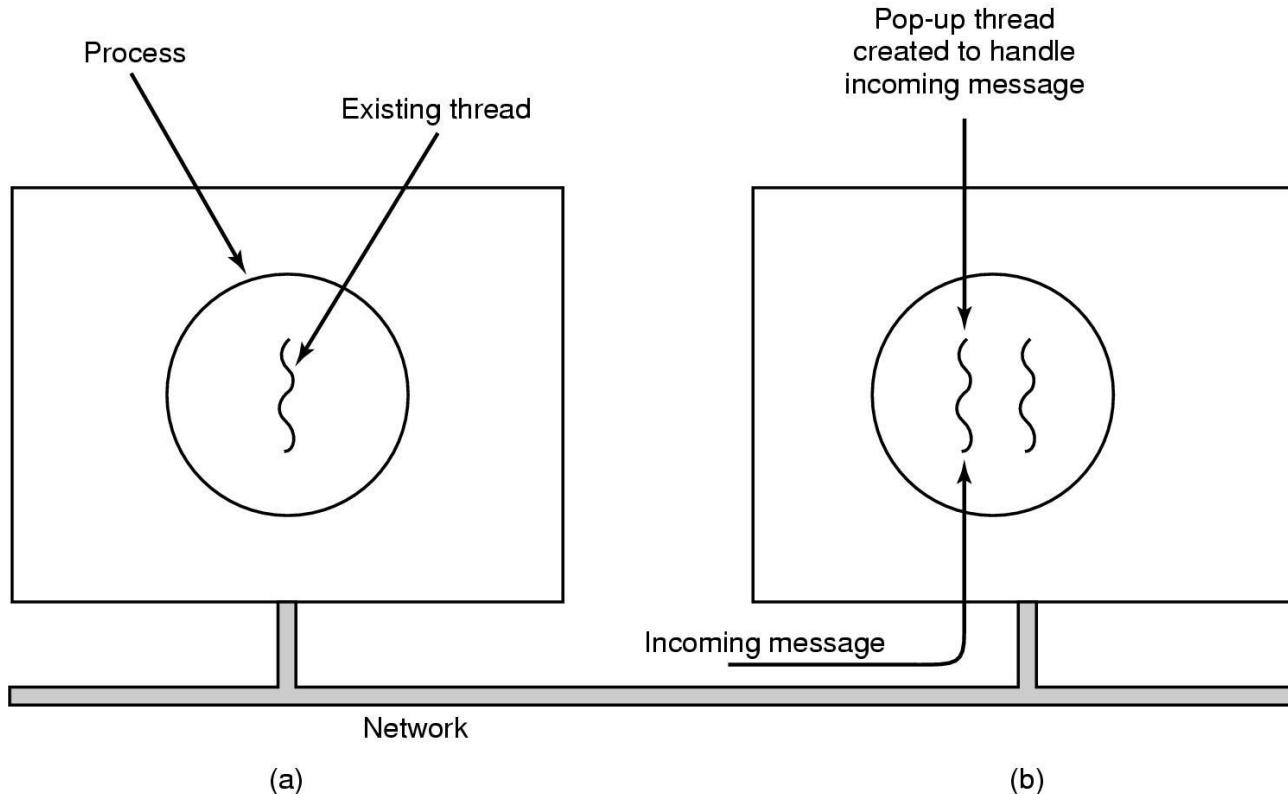


Multiplexing user-level threads onto kernel-level threads

# Scheduler Activations

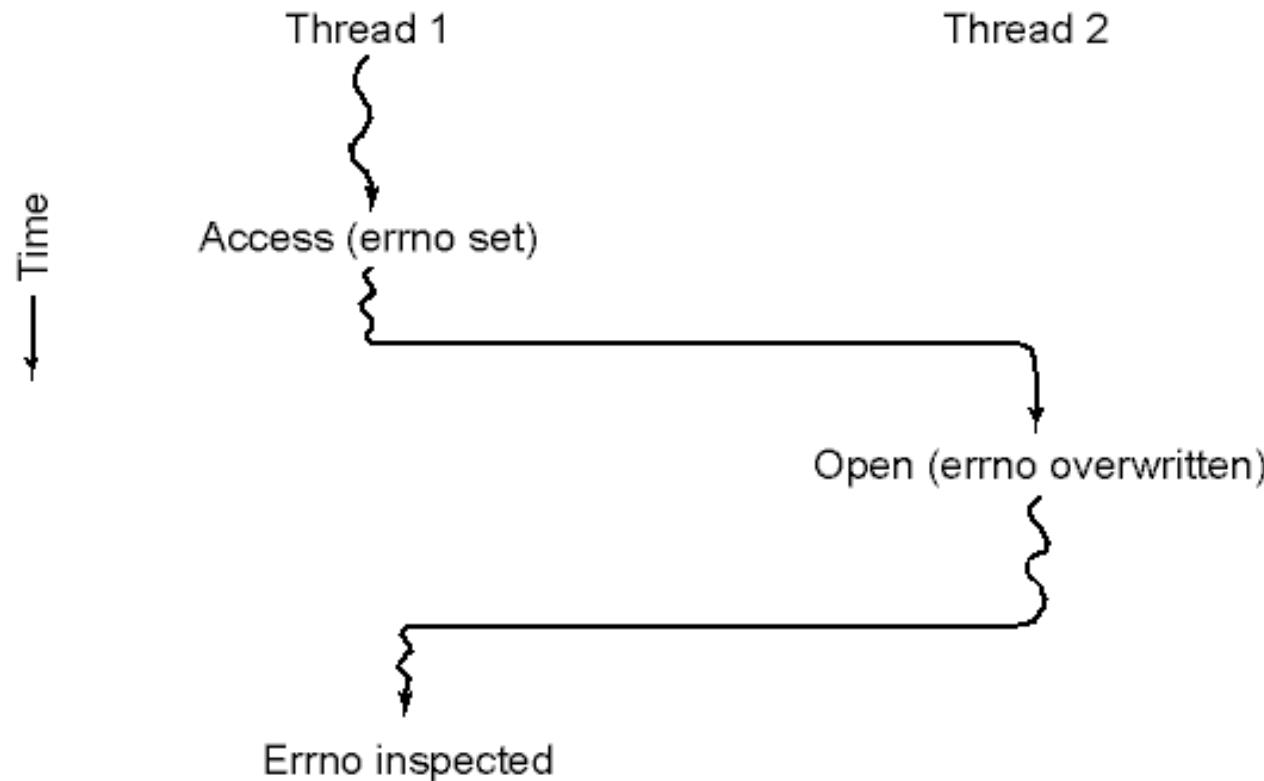
- Goal – mimic functionality of kernel threads
  - gain performance of user space threads
- Avoids unnecessary user/kernel transitions
  - eg. block by synchronizing thread in user space
- Kernel assigns virtual processors to each process
  - lets runtime system allocate threads to processors
- Problem:
  - Fundamental reliance on kernel (lower layer)
  - calling procedures in user space (higher layer)

# Pop-Up Threads



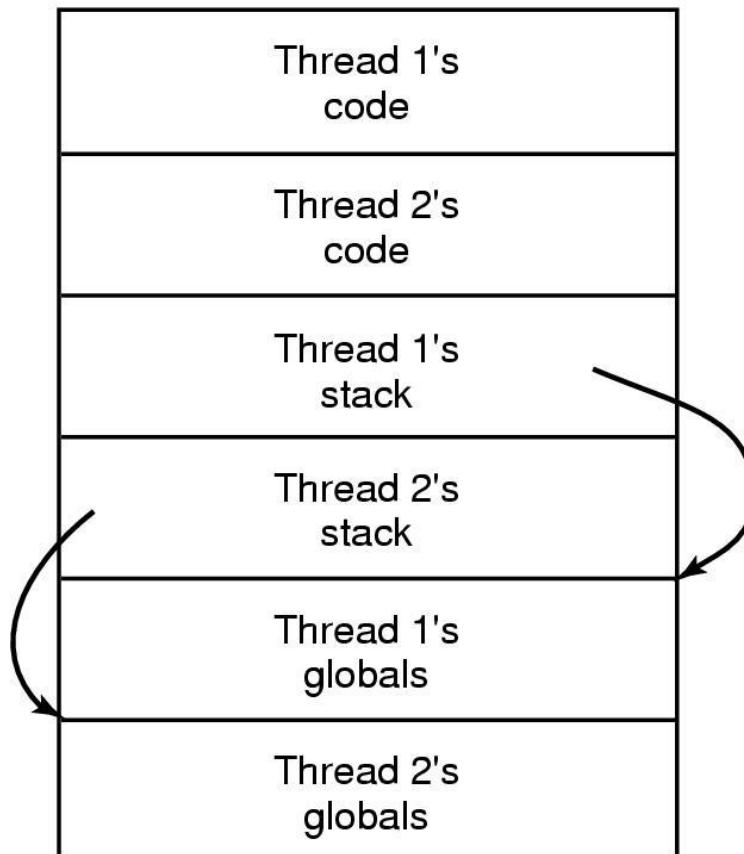
- Creation of a new thread when message arrives
  - (a) before message arrives
  - (b) after message arrives

# Making Single-Threaded Code Multithreaded (1)



Conflicts between threads over the use of a global variable

# Making Single-Threaded Code Multithreaded (2)

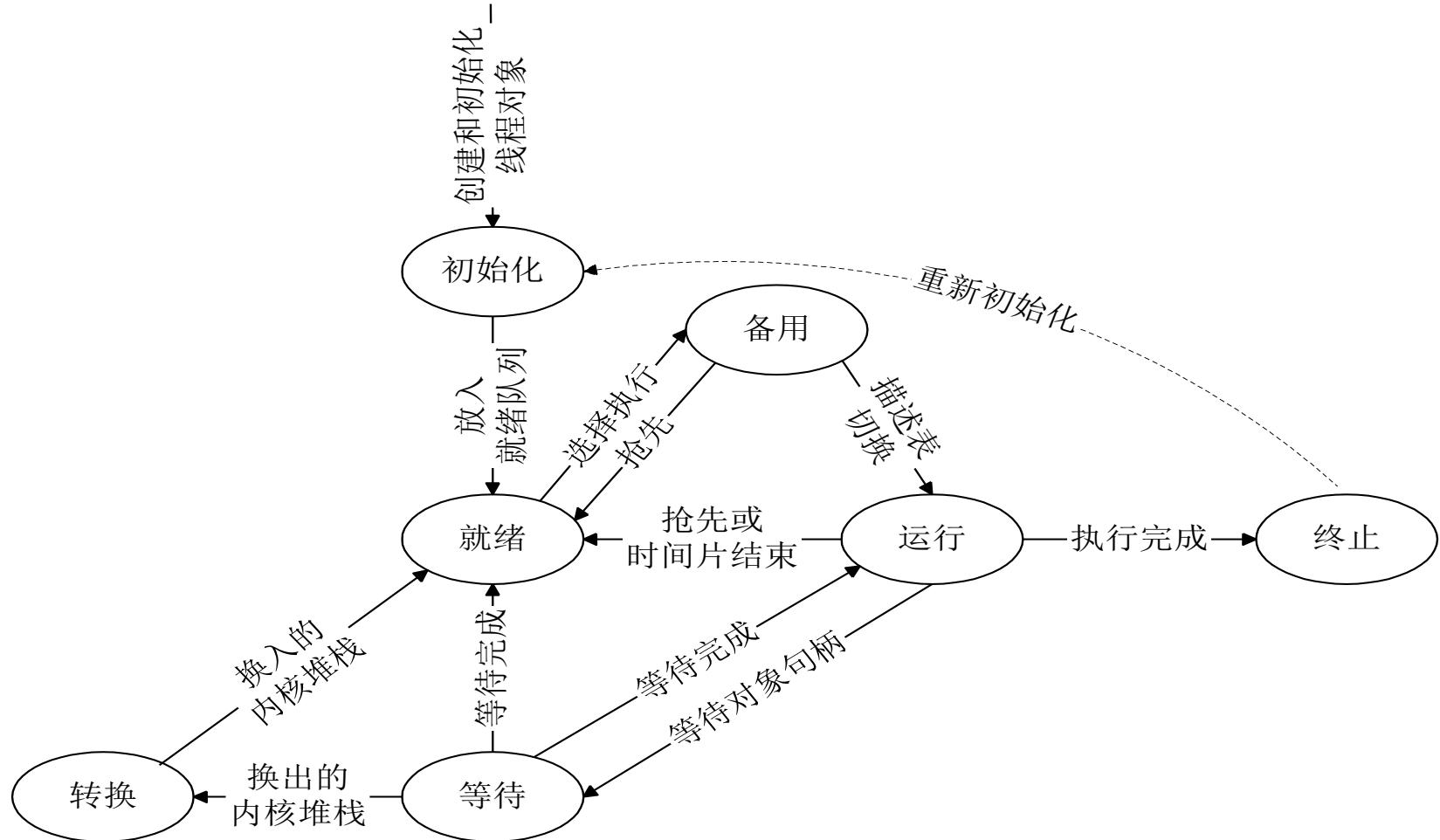


Threads can have private global variables

# Windows 线程

- 就绪状态(Ready): 进程已获得除处理机外的所需资源，等待执行。
- 备用状态(Standby): 特定处理器的执行对象，系统中每个处理器上只能有一个处于备用状态的线程。
- 运行状态(Running): 完成描述表切换，线程进入运行状态，直到内核抢先、时间片用完、线程终止或进行等待状态。
- 等待状态(Waiting): 线程等待对象句柄，以同步它的执行。等待结束时，根据优先级进入运行、就绪状态。
- 转换状态(Transition): 线程在准备执行而其内核堆栈处于外存时，线程进入转换状态；当其内核堆栈调回内存，线程进入就绪状态。
- 终止状态(Terminated): 线程执行完就进入终止状态；如执行体有一指向线程对象的指针，可将线程对象重新初始化，并再次使用。
- 初始化状态(Initialized): 线程创建过程中的线程状态；

