Chapter 2

Processes and Threads

2.1 Processes
2.2 Threads
2.3 Interprocess communication
2.4 Classical IPC problems
2.5 Scheduling

PROCESSES

- The Process Model
- Process Creation
- Process Termination
- Process Hierarchies
- Process States
- Implementation of Processes

What is a process?

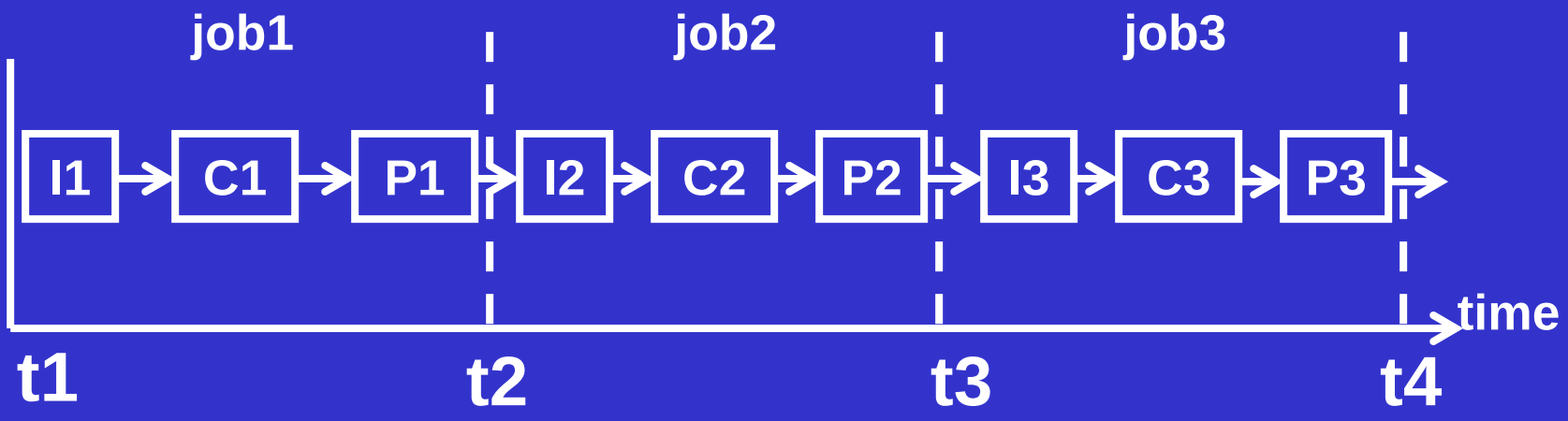
A conceptual model that makes parallelism easier to deal with.