# Windows 线程(con.)



# Windows 线程(con.)

- CreateThread()函数在调用进程的地址空间上创建一个线程，以执行指定的函数；返回值为所创建线程的句柄。
- ExitThread()函数用于结束本线程。
- SuspendThread()函数用于挂起指定的线程。
- ResumeThread()函数递减指定线程的挂起计数，挂起计数为0时，线程恢复执行。

# 进程通信 (IPC)

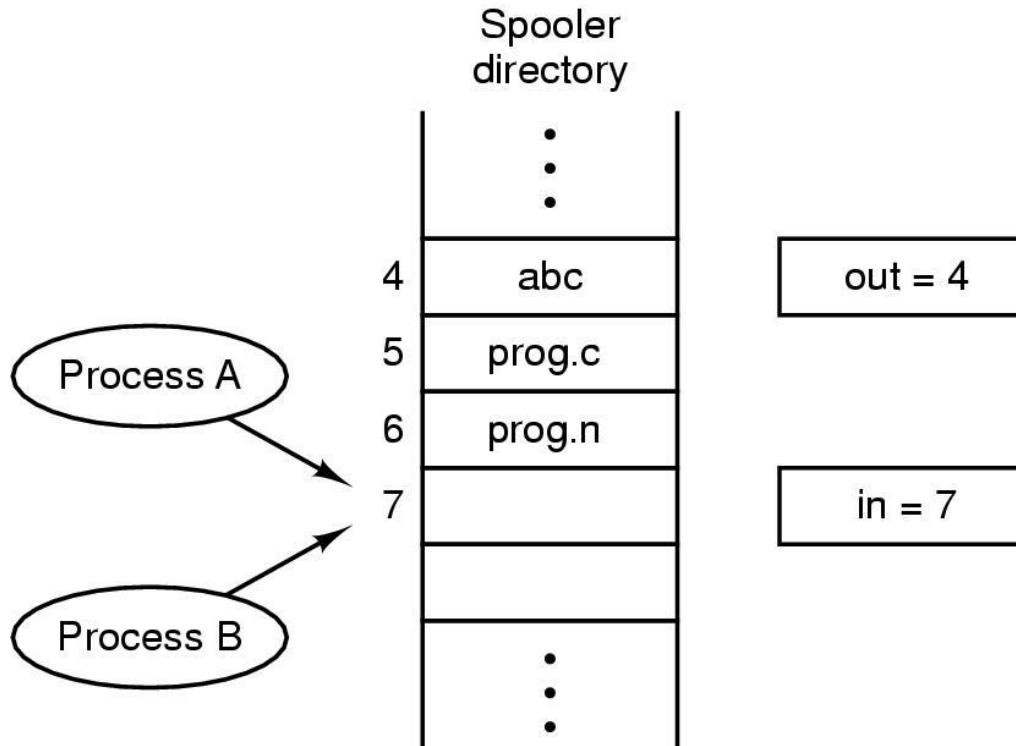
- 进程间的关系
  - 完全无关（异步）：不同进程间无任何关联
  - 使用共享数据（互斥）：有效保护各个进程的正确运行
  - 存在先后顺序（同步）：保证进程运行顺序的正确
- 进程通信需要考虑的问题
  - 如何实现两个进程间的信息传递
  - 如何协调两个或者多个进程之间的互斥和同步情况
- 问题难点分析
  - 金字塔法则：20%的简单问题 + 80% 的困难问题
  - 难点：临界区的保护

# Interprocess Communication

- How one process can pass information to another: 管道、共享存储器、消息传递
- Making sure two or more process do not get into each other's way when engaging in critical activities: 信号量和管程机制
- Proper sequencing when dependencies are present.
- Note: passing information is easy for threads since they share a common address space. However, keeping out of each other's hair and proper sequencing apply equally well to threads

# Interprocess Communication

## Race Conditions



Two processes want to access shared memory at same time

# Spooler 目录问题（互斥）

## Spooler 目录

.....
4: File1
5: File2
6: File3
7: Null
8: Null
.....

Out: 4

In: 7

进程A:  $N\_f\_s = In; //In == 7$

`InsertFileIntoSpooler(N_f_s);`

$In=N\_f\_s++; //In == 8$



进程切换，一切正常

进程B:  $N\_f\_s = In; //In == 8$

`InsertFileIntoSpooler(N_f_s);`

$In=N\_f\_s++; //In == 9$

## Spooler目录

.....
<b>4: File1</b>
<b>5: File2</b>
<b>6: File3</b>
<b>7: Null</b>
<b>8: Null</b>
.....

**Out: 4**

**In: 7**

进程A: **N\_f\_s = In; //In == 7**



进程切换

进程B: **N\_f\_s = In; //In == 7**

**InsertFileIntoSpooler(N\_f\_s);**

**In=N\_f\_s++; //In == 8**



进程切换, 进程B数据丢失

进程A: **InsertFileIntoSpooler(N\_f\_s);**

**In=N\_f\_s++; //In == 8**

# Interprocess Communication

- 进程调度机制的深入理解
  - 何时发生进程调度——时钟中断
  - 进程切换时的操作——压栈，进入就绪队列
- 边界情况的进程通信困难
  - 互斥情况——调度顺序影响运行结果
  - 同步情况——运行过程影响应用效果
- 多进程分时运行的临界情况
  - 正常情况下不存在任何问题
  - 临界情况下将产生不可预知的后果（墨菲法则）
- 解决的关键
  - 保持进程之间正确的运行顺序
  - OS与普通应用程序设计都需要考虑
  - 解决机制：提升系统运行的稳定性

# 一些IPC概念

- 竞争条件
  - 两个或者多个进程读写共享数据，而运行结果取决于进程运行时的精确时序，则这种情况称之为竞争条件
  - 当竞争条件存在时，进程处理结果可能失效甚至发生错误
- 临界区
  - 对共享内存（数据）进行访问的程序片断称为临界区
- 互斥
  - 防止多个进程同时操作相同共享数据的手段(多个进程不能同时使用同一个资源)
- 死锁
  - 多个进程互不相让，都得不到足够的资源
- 饥饿
  - 一个进程一直得不到资源（其他进程可能轮流占用资源）

# Interprocess Communication

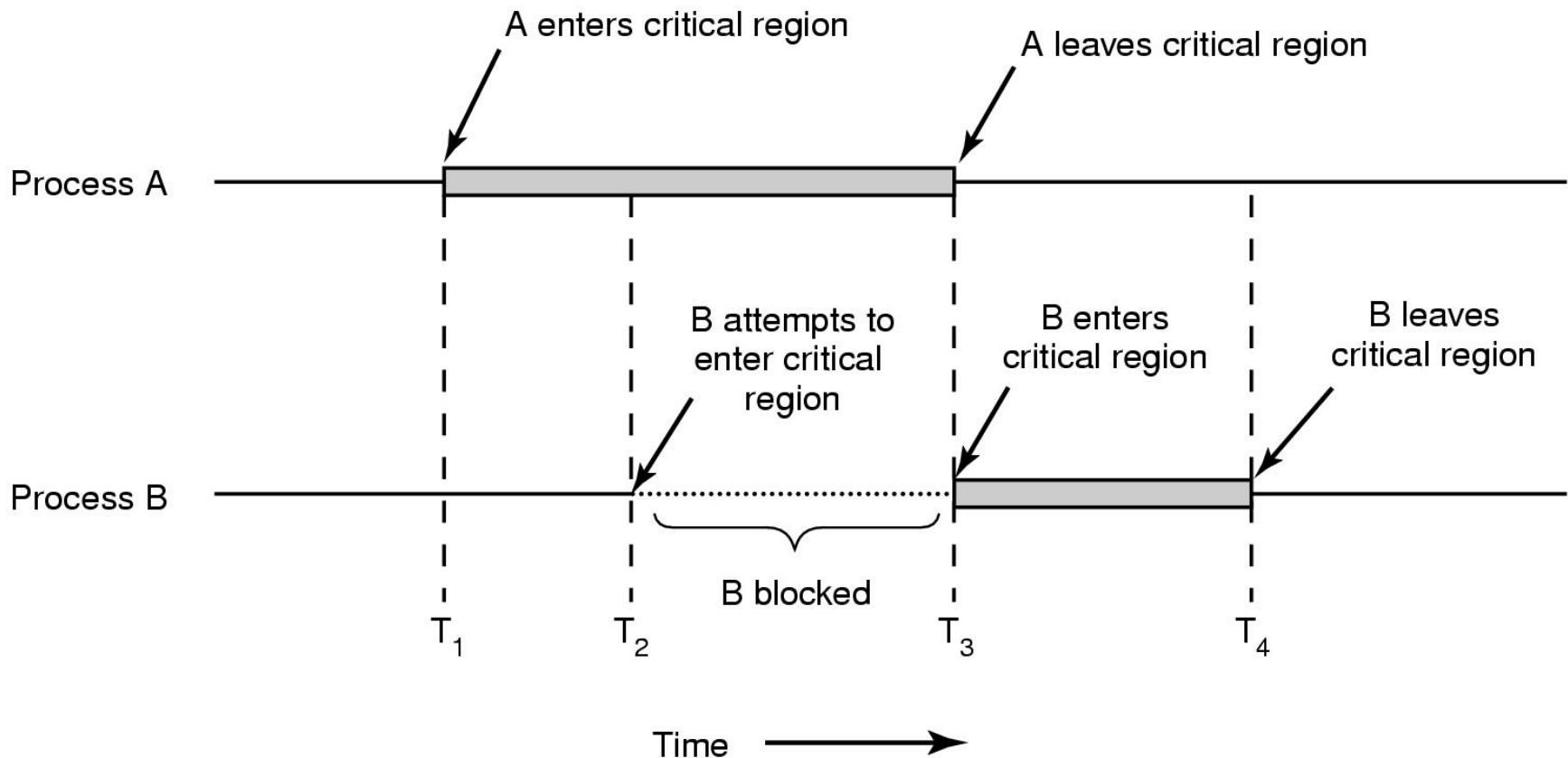
- 同步关系分析
  - 两个或者多个进程在运行顺序上存在固定的规则
  - 由于分时操作系统的不确定性，导致同步关系无法保持
  - 必须进行显式的程序控制，保证同步关系的正确
- 同步与互斥的区别
  - 互斥：体现为排他性，可表现为“0—1”关系(保护临界区，防止多个进程同时进入)
  - 同步：体现为时序性，比互斥更加复杂和多变(保证进程运行的顺序合理)
- 解决思路
  - 保证解决办法对互斥和同步的适用性

# Critical Regions (1)

Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region
  2. No assumptions made about speeds or numbers of CPUs
  3. No process running outside its critical region may block another process
  4. No process must wait forever to enter its critical region
- 互斥原则：任何两个进程不能同时处于临界区
  - 通用性原则：不应对CPU的速度和数目进行任何假设
  - 有效性原则：临界区外的进程不应阻塞其他进程
  - 合理性原则：不得使进程在临界区外无限制的等待；当进程无法进入临界区时，应放弃CPU资源

# Critical Regions (2)



Mutual exclusion using critical regions

# IPC机制

- 忙等待模型（只解决互斥问题）
  - 进程进入临界区时进行严格的安全检查
- 睡眠和唤醒模型（互斥与同步）
  - 通过改变进程的状态来实现互斥和同步
- 消息传递模型（复杂的IPC机制）
  - 以公共的通信机制来控制进程状态变化， 实现同步和互斥

# 忙等待模型

- 模型思想
  - 设定一个变量，标识临界区的状态
  - 互斥进程均检查这个变量，只有当其满足条件时方可进入临界区
- 模型方法
  - 关中断
  - 锁变量
  - 严格轮转
  - Peterson解决方案

# 关中断

进程A

```
Close_INT;  
Critical_region();  
Open_INT;
```

进程B

```
Close_INT;  
Critical_region();  
Open_INT;
```

- 解决思想
  - 在临界区中防止发生进程调度
  - 保证临界区操作的完整性
- 方法分析
  - 用户控制系统中断是非常危险的
  - 本方法对多个CPU系统将失去作用

# 锁变量

**lock = 0**

进程A

**While(lock != 0);**

**lock = 1;**

**Critical\_region();**

**lock = 0;**

进程B

**While(lock != 0);**

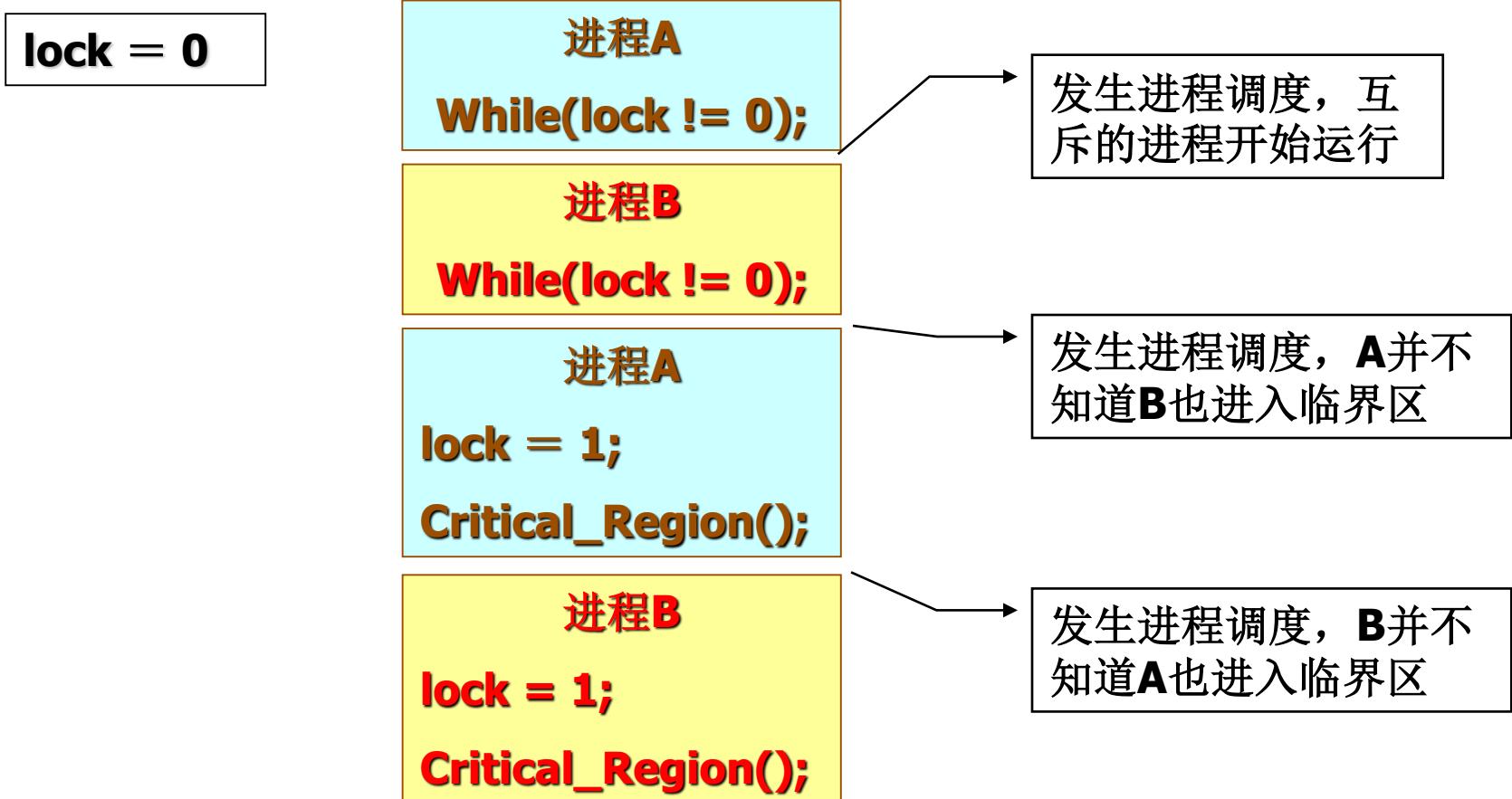
**lock = 1;**

**Critical\_region();**

**lock = 0;**

- 解决思想
  - 设定监控变量（lock），通过其值变化控制进程操作
- 方法分析
  - 依然存在竞争条件，不能根本解决互斥问题
  - 致命缺陷：不具有操作的原子性

# 锁变量的缺陷示例



# Mutual Exclusion with Busy Waiting

```
while (TRUE) {  
    while (turn != 0)      /* loop */ ;  
    critical_region( );  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1)      /* loop */ ;  
    critical_region( );  
    turn = 0;  
    noncritical_region();  
}
```

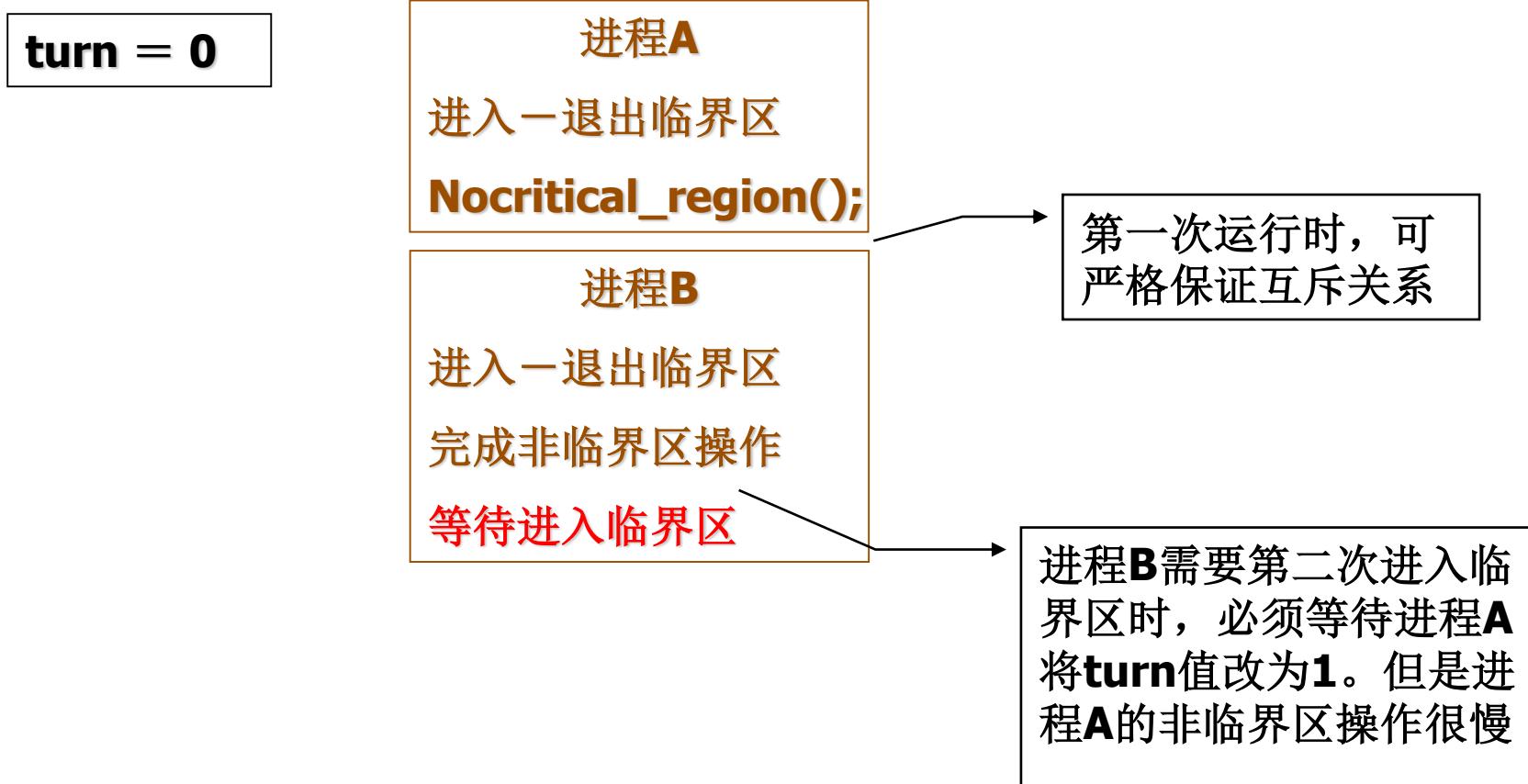
(b)

## Proposed solution to critical region problem

- (a) Process 0.      (b) Process 1.

严格轮转法：忙等待浪费CPU时间；存在边界情况，违反  
解决原则第三条

# 严格轮转法的缺陷示例



# Mutual Exclusion with Busy Waiting (2)

```
#define FALSE 0
#define TRUE 1
#define N      2          /* number of processes */

int turn;                  /* whose turn is it? */
int interested[N];         /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;               /* number of the other process */

    other = 1 - process;     /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;          /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion

# Use Peterson's solution

进程A

**While(TRUE)**

{

**enter\_region(AID);**

**Critical\_region;**

**leave\_region(AID);**

}

进程B

**While(TRUE)**

{

**enter\_region(BID);**

**Critical\_region;**

**leave\_region(BID);**

}

# Mutual Exclusion with Busy Waiting (3)

测试并加锁：硬件TSL指令

硬件实现提高速度，适用于多处理机；依然存在忙等待的问题

TSL RX LOCK: 硬件实现锁变量机制，保证LOCK读写原子性

enter\_region:

TSL REGISTER,LOCK	copy lock to register and set lock to 1
CMP REGISTER,#0	was lock zero?
JNE enter_region	if it was non zero, lock was set, so loop
RET   return to caller; critical region entered	

leave\_region:

MOVE LOCK,#0	store a 0 in lock
RET   return to caller	

Entering and leaving a critical region using the  
TSL instruction

# 忙等待模型分析

- 优点分析
  - 实现机制简单易懂，可有效保证互斥
- 缺点分析
  - 只适用于两个进程间互斥，不具有通用性
  - 忙等待严重浪费CPU资源，降低硬件效率
  - “优先级调度” + “忙等待” = 优先级反转
    - 进程H和L，H优先级比L优先级高
    - L先进入临界区，H后进入临界区
    - H开始忙等待，而L由于无法获得CPU也不能离开临界区

# 睡眠-唤醒模型

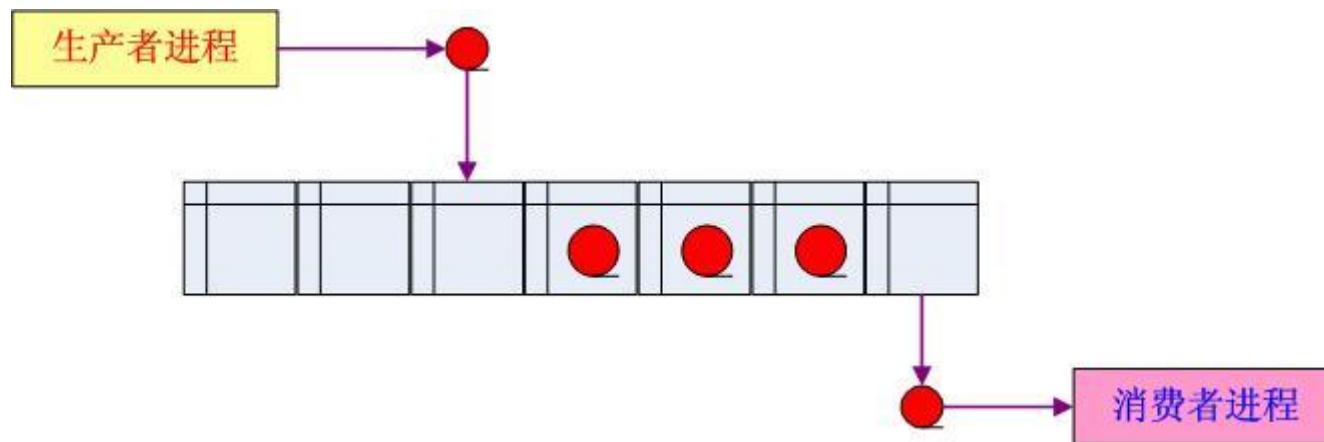
- 模型思想
  - OS提供系统调用原语（Atomic Action），改变进程状态
  - 无法进入临界区的进程转为阻塞态，条件满足则被唤醒
  - 可有效克服忙等待造成的资源浪费
  - 更重要的优点：可同时实现同步与互斥
- 模型方法
  - 简单的睡眠—唤醒方法
  - 信号量机制
  - 管程方法

# 简单的睡眠—唤醒方法

- 操作系统原语设计
  - Sleep(): 调用该原语的进程将变为阻塞态
  - Wakeup(ID): 该原语将唤醒ID标识的进程
- 原语使用思想
  - 进入临界区前检查竞争条件，如不满足则睡眠
  - 离开临界区后唤醒互斥进程

# 生产者—消费者问题

- 问题描述
  - 一个有限空间的共享缓冲区，负责存放货物
  - 生产者向缓冲区中放物品，缓冲区满则不能放
  - 消费者从缓冲区中拿物品，缓冲区空则不能拿



# Sleep and Wakeup

```
#define N 100                                /* number of slots in the buffer */
int count = 0;                                /* number of items in the buffer */

void producer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        item = produce_item();                /* generate next item */
        if (count == N) sleep();              /* if buffer is full, go to sleep */
        insert_item(item);                  /* put item in buffer */
        count = count + 1;                  /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);    /* was buffer empty? */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                            /* repeat forever */
        if (count == 0) sleep();              /* if buffer is empty, got to sleep */
        item = remove_item();                /* take item out of buffer */
        count = count - 1;                  /* decrement count of items in buffer */
        if (count == N - 1) wakeup(producer); /* was buffer full? */
        consume_item(item);                /* print item */
    }
}
```

Producer-consumer problem with fatal race condition

# 潜在的竞争条件