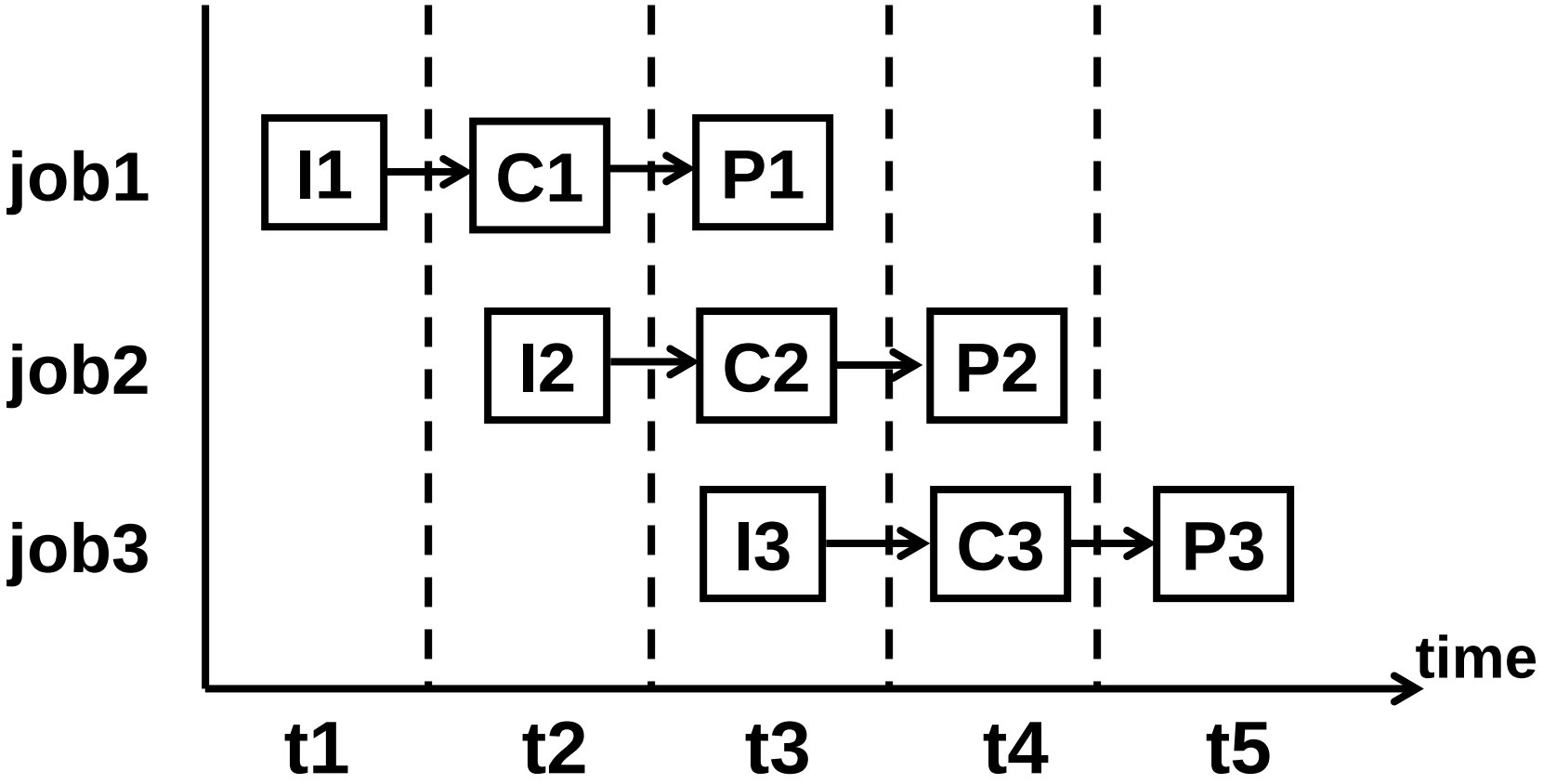
In early, there is one processor, one user, one job.

By raising the efficient of processor, we are working at one processor, more user, more jobs.

Parallelism is applied.

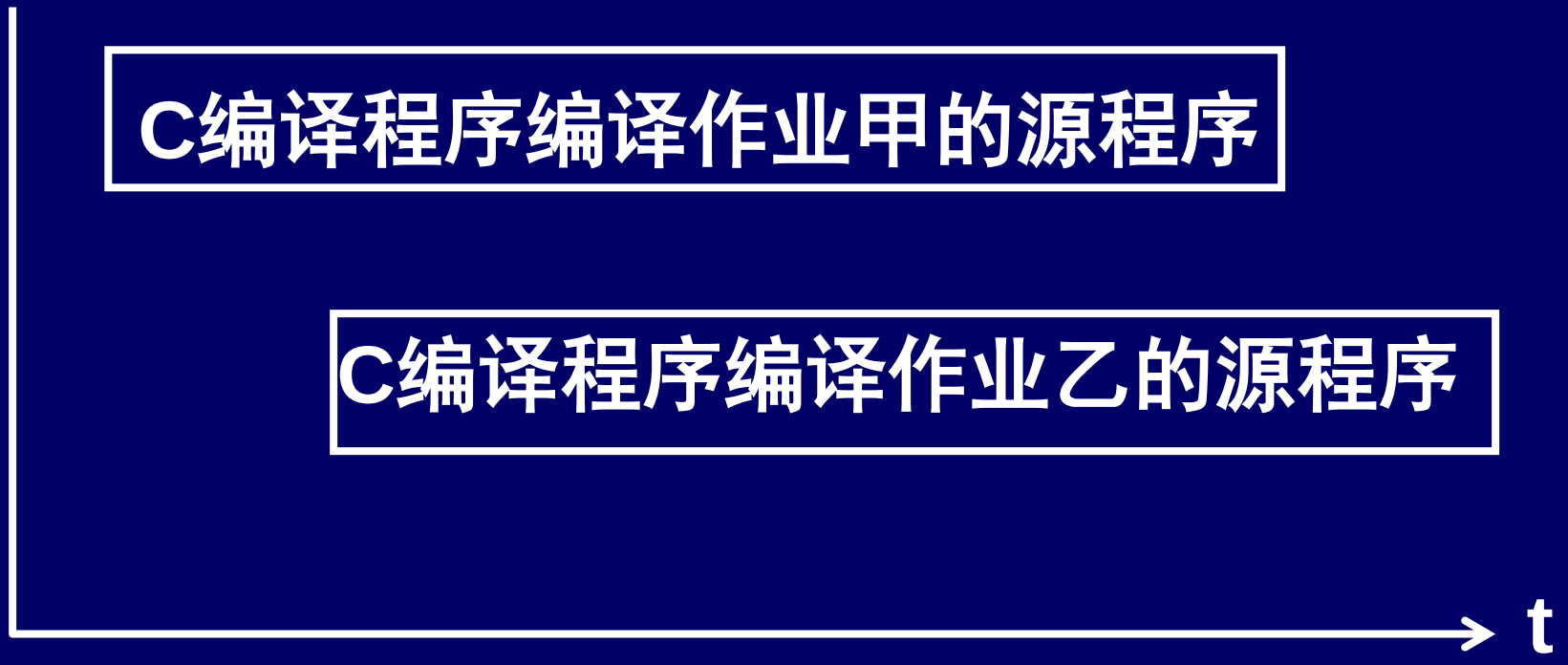
Or be called pseudo-parallelism for one processor.


job1
job2
job3
I1
C1
P1
I2
C2
P2
I3
C3
P3
time
t1
t2
t3
t4


job1
job2
job3
I1
C1
P1
I2
C2
P2
I3
C3
P3
time
t1
t2
t3
t4
t5

```
begin integer N;
    N:=0;
    cobegin
        program A:
            begin
                L1: ……;
                 N:=N+1;
                 goto L1;
            end;
```

```
        program B:
            begin
                L2: ……;
                print(N);
                N:=0;
                goto L2;
            end;
    coend;
end;
```