```
#define N 100  
  
int lock = 0  
  
int count = 0
```



- 临界情况
  - count的访问未加限制，形成竞争条件
  - 必须增加唤醒等待位，以解决互斥问题
- 方法分析
  - 较忙等待更进一步，有效节省CPU资源
  - 存在竞争条件，需要额外处理
  - 当互斥进程数增加时，方法有效性下降
- 改进办法
  - 由OS提供宏观调控管理机制
  - 彻底消除调度顺序引起的错乱
  - OS实现更高层次的原语级操作

# 信号量机制

- 基本概念
  - 信号量：表示累积的睡眠或者唤醒操作
  - Down与Up原语（P/V、S/W原语）
  - Dijkstra于1965年提出Probern和Verhogen原语
- 核心思想
  - 引入新的数据结构定义：信号量
  - 利用信号量，提供更为复杂的原语级操作
  - 从根本上解决“潜在竞争条件”问题
  - 可以方便的推广到一般情况，适用于多进程的互斥与同步
- 原语（primitive）

由若干条指令构成的“原子操作(atomic operation)”过程，作为一个整体而不可分割——要么全都完成，要么全都不做。许多系统调用是原语。但并不是所有的系统调用都是原语。

# 信号量与Down/Up原语

- 基本概念

- 对于一种互斥或者同步关系，用一个整型变量来描述
- 当信号量大于0时，说明“环境安全”，可继续运行
- 当信号量小于0时，说明“互斥等待”，只能阻塞
- 推广到一般情况，信号量可解决复杂的同步互斥问题

## Down原语

**Down(s)**

{

**s = s - 1;**

**if(s < 0) wait;**

}

## Up原语

**Up(s)**

{

**s = s+1;**

**if(s <= 0) wakeup();**

}

原语：OS以关中断形式实现的不可打断操作

# 最基本互斥机制的实现

```
int mutex = 1
```

进程A

```
while(TRUE)
{
    nocritical_region();
    Down(mutex);
    critical_region();
    Up(mutex);
    nocritical_region();
}
```

进程B

```
while(TRUE)
{
    nocritical_region();
    Down(mutex);
    critical_region();
    Up(mutex);
    nocritical_region();
}
```

# Semaphores

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

/* number of slots in the buffer */
/* semaphores are a special kind of int */
/* controls access to critical region */
/* counts empty buffer slots */
/* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

/* TRUE is the constant 1 */
/* generate something to put in buffer */
/* decrement empty count */
/* enter critical region */
/* put new item in buffer */
/* leave critical region */
/* increment count of full slots */

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

/* infinite loop */
/* decrement full count */
/* enter critical region */
/* take item from buffer */
/* leave critical region */
/* increment count of empty slots */
/* do something with the item */
```

The producer-consumer problem using semaphores

- 互斥信号量
  - mutex: 防止多个进程同时进入临界区
  - 读者和写者不能同时进入共享数据区
  - 多个写者不能同时进入共享数据区
  - 多个读者可以同时进入共享数据区
- 同步信号量
  - empty和full: 保证事件发生的顺序
  - 缓冲区满时, Producer停止运行
  - 缓冲区空时, Consumer停止运行
  - 读者进入缓冲区, 写者必须等待
  - 写者进入缓冲区, 读者必须等待
  - 读者优先: 一旦有读者进入, 则后续读者均可进入
  - 合理顺序: 读者在先来的写者之后
  - 写者优先: 只要有写者等待, 则后续读者必须等待

# Mutexes

mutex\_lock:

```
TSL REGISTER,MUTEX  
CMP REGISTER,#0  
JZE ok  
CALL thread_yield  
JMP mutex_lock
```

```
| copy mutex to register and set mutex to 1  
| was mutex zero?  
| if it was zero, mutex was unlocked, so return  
| mutex is busy; schedule another thread  
| try again later
```

ok: RET | return to caller; critical region entered

mutex\_unlock:

```
MOVE MUXTEX,#0  
RET | return to caller
```

```
| store a 0 in mutex
```

Implementation of *mutex\_lock* and *mutex\_unlock*

# 管程方法

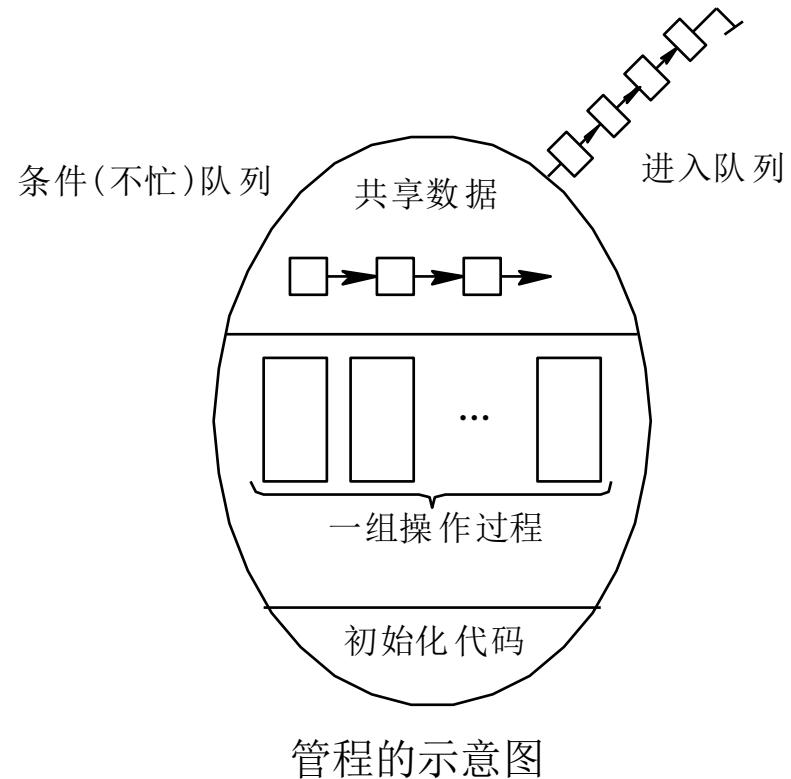
- 死锁的危险
  - 一旦信号量的处理代码发生错误，则会引起进程死锁
  - 引入新的进程同步工具——管程（Monitors）
- 核心思想
  - 实现一种包含过程、变量、数据结构的独立模块
  - 任何时刻，只能有一个活跃进程在管程内
  - 由编译器提供支持，实现进程间的互斥
- 方法分析
  - 优点：实现了进程互斥的自动化
  - 缺点：依赖于编译器，无法通用或普及

# 管程方法(con.)

- 管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程，改变管程中的数据。
- 管程由三部分组成：
  - 局部于管程的共享变量说明
  - 对该数据结构进行操作的一组过程
  - 对管程中数据设置初值的语句
- 任何管程外的过程都不能访问管程内的数据结构。管程相当于围墙，将共享变量和对它进行操作的若干过程围了起来，进程只要访问临界资源就必须通过管程。
- 管程每次只允许一个进程进入管程，实现了互斥。

# 管程的引入

- 1973年，由Hoare和Hansen提出的一种高级同步原语，将共享变量以及对共享变量能够进行的所有操作集中在一个模块中。
- 管程的定义：一个管程是一个由过程、变量及数据结构等组成的集合，它们组成一个特殊的模块或软件包。进程可在任何需要的时候调用管程中的过程，但它们不能在管程之外声明的过程中直接访问管程内的数据结构。



# 管程的主要特性

- 为了保证管程共享变量的数据完整性，规定管程互斥进入，即：任一时刻管程中只能有一个活跃进程，这一特性使管程能有效地完成互斥。
- 进入管程时的互斥由编译器负责，而不是由程序员来安排互斥，出错的可能性小。

# 管程的多个进程进入

- 管程提供了实现互斥的途径，此时，还需要一种办法使得进程在无法继续运行时被阻塞，其解决的办法是：引入条件变量和wait、signal两个操作。
- 当一个管程过程无法继续运行时，将在某个条件变量上执行wait → 被阻塞
- 条件变量不能象信号量那样积累信号，为了防止信号丢失，规定： wait操作必须在signal操作之前。

# 管程中的多个进程进入

- 当一个进入管程的进程执行wait操作时，它应当释放管程的互斥权；当一个进入管程的进程执行signal操作时（如 P 唤醒 Q），管程中便存在两个同时处于活动状态的进程。
- Brinch Hansen的处理方法：执行signal操作的进程必须立即退出管程，即， signal语句只可作为管程过程的最后一条语句。
- 如果在一个条件变量上有多个进程正在等待，则对该条件变量的signal后， 系统调度程序只能在其中选择一个使其恢复运行。

# Monitors (1)

**monitor** *example*

**integer** *i*;

**condition** *c*;

**procedure** *producer()*;

        .

        .

        .

**end**;

**procedure** *consumer()*;

        .

        .

        .

**end**;

**end monitor**;

Example of a monitor

# Monitors (2)

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
end;
function remove: integer;
begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
end;
count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
begin
    item = produce_item;
    ProducerConsumer.insert(item)
end
end;
procedure consumer;
begin
    while true do
begin
    item = ProducerConsumer.remove;
    consume_item(item)
end
end;
```

- Outline of producer-consumer problem with monitors
  - only one monitor procedure active at one time
  - buffer has  $N$  slots

# Monitors (3)

```
public class ProducerConsumer {  
    static final int N = 100;           // constant giving the buffer size  
    static producer p = new producer(); // instantiate a new producer thread  
    static consumer c = new consumer(); // instantiate a new consumer thread  
    static our_monitor mon = new our_monitor(); // instantiate a new monitor  
    public static void main(String args[]) {  
        p.start();                    // start the producer thread  
        c.start();                    // start the consumer thread  
    }  
    static class producer extends Thread {  
        public void run() {           // run method contains the thread code  
            int item;  
            while (true) {             // producer loop  
                item = produce_item();  
                mon.insert(item);  
            }  
        }  
        private int produce_item() { ... } // actually produce  
    }  
    static class consumer extends Thread {  
        public void run() {           // run method contains the thread code  
            int item;  
            while (true) {             // consumer loop  
                item = mon.remove();  
                consume_item (item);  
            }  
        }  
        private void consume_item(int item) { ... } // actually consume  
    }  
}
```

Solution to producer-consumer problem in Java (part 1)

# Monitors (4)

```
static class our_monitor {           // this is a monitor
    private int buffer[] = new int[N];
    private int count = 0, lo = 0, hi = 0; // counters and indices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep(); // if the buffer is full, go to sleep
        buffer [hi] = val;          // insert an item into the buffer
        hi = (hi + 1) % N;          // slot to place next item in
        count = count + 1;          // one more item in the buffer now
        if (count == 1) notify();    // if consumer was sleeping, wake it up
    }
    public synchronized int remove() {
        int val;
        if (count == 0) go_to_sleep(); // if the buffer is empty, go to sleep
        val = buffer [lo];          // fetch an item from the buffer
        lo = (lo + 1) % N;          // slot to fetch next item from
        count = count - 1;          // one few items in the buffer
        if (count == N - 1) notify(); // if producer was sleeping, wake it up
        return val;
    }
    private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
}
```

Solution to producer-consumer problem in Java (part 2)

# 信号量模型分析

- 优点分析
  - 彻底解决忙等待弊端，提高OS的管理层次
  - 可实现复杂的同步与互斥情况，特别是多进程间通信
  - 可最大限度的保证并发效率
- 缺点分析
  - 实现机制复杂，互斥和同步关系分析困难
  - 存在死锁陷阱，需要谨慎小心严密的设计

# 消息传递模型

- 模型思想
  - 提供Send和Receive原语，用来传递消息
  - 进程通过发送和接收消息来实现互斥与同步
  - 重要的优点：不仅实现了同步，还能够传送大容量信息
- 设计要点
  - 如何保证消息传递中不丢失？（ACK机制）
  - 如何命名进程，使得其地址唯一并确认身份
  - 如何规范消息格式，降低处理时间

# Message Passing

```
#define N 100                                     /* number of slots in the buffer */

void producer(void)
{
    int item;
    message m;                                     /* message buffer */