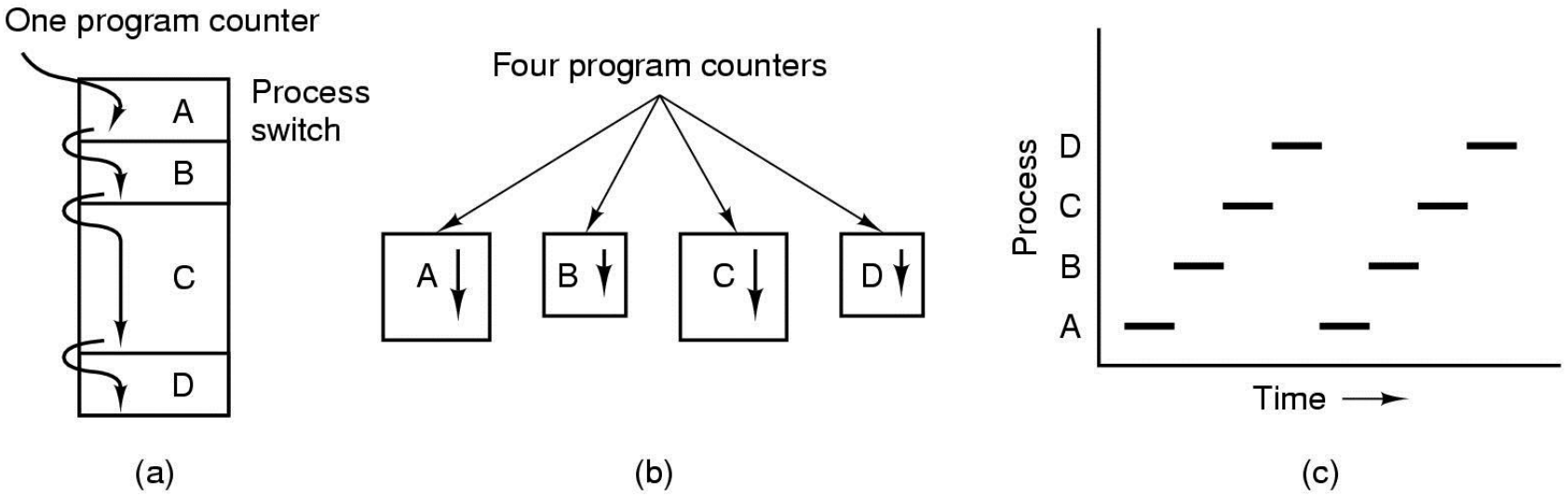

C编译程序编译作业甲的源程序
C编译程序编译作业乙的源程序
t

Processes

A process is just an executing program.

进程是一个具有一定独立功能的程序关于某个数据集合的一次运行活动，它是系统进行资源分配和调度的一个独立单位。

The Process Model



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

The Difference Between Process and Program

- Process is dynamic, and the program is static
- Process is temporary, the program is permanence
- The elements of Process and program is different(code,data,PCB)
- The relationships of Process and program are complex

Process Creation

Principal events that cause process creation

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 Termination

Conditions which terminate processes

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

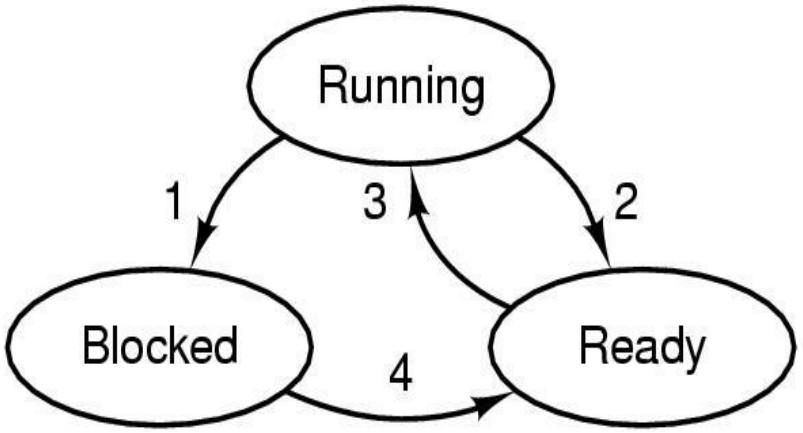
Process Hierarchies

- 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

进程与作业的关系

- 作业是用户向计算机提交任务的任务实体；
  进程是完成用户任务的执行实体。
- 作业与进程的对应关系：一个作业可由多个进程组成，反过来不成立。
- 作业的概念主要用在批处理系统中；
  进程的概念主要用在几乎所有的多道系统中。

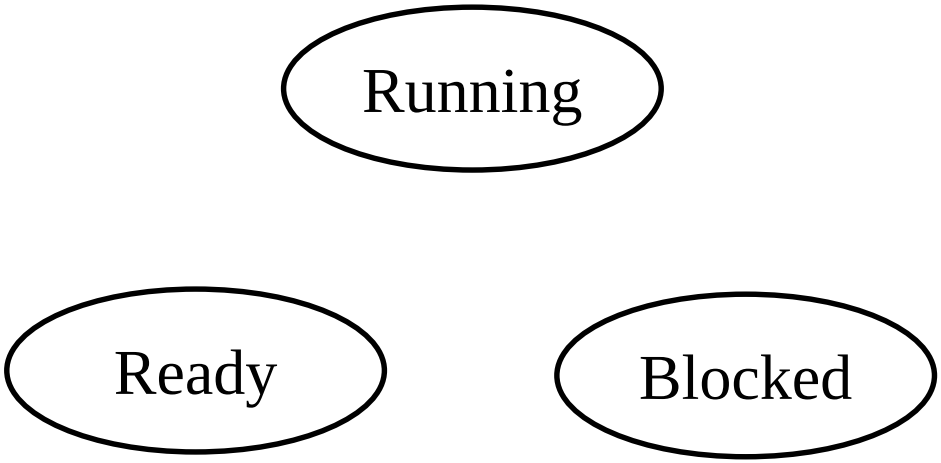
Process States (1)