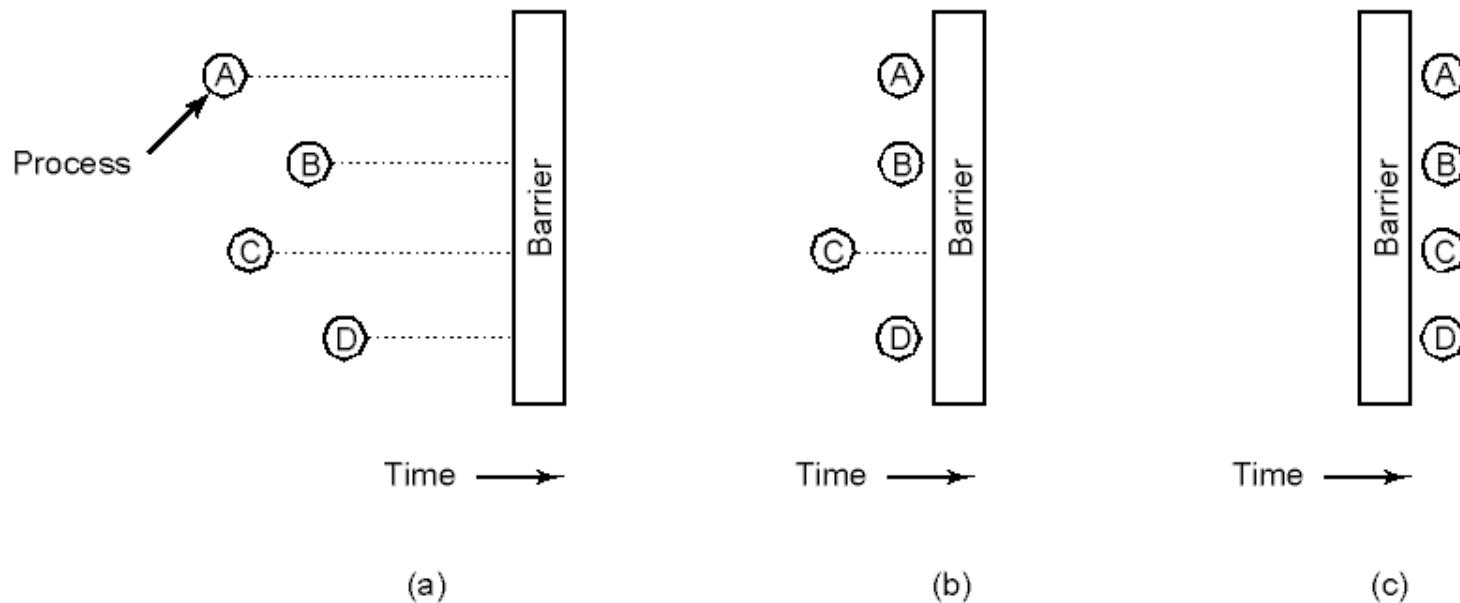
    while (TRUE) {
        item = produce_item();                      /* generate something to put in buffer */
        receive(consumer, &m);                      /* wait for an empty to arrive */
        build_message(&m, item);                    /* construct a message to send */
        send(consumer, &m);                         /* send item to consumer */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);                     /* get message containing item */
        item = extract_item(&m);                  /* extract item from message */
        send(producer, &m);                       /* send back empty reply */
        consume_item(item);                      /* do something with the item */
    }
}
```

The producer-consumer problem with N messages

# Barriers

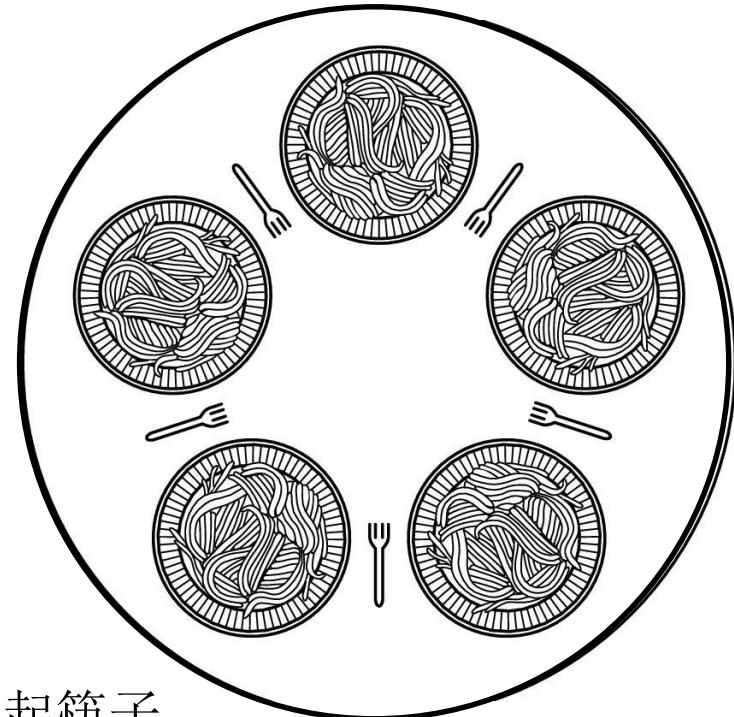


- Use of a barrier
  - processes approaching a barrier
  - all processes but one blocked at barrier
  - last process arrives, all are let through

# Dining Philosophers (1)

- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time
- How to prevent deadlock

- 互斥分析
  - 筷子: 同一时刻只能有一个哲学家拿起筷子
- 同步分析
  - 就餐: 只有获得两个筷子后才能进餐
- 特殊情况
  - 死锁: 如果每个哲学家都拿起一只筷子, 都饿死
  - 并行程度: 五只筷子允许两人同时进餐



# Dining Philosophers (2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */

{
    while (TRUE) {
        think();                                /* philosopher is thinking */
        take_fork(i);                            /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat();                                   /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                   /* put right fork back on the table */
    }
}
```

A nonsolution to the dining philosophers problem

# Dining Philosophers (3)

```
#define N          5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N    /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)           /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {
        think();                 /* repeat forever */
        take_forks(i);           /* philosopher is thinking */
        eat();                   /* acquire two forks or block */
        put_forks(i);            /* yum-yum, spaghetti */
    }                           /* put both forks back on table */
}
```

Solution to dining philosophers problem (part 1) 108

# Dining Philosophers (4)

```
void take_forks(int i)                                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = HUNGRY;                               /* record fact that philosopher i is hungry */
    test(i);                                         /* try to acquire 2 forks */
    up(&mutex);                                      /* exit critical region */
    down(&s[i]);                                     /* block if forks were not acquired */
}

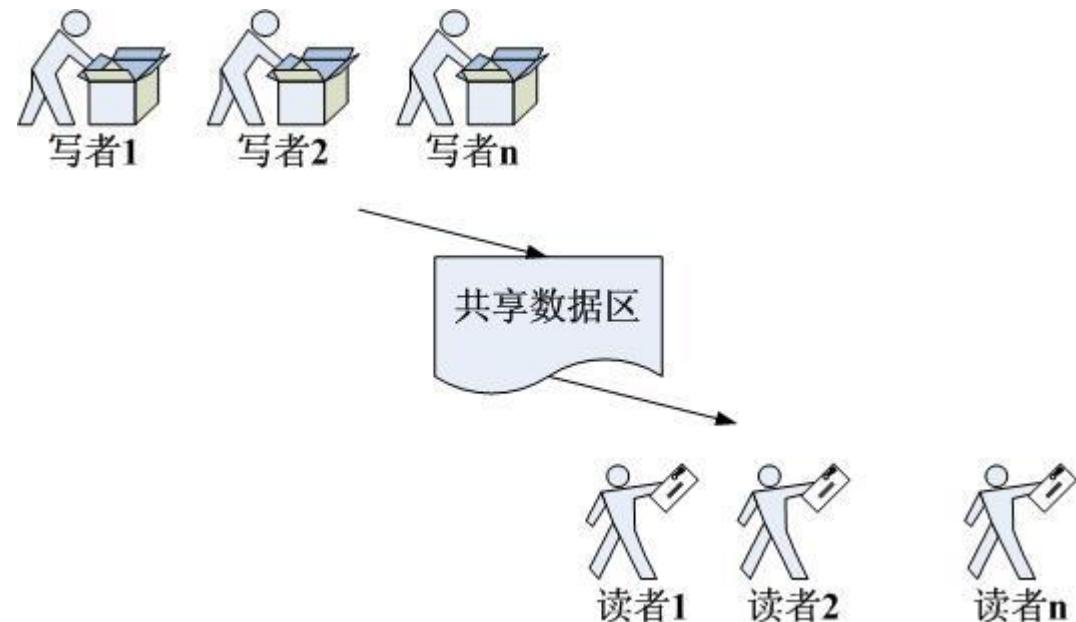
void put_forks(i)                                    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                                     /* enter critical region */
    state[i] = THINKING;                            /* philosopher has finished eating */
    test(LEFT);                                      /* see if left neighbor can now eat */
    test(RIGHT);                                     /* see if right neighbor can now eat */
    up(&mutex);                                      /* exit critical region */
}

void test(i)                                         /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Solution to dining philosophers problem (part 2) 109

# 读者-写者问题

- 问题描述
  - 写者向数据区放数据，读者从数据区获取数据
  - 多个读者可同时读取数据
  - 多个写者不能同时写数据
  - 读者和写者的控制策略变化多端



- **互斥分析**
  - 读者和写者不能同时进入共享数据区
  - 多个写者不能同时进入共享数据区
  - 多个读者可以同时进入共享数据区
- **同步分析**
  - 读者进入缓冲区，写者必须等待
  - 写者进入缓冲区，读者必须等待
  - 读者优先：一旦有读者进入，则后续读者均可进入
  - 合理顺序：读者在先来的写者之后
  - 写者优先：只要有写者等待，则后续读者必须等待

# The Readers and Writers Problem

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
```

/\* use your imagination \*/  
/\* controls access to 'rc' \*/  
/\* controls access to the database \*/  
/\* # of processes reading or wanting to \*/

/\* repeat forever \*/  
/\* get exclusive access to 'rc' \*/  
/\* one reader more now \*/  
/\* if this is the first reader ... \*/  
/\* release exclusive access to 'rc' \*/  
/\* access the data \*/  
/\* get exclusive access to 'rc' \*/  
/\* one reader fewer now \*/  
/\* if this is the last reader ... \*/  
/\* release exclusive access to 'rc' \*/  
/\* noncritical region \*/

/\* repeat forever \*/  
/\* noncritical region \*/  
/\* get exclusive access \*/  
/\* update the data \*/  
/\* release exclusive access \*/

A solution to the readers and writers problem

# The Sleeping Barber Problem (1)

- 互斥分析
  - 理发椅上只能有一位顾客
  - 等待座位是有限缓冲区
- 同步分析
  - 只要存在顾客，理发师就不能睡觉
- 信号量设计
  - 互斥信号量：实现对“等待顾客数”的互斥
  - 同步信号量1：理发师“睡眠”和“工作”的同步
  - 同步信号量2：等待顾客与等待座位的同步



# The Sleeping Barber Problem (2)

```
#define CHAIRS 5          /* # chairs for waiting customers */

typedef int semaphore;    /* use your imagination */

semaphore customers = 0;   /* # of customers waiting for service */
semaphore barbers = 0;     /* # of barbers waiting for customers */
semaphore mutex = 1;       /* for mutual exclusion */
int waiting = 0;           /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);      /* go to sleep if # of customers is 0 */
        down(&mutex);          /* acquire access to 'waiting' */
        waiting = waiting - 1;  /* decrement count of waiting customers */
        up(&barbers);          /* one barber is now ready to cut hair */
        up(&mutex);            /* release 'waiting' */
        cut_hair();             /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);           /* enter critical region */
    if (waiting < CHAIRS) {  /* if there are no free chairs, leave */
        waiting = waiting + 1; /* increment count of waiting customers */
        up(&customers);      /* wake up barber if necessary */
        up(&mutex);           /* release access to 'waiting' */
        down(&barbers);       /* go to sleep if # of free barbers is 0 */
        get_haircut();         /* be seated and be serviced */
    } else {
        up(&mutex);           /* shop is full; do not wait */
    }
}
```

Solution to sleeping barber problem.

# 调度程序

- 对CPU资源进行合理的分配使用，以提高处理机利用率，并使各用户公平地得到处理机资源。
- 可从不同的角度来判断处理机调度算法的性能，如用户的角度、处理机的角度和算法实现的角度。实际的处理机调度算法选择是一个综合的判断结果。

# 调度程序(con.)

- 三级调度---高级、中级和低级调度：
  - 高级调度---又称作业调度或长调度：用于决定把外存上后备队列中哪些作业调入内存，并为它们创建进程、分配必要的资源，然后将新创建的进程插入到就绪队列中，准备运行。决定：1) 接纳多少个作业；2) 接纳哪些作业。
  - 低级调度---又称进程调度或短调度：用来决定就绪队列中的哪个进程应获得处理机，然后再由分派程序执行把处理机分配给该进程的具体操作。进程调度分两种调度方式：非抢占方式、抢占方式。
  - 中级调度：挂起和激活，存储器管理中的对换功能。主要目的是为了提高内存的利用率和系统的吞吐量。

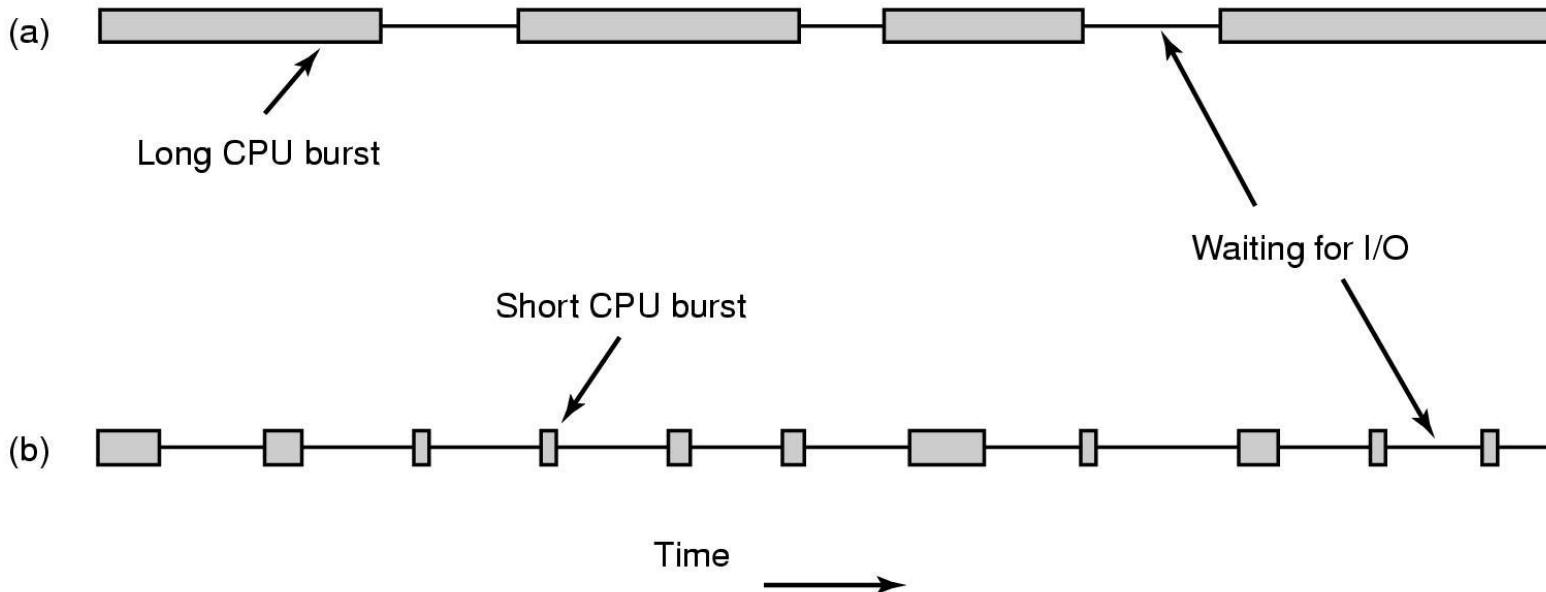
# 调度程序(con.)