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

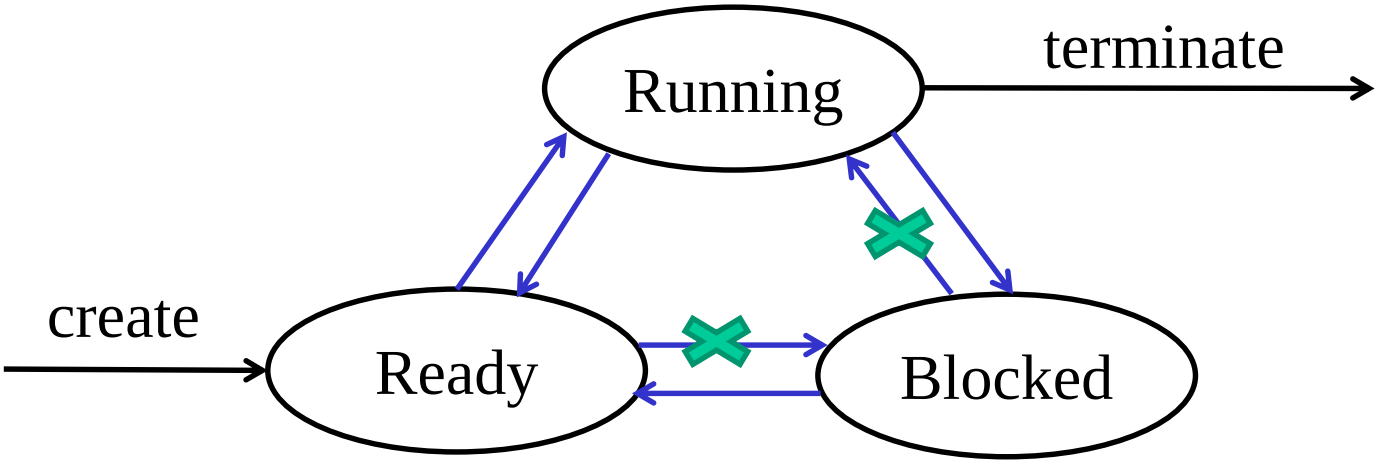
- Possible process states
  - running
  - Ready
  - blocked
- Transitions between states shown