- 记录所有进程的运行状况（静态和动态）
  - 当进程出让CPU或调度程序剥夺执行状态进程占用的CPU时，选择适当的进程分派CPU
  - 完成上下文切换
- 进程的上下文切换过程：
    - 用户态执行进程A代码——进入OS核心（通过时钟中断或系统调用）
    - 保存进程A的上下文，恢复进程B的上下文（CPU寄存器和一些表格的当前指针）
    - 用户态执行进程B代码

注：上下文切换之后，指令和数据快速缓存cache通常需要更新，执行速度降低

# Scheduling

## Introduction to Scheduling (1)



- Bursts of CPU usage alternate with periods of I/O wait
  - a CPU-bound process
  - an I/O bound process

CPU繁忙型作业——需要大量的CPU时间进行计算，而很少请求I/O。如，科学计算

I/O繁忙型作业——是指CPU进行处理时，需频繁地请求I/O。如，大多数事务处理

# Introduction to Scheduling (2)

## All systems

Fairness - giving each process a fair share of the CPU

Policy enforcement - seeing that stated policy is carried out

Balance - keeping all parts of the system busy

## Batch systems

Throughput - maximize jobs per hour

Turnaround time - minimize time between submission and termination

CPU utilization - keep the CPU busy all the time

## Interactive systems

Response time - respond to requests quickly

Proportionality - meet users' expectations

## Real-time systems

Meeting deadlines - avoid losing data

Predictability - avoid quality degradation in multimedia systems

## Scheduling Algorithm Goals

# 面向用户的调度性能准则

- 周转时间：作业从提交到完成（得到结果）所经历的时间。包括：在收容队列中等待，CPU上执行，就绪队列和阻塞队列中等待，结果输出等待——批处理系统
  - 平均周转时间T
  - 平均带权周转时间（带权周转时间W是  $T(\text{周转})/T(\text{CPU执行})$ ）
- 响应时间：用户输入一个请求（如击键）到系统给出首次响应（如屏幕显示）的时间——分时系统
- 截止时间：开始截止时间和完成截止时间——实时系统，与周转时间有些相似。
- 公平性：不因作业或进程本身的特性而使上述指标过分恶化。如长作业等待很长时间。
- 优先级：可以使关键任务达到更好的指标。

# 面向系统的调度性能准则

- 吞吐量：单位时间内所完成的作业数，跟作业本身特性和调度算法都有关系——批处理系统
  - 平均周转时间不是吞吐量的倒数，因为并发执行的作业在时间上可以重叠。如：在2小时内完成4个作业，而每个周转时间是1小时，则吞吐量是2个作业/小时
- 处理机利用率：——大中型主机
- 各种设备的均衡利用：如CPU繁忙的作业和I/O繁忙（指次数多，每次时间短）的作业搭配——大中型主机

注：调度算法本身的调度性能准则：易于实现、执行开销比

# 常用调度算法

- 先来先服务调度算法
- 短作业(进程)优先调度算法
- 高优先权优先调度算法
- 高响应比优先调度算法
- 基于时间片的轮转调度算法

# 先来先服务 (FCFS, First Come First Service)

- 按照作业提交或进程变为就绪状态的先后次序，分派CPU；
- 当前作业或进程占用CPU，直到执行完或阻塞，才出让CPU（非抢占方式）。
- 在作业或进程唤醒后（如I/O完成），并不立即恢复执行，通常等到当前作业或进程出让CPU。最简单的算法。
- 特点：比较有利于长作业，而不利于短作业；有利于CPU繁忙的作业，而不利于I/O繁忙的作业。

**【例】** 设在单道系统中用FCFS算法调度如下作业，请完成下表。

进程名	A	B	C	D	E	平均
到达时间	<b>9:00</b>	<b>9:10</b>	<b>9:30</b>	<b>10:00</b>	<b>10:15</b>	
服务时间	<b>30分钟</b>	<b>1小时</b>	<b>10分钟</b>	<b>50分钟</b>	<b>20分钟</b>	
完成时间	<b>9:30</b>	<b>10:30</b>	<b>10:40</b>	<b>11:30</b>	<b>11:50</b>	
周转时间	<b>30分钟</b>	<b>80分钟</b>	<b>70分钟</b>	<b>90分钟</b>	<b>95分钟</b>	<b>73分钟</b>
带权周转时间	<b>1</b>	<b>1.33</b>	<b>7</b>	<b>1.8</b>	<b>4.75</b>	<b>3.176</b>

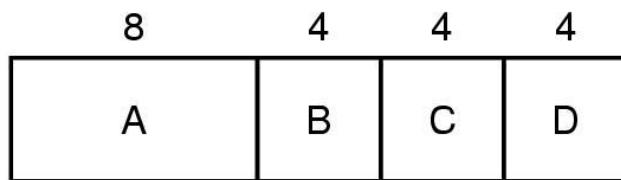
FCFS算法比较有利于长作业（进程），不利于短作业（进程）。

有利于CPU繁忙型作业（进程），不利于I/O繁忙型作业（进程）——因非抢占式

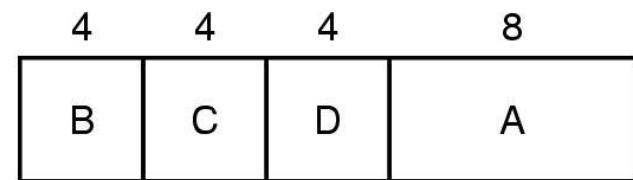
# 最短作业优先 (SJF, Shortest Job First)

- 对FCFS算法的改进，其目标是减少平均周转时间。
- 对预计执行时间短的作业（进程）优先分派处理机。通常后来的短作业不抢先正在执行的作业。
- 优点：比FCFS改善平均周转时间和平均带权周转时间，缩短作业的等待时间；提高系统的吞吐量；
- 缺点：对长作业非常不利，可能长时间得不到执行；未能依据作业的紧迫程度来划分执行的优先级；难以准确估计作业（进程）的执行时间，从而影响调度性能。

# Scheduling in Batch Systems (1)



(a)



(b)

An example of shortest job first scheduling

# 最短剩余时间优先SRT(Shortest Remaining Time)

- SJF的变型
- 允许比当前进程剩余时间更短的进程来抢占

# 高优先权优先调度算法

## 引入的目的：

为了照顾紧迫型作业，使之在进入系统后便获得优先处理。

## 适用范围：

- 批处理系统的作业调度
- 多种操作系统的进程调度
- 还适用于实时系统

优先权作业调度—— 系统从后备中选择一个或几个优先权最高的作业，将它调入内存运行。

优先权进程调度—— 系统将处理机分配给就绪队列中一个优先权最高的进程。

优先权的类型：静态优先权、动态优先权

# 优先权进程调度算法的类型

- 非抢占式优先权算法
- 抢占式优先权算法

## 非抢占式优先权算法——

系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直到完成，或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一个优先权最高的进程。

## 抢占式优先权算法——

系统把处理机分配给就绪队列中优先权最高的进程，使之执行，但在其执行期间，只要出现了另一个优先权更高的进程，系统就立即停止当前进程的执行，重新将处理机分配给新的优先权最高的进程。

它能更好地满足紧迫作业的要求。常用于实时系统中，以及对实时性能要求较高的批处理系统和分时系统中。

## 静态优先权

静态优先权的优缺点：

**优点：**简单易行，系统开销小。

**缺点：**优先权低的作业（进程）可能长期得不到调度。

### 1) 静态优先权

静态优先权——它是在创建进程时确定的，且在进程整个运行期间保持不变。

优先权一般用某一范围内一个整数来表示。

有的系统用“0”表示最高优先权，数值越大，优先权越低；有的系统恰恰相反。

确定优先权的依据：——常有三方面

- **进程类型** 系统进程（如接收进程、对换进程、磁盘I/O进程等）的优先权高于一般用户进程的优先权。
- **进程对资源的需求** 如进程的估计执行时间及内存需求量的多少，对这些要求少的进程赋予较高的优先权。
- **用户要求** 这是由用户进程的紧迫程度和用户所付费用的多少来确定优先权的。

## 动态优先权

动态优先权——在创建进程时所赋予的优先权，是可以随进程得推进，或随其等待时间的增加而改变的，以便获得更好的调度性。

例如：

- 在就绪队列中的进程，随其等待时间的增长，其优先权以速率  $\alpha$  提高；
- 在采用抢占式优先权调度算法时，如果再规定当前执行进程以速率  $\beta$  下降，则可防止一个长作业长期垄断处理机。

**UNIX**采用计算的方法动态改变进程的优先数。在**UNIX System V**版本中，进程优先数p\_pri的算式如下：

$$p_{pri} = p_{cpu}/2 + PUSER + p_{nice} + NZERO$$

其中，PUSER和NZERO是偏置常数，分别为25和20。p\_cpu和p\_nice是基本进程控制块中的两个项，分别表示进程使用处理器的情况和用户自己设置的计算优先数的偏置量。

系统对正在占用CPU的进程每隔一个时钟周期(20ms)对其p\_cpu加1。这使得占用处理器时间长的进程的p\_cpu值增大，其优先数也增大，优先权就相应降低。

系统每隔1s对所有进程执行p\_cpu/2，这使就绪进程优先级提高。

p\_nice的值允许用户根据任务的轻重缓急程度来设置，其值在0~39之间。一旦一个进程的p\_nice设置后，此后用户只能使其值增加。

**【例】**设有一个最多可有两道作业同时装入内存执行的批处理系统，作业调度采用最短作业优先调度算法，进程调度采用抢占式静态优先权调度算法，今有如下纯计算型作业序列（表中所列进程优先数中，数值越小优先权越高）：

作业名	到达时间	估计运行时间	进程优先数
J1	10:10	20分钟	5
J2	10:20	30分钟	3
J3	10:30	25分钟	4
J4	10:50	20分钟	6

- (1)列出所有作业进入内存时间及结束时间。
- (2)计算平均周转时间。

作业名	提交时间	进入时间	结束时间	周转时间
J1	10: 10	10: 10	11: 00	50分钟
J2	10: 20	10: 20	10: 50	30分钟
J3	10: 30	11: 00	11: 25	55分钟
J4	10: 50	10: 50	11: 45	55分钟

$$\text{平均周转时间} = (50+30+55+55)/4 = 47.5(\text{分钟})$$

# 高响应比优先调度算法

响应比 = (等待时间+要求服务时间)/要求服务时间

(实际上响应比是动态优先权)

高响应比优先调度算法——

每次要进行作业调度时，系统首先计算后备队列中各作业的响应比，然后选择一个或若干个响应比最高的作业调入内存执行。

该算法综合了FCFS和SJF算法的**优点**——既考虑公平性，又考虑平均周转时间

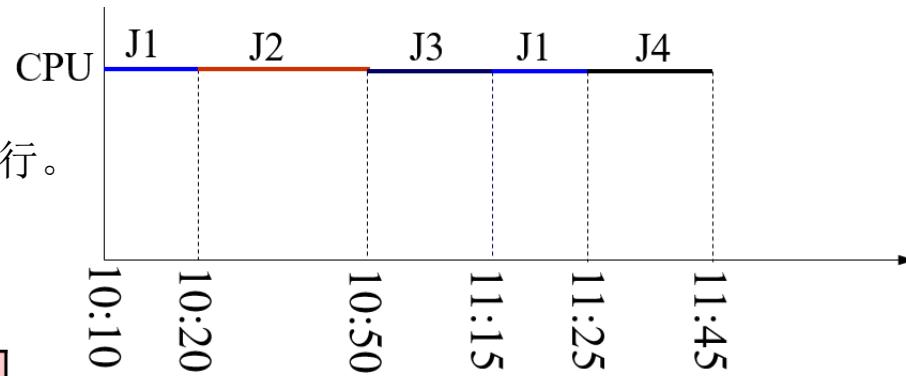
**缺点**是会增加系统开销——每次调度都要计算响应比。

**【例】**设有一个最多可有两道作业同时装入内存执行的批处理系统，作业调度采用高响应比优先调度算法，进程调度采用抢占式静态优先权调度算法，今有如下纯计算型作业序列（假设表中所列进程优先数中，数值越小优先权越高）：

- (1) 列出所有作业进入内存时间及结束时间。
- (2) 计算平均周转时间。

作业名	到达时间	估计运行时间	进程优先数
J1	10:10	20分钟	5
J2	10:20	30分钟	3
J3	10:30	25分钟	4
J4	10:50	20分钟	6

- 10:10 只有J1一个作业，调入内存执行。  
 10:20 J2到达，调入内存，因其优先级高，J2执行。  
 10:50 J3响应比高，J3调入内存并执行。  
 11:15 J4调入内存，J1恢复执行。

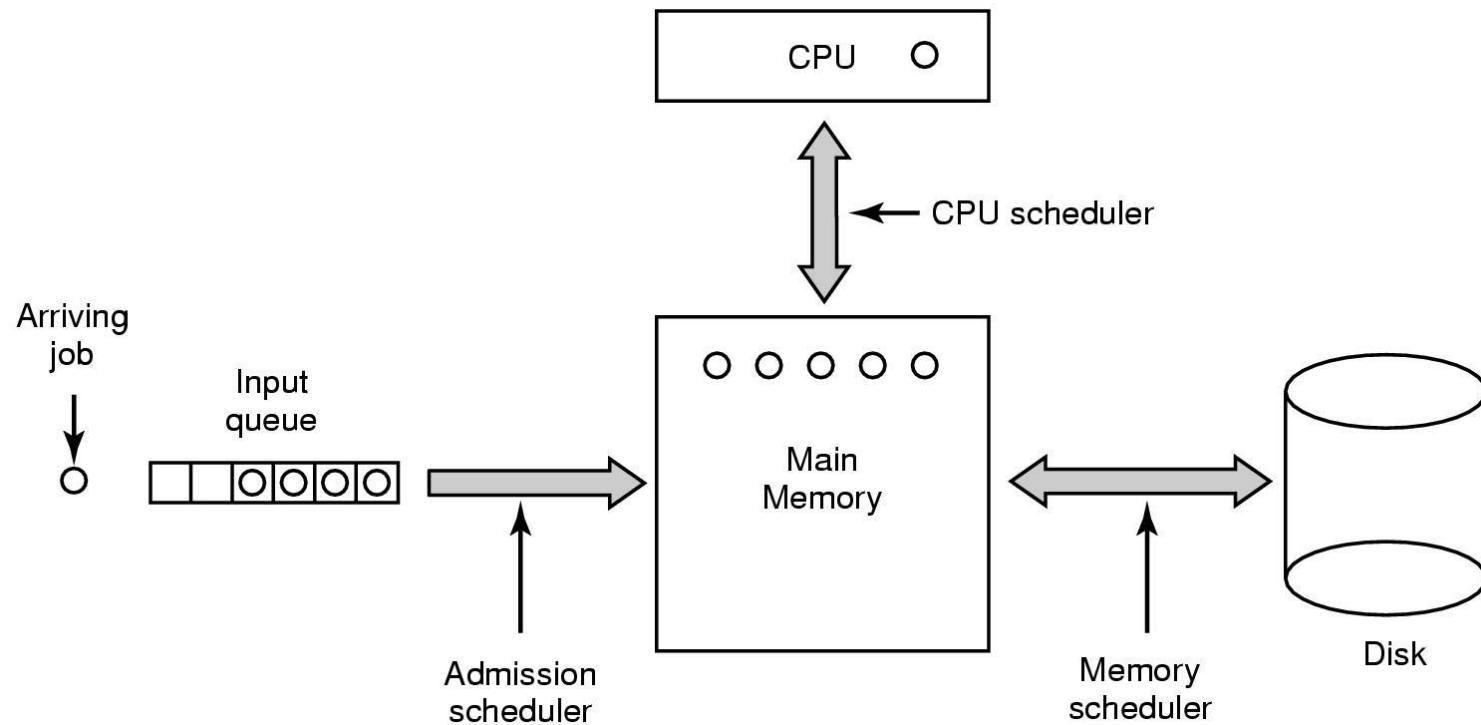


作业名	到达时间	调入时间	结束时间	周转时间
J1	10:10	10:10	11:25	75分钟
J2	10:20	10:20	10:50	30分钟
J3	10:30	10:50	11:15	45分钟
J4	10:50	11:15	11:45	55分钟

平均周转时间  

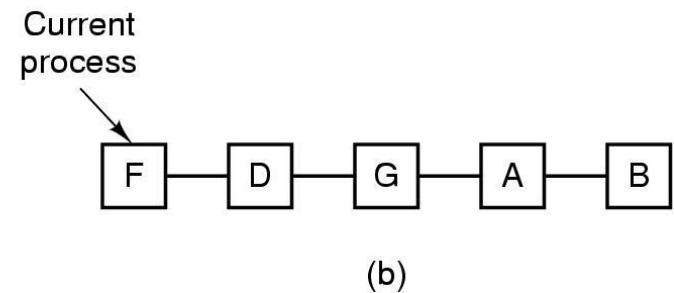
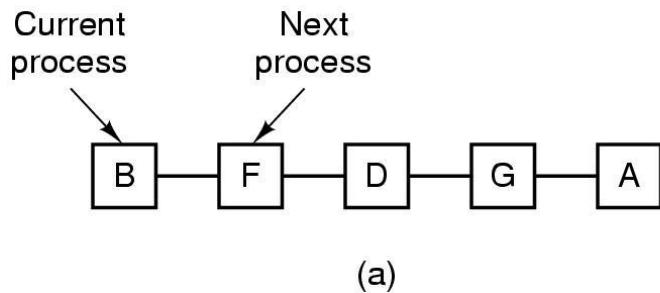
$$=(75+30+45+55)/4=51.25(\text{分钟})$$

# Scheduling in Batch Systems (2)



Three level scheduling

# Scheduling in Interactive Systems (1)



- Round Robin Scheduling
  - list of runnable processes
  - list of runnable processes after B uses up its quantum

# 时间片轮转调度

- 核心思想：
  - 每个进程运行固定的时间片，然后调入下一个进程
- 实现机理：
  - 维护就绪进程队列，采用FIFO方式一次读取
- 特殊控制：
  - 时间片内发生阻塞或结束，则立即放弃时间片
- 优缺点分析
  - 优点：绝对公平
  - 缺点：公平即合理吗？时间片如何设计才能保证效率？

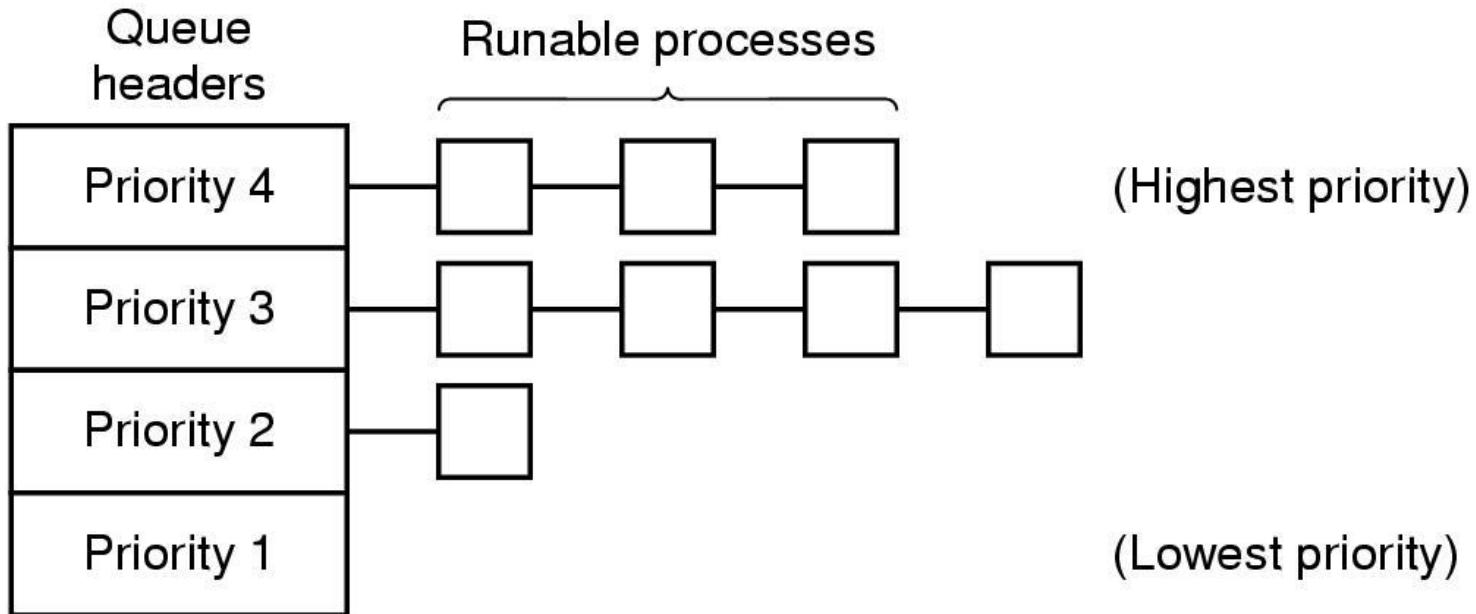
# 优先级调度

- 核心思想：
  - 为每个进程赋予不同级别的优先级，越高越优先
- 实现机理：
  - 维护一个优先级队列，自顶向下依次读取
- 特殊控制：
  - 静态优先级与动态优先级概念
- 优缺点分析
  - 优点：响应时间快，易于调整。最通用的方法。
  - 缺点：死规则，如何保证周转时间和吞吐量？

# 多级队列算法 (Multiple-level Queue)

- 引入多个就绪队列，通过各队列的区别对待，达到一个综合的调度目标；
- 根据作业或进程的性质或类型的不同，将就绪队列再分为若干个子队列。
- 每个作业固定归入一个队列。
- 各队列的不同处理：不同队列可有不同的优先级、时间片长度、调度策略等。如：系统进程、用户交互进程、批处理进程等。

# Scheduling in Interactive Systems (2)

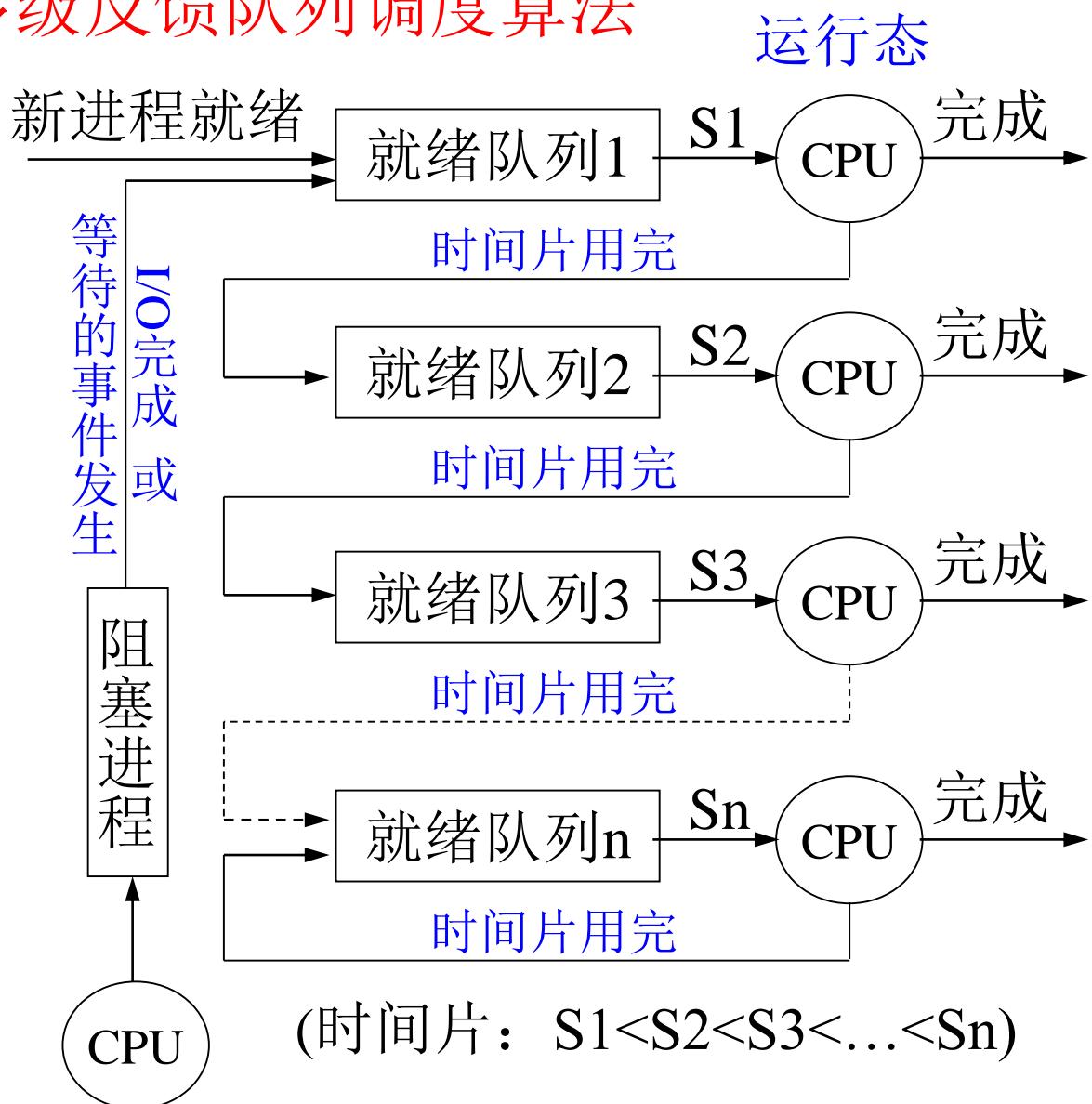


A scheduling algorithm with four priority classes

# 多级反馈队列调度算法

不必事先知道各进程所需执行时间，而且还可以满足各种类型的需要。

目前被公认的一种较好的进程调度算法。



# 最短进程优先

- 核心思想：
  - 保证响应时间最快、平均周转时间最短
- 实现机理：
  - 依据先验信息，将进程按照运行时间增序调度
  - 估计值(aging)： $T_0, T_0/2+T_1/2, T_0/4+T_1/4+T_2/2, T_0/8+T_1/8+T_2/4+T_3/2,$
- 特殊控制：
  - 如何确定最短作业？（老化算法）
- 优缺点分析
  - 优点：保证了CPU的利用效率
  - 缺点：无法通用，约束条件多。