Process States



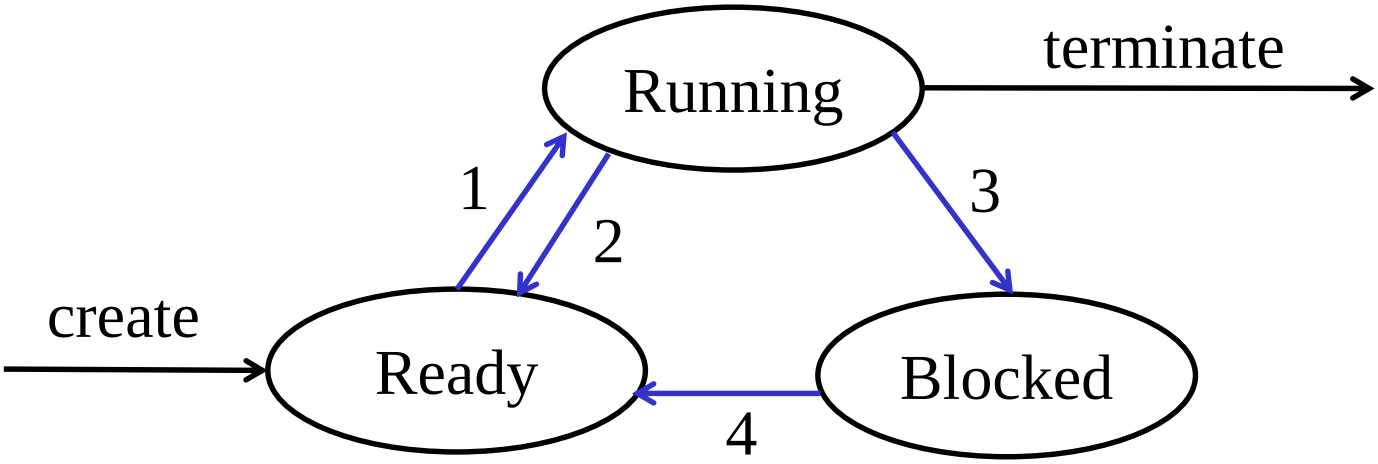
- Possible process states
  - Running
  - Ready
  - Blocked

Process States



- Transitions between states shown

Process States

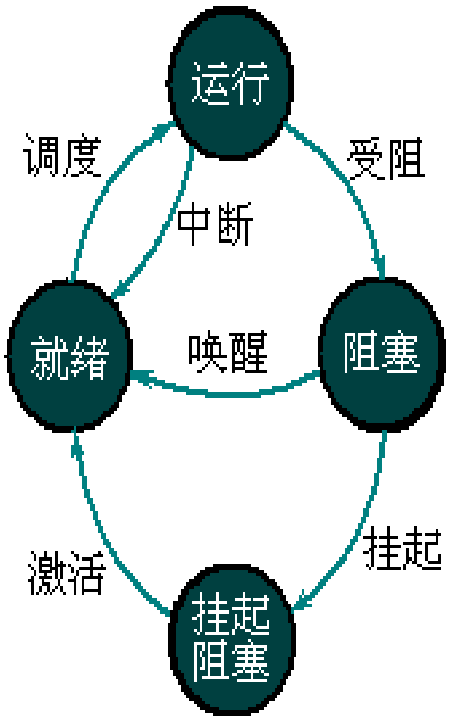


- Transitions between states shown
  1. Scheduler picks this process
  2. Scheduler picks another process
  3. Process blocks for input
  4. Input becomes available