**【例】**设有一个最多可有两道作业同时装入内存执行的批处理系统，作业调度采用高响应比优先调度算法，进程调度采用时间片轮转调度算法（假设时间片为100ms），今有如下纯计算型作业序列：

作业名	到达时间	估计运行时间
J1	10:10	20分钟
J2	10:20	30分钟
J3	10:30	25分钟
J4	10:50	20分钟

- (1)列出所有作业进入内存时间及结束时间。
- (2)计算平均周转时间。

先分析：

- 10:10 J1到达， 调入内存执行
- 10:20 J2到达， 调入内存与J1一起均分CPU运行。 J1已运行10分钟， 还需与J2一起运行20分钟
- 10:30 J3到达， 不调入内存
- 10:40 J1结束， J3调入内存与J2一起均分CPU运行。  
J2已运行10分钟， 还需与J3一起运行40分钟
- 10:50 J4到达， 不调入内存
- 11:20 J2结束， J4调入内存与J3一起均分CPU运行。  
J3已运行20分钟， 还需与J4一起运行10分钟
- 11:30 J3结束， J4还需单独运行15分钟
- 11:45 J4结束

解：(1)各作业进入内存时间及结束时间如下表所示。

作业名	调入时间	结束时间	周转时间
J1	10:10	11:40	30分钟
J2	10:20	11:20	60分钟
J3	10:40	11:30	60分钟
J4	11:20	11:45	55分钟

(2)平均周转时间=(30+60+60+55)/4=51.25(分钟)

**【例】**有一个多道程序设计系统，采用不可移动的可变分区方式管理主存空间，设主存空间为100K，采用最先适应分配算法分配主存，作业调度采用响应比高者优先算法，进程调度采用时间片轮转算法(即内存中的作业均分CPU时间)，今有如下作业序列：

作业名	提交时间	需要执行时间	要求主存量
J1	10 : 00	40分钟	25K
J2	10 : 15	30分钟	60K
J3	10 : 30	20分钟	50K
J4	10 : 35	25分钟	18K
J5	10 : 40	15分钟	20K

假定所有作业都是计算型作业且忽略系统调度时间。回答下列问题：

- (1) 列表说明各个作业被装入主存的时间、完成时间和周转时间；
- (2) 写出各作业被调入主存的顺序；
- (3) 计算5个作业的平均周转时间。

解：通过分析([在黑板上分析](#))，可得如下结果：

- (1) 各个作业被装入主存的时间、完成时间和周转时间如下表所示：

作业名	装入主存时间	作业完成时间	周转时间
J1	10: 00	11: 05	65
J2	10: 15	11: 15	60
J3	11: 15	11: 55	85
J4	11: 35	12: 10	95
J5	11: 05	11: 35	55

- (2) 作业被调入主存的顺序为J1,J2,J5,J3,J4。

- (3) 平均周转时间= $(65+60+85+95+55)/5=72$ (分钟)

**【例】**多道批处理系统中配有一个处理器和2台外设(D1和D2)，用户存储空间为100MB。已知系统采用可抢占式的高优先数调度算法和不允许移动的可变分区分配策略，设备分配按照动态分配原则。今有4个作业同时提交给系统，如下表所示。

作业运行时间和I/O时间按下述顺序进行：

- A. CPU(1分钟), D1(2分钟), D2(2分钟)
- B. CPU(3分钟), D1(1分钟)
- C. CPU(2分钟), D1(3分钟), CPU(2分钟)
- D. CPU(4分钟), D1(2分钟)

忽略其他辅助操作，求4个作业的平均周转时间是多少分钟。

作业名	优先数	运行时间	内存需求
<b>A</b>	6	5分钟	50M
<b>B</b>	3	4分钟	10M
<b>C</b>	8	7分钟	60M
<b>D</b>	4	6分钟	20M

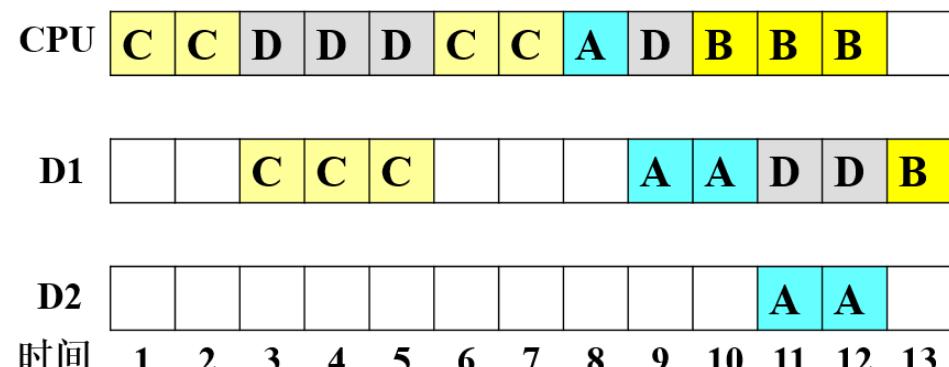
解答：参考上图

A的周转时间为12分钟

B的周转时间为13分钟

C的周转时间为7分钟

D的周转时间为12分钟



所以平均周转时间为 $(12+13+7+12)/4=11$ (分钟)

# 其它调度算法

- 保证调度：  $n$ 个用户， 某个用户得到 $1/n$ 的CPU能力
- 公平分享调度： 以用户而不是进程为考虑原则

# 彩票调度算法

- 核心思想:
  - 保证响应速度最快、依据进程对CPU资源的需求量调度
- 实现机理:
  - 维护定量的彩票，不同进程获取数量不同，随机选择
- 特殊控制:
  - 彩票交换：进程间可以动态自主调节调度顺序
- 优缺点分析
  - 优点：响应速度最快、CPU利用率最高
  - 缺点：OS实现机制复杂、吞吐量和周转时间无法保证

# 实时调度 Scheduling in Real-Time Systems

实时系统中，硬实时任务(存在着必须满足的时间限制)和软实时任务(偶尔超过时间限制是可以容忍的)都联系着一个截止时间。为了保证系统能正常工作，实时调度必须满足实时任务对截止时间的要求。

## Schedulable real-time system

- Given
  - $m$  periodic events
  - event  $i$  occurs within period  $P_i$  and requires  $C_i$  seconds
- Then the load can only be handled if

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

# 实时调度需要的信息

系统应向调度程序提供有关任务的下述信息：

- (1) 就绪时间。这是该任务成为就绪状态的起始时间
- (2) 开始截止时间和完成截止时间。对于典型的实时应用，只需知道开始截止时间，或者知道完成截止时间。
- (3) 处理时间。一个任务从开始执行直至完成所需的时间。在某些情况下，该时间也是系统提供的。
- (4) 资源要求。
- (5) 优先级。若某任务的开始截止时间已经错过，就会引起故障，则应赋予该任务“绝对”优先级；如果对系统的继续运行无重大影响，则可赋予“相对”优先级，供调度参考。

例如

设系统中有**6**个硬实时任务，它们的周期都是**50ms**，而每次的处理时间都是**10ms**，此时不能满足上式，因而系统是不可调度的。

又例如

一个实时系统中有**3**个实时事件流，其周期分别为**100ms**、**200ms**和**500ms**，每次的处理时间分别为**50ms**、**30ms**和**100ms**，则因为

$$0.5 + 0.15 + 0.2 \leq 1$$

故系统是可调度的。如果加入周期为**1**秒的第**4**个任务，则只要其处理时间不超过**150ms**，系统仍是可调度的。

当然，上述运算的隐含条件是进行切换的时间足够小，可以忽略。

# 实时调度:抢占式 / 非抢占

- 含有硬实时任务的实时系统中，广泛采用**抢占机制**。这样可满足硬实时任务对截止时间的要求，但这种调度机制比较复杂。
- 对于一些小的实时系统，如果能预知任务的开始截止时间，则可采用**非抢占调度机制**以简化调度程序和对任务调度所花费的系统开销。此时，应使所有的实时任务都比较小，并在执行完关键性程序和临界区后，能及时地把自己阻塞起来，以便释放处理机，供调度程序去调度那种开始截止时间即将到达的任务。
- 具有**快速切换机制**: 为了保证要求较高的硬实时任务能及时运行，在实时系统中还应具有快速切换机制，以保证能进行任务的快速切换。
  - 对外部中断的快速响应能力。
  - 快速的任务分派能力。

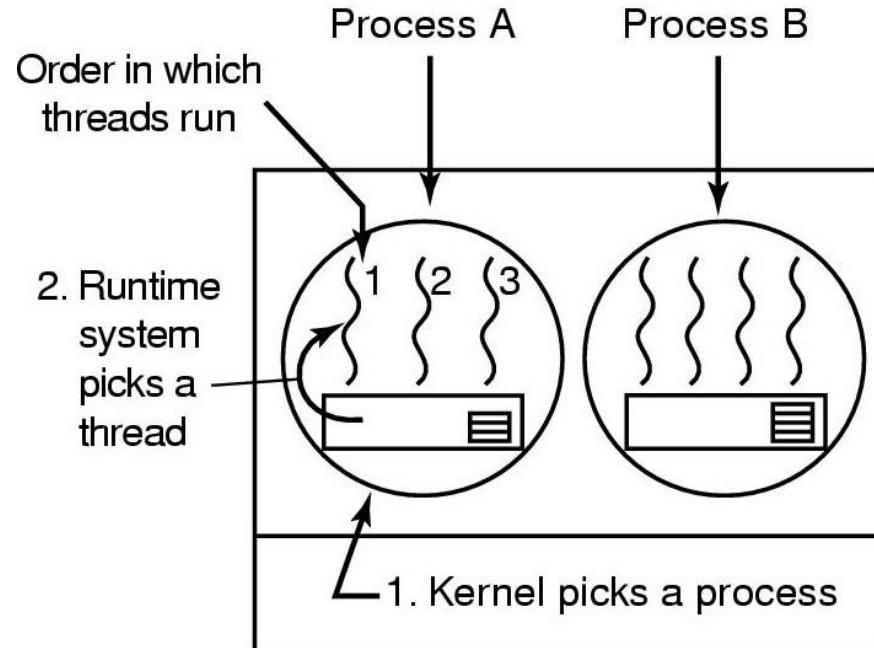
# 实时调度算法的分类

- 按实时任务性质不同来划分：
  - 硬实时调度算法
  - 软实时调度算法
- 按调度方式的不同
  - 非抢占调度算法
  - 抢占调度算法
- 因调度程序调度时间的不同
  - 静态调度算法：进程执行前，调度程序便已经决定了个进程间的执行顺序
  - 动态调度算法
- 多处理机环境下
  - 集中式调度
  - 分布式调度

# Policy versus Mechanism

- 调度机制
  - 不同调度算法适用于不同环境和不同目的
  - 调度算法一旦固定，则其最优、最坏情况均无法避免
  - 如能根据具体情况动态调整，则效果更佳
- 调度策略
  - 为用户提供改变调整调度机制的渠道
  - 实现方法——提供系统调用，能够改变调度机制

# Thread Scheduling (1)



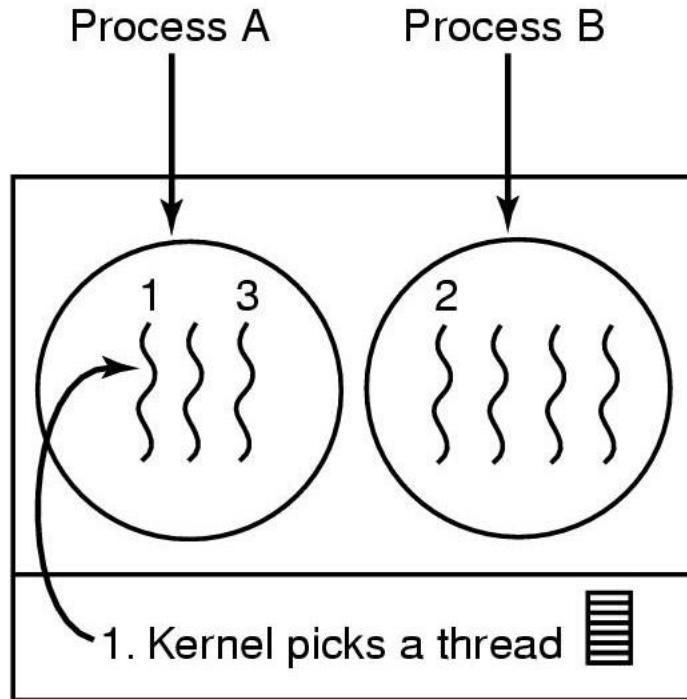
Possible: A1, A2, A3, A1, A2, A3

Not possible: A1, B1, A2, B2, A3, B3

## Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst

# Thread Scheduling (2)



Possible: A1, A2, A3, A1, A2, A3

Also possible: A1, B1, A2, B2, A3, B3

## Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec/CPU burst