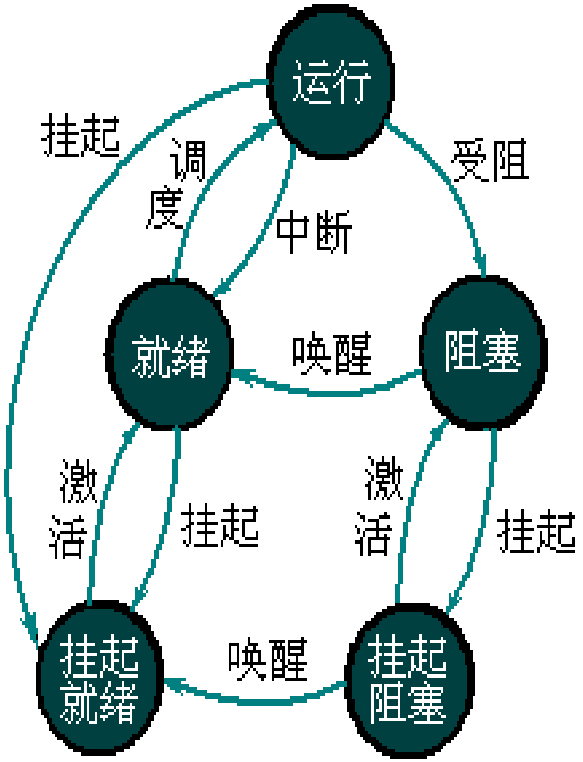
当前系统并发10个进程，问处于各种状态的进程分别最多和最少的个数。

- Ready: ~
- Running: ~
- Blocked: ~

含有挂起状态的进程变迁图

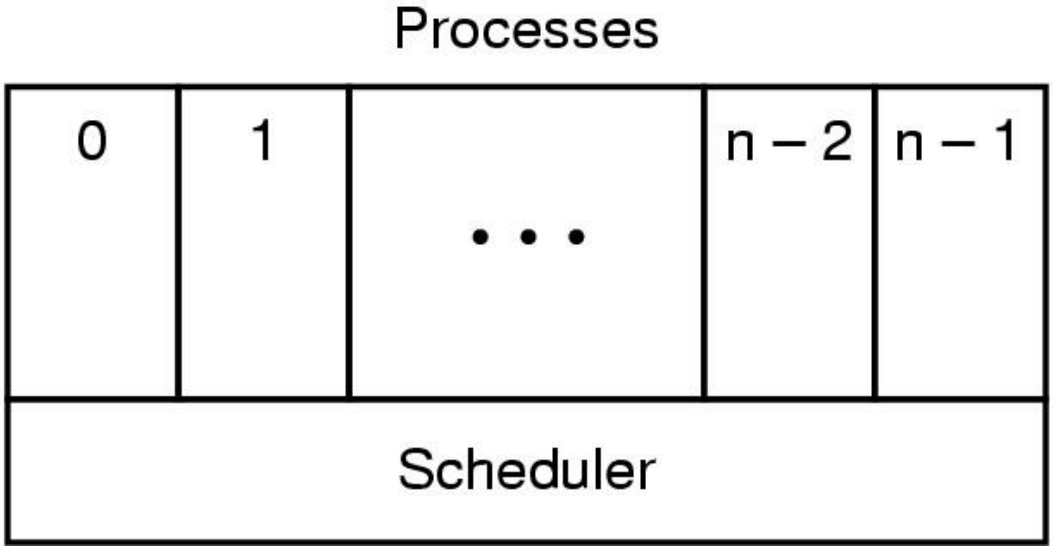


（a）一个挂起状态



（b）两个挂起状态

Process States (2)



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

Implementation of Processes (1)

- Elements: code + data + ?

  Process Table (Process Control Block)

  – A memory block which record the information of process, allocated and maintained by OS
- OS creates process, allocates memory, I/O devices, files, and so on to user program.

  + Create the PCB(the number is limited)
- Always kept in OS kernel, visited by using system call

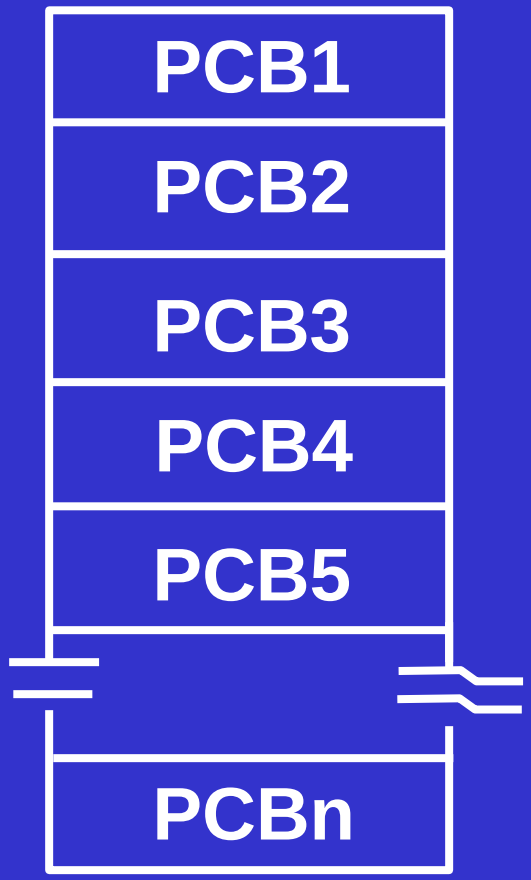
Implementation of Processes (2)

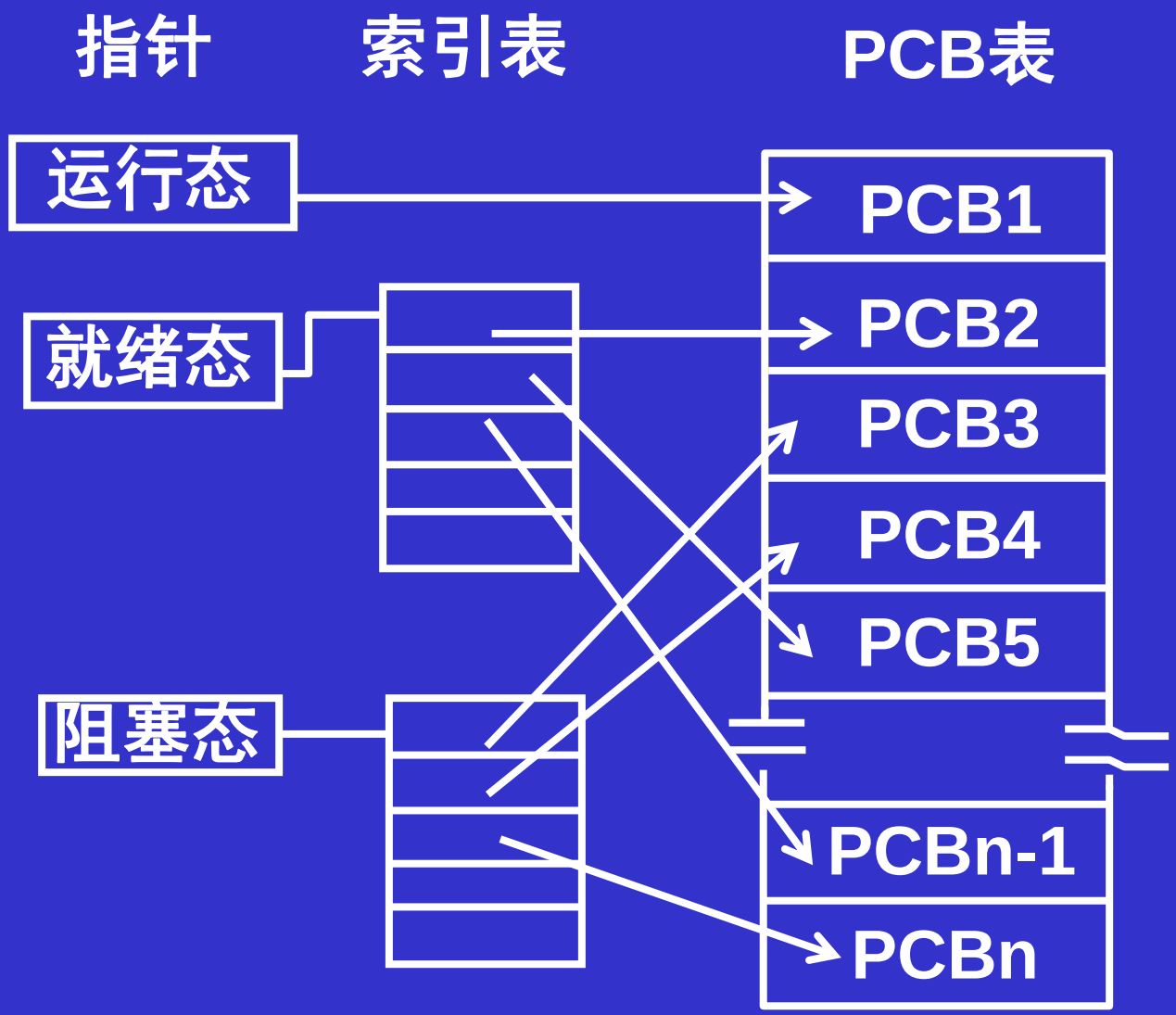
Process management	Memory management	File management
Registers	Pointer to text segment	Root directory
Program counter	Pointer to data segment	Working directory
Program status word	Pointer to stack segment	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters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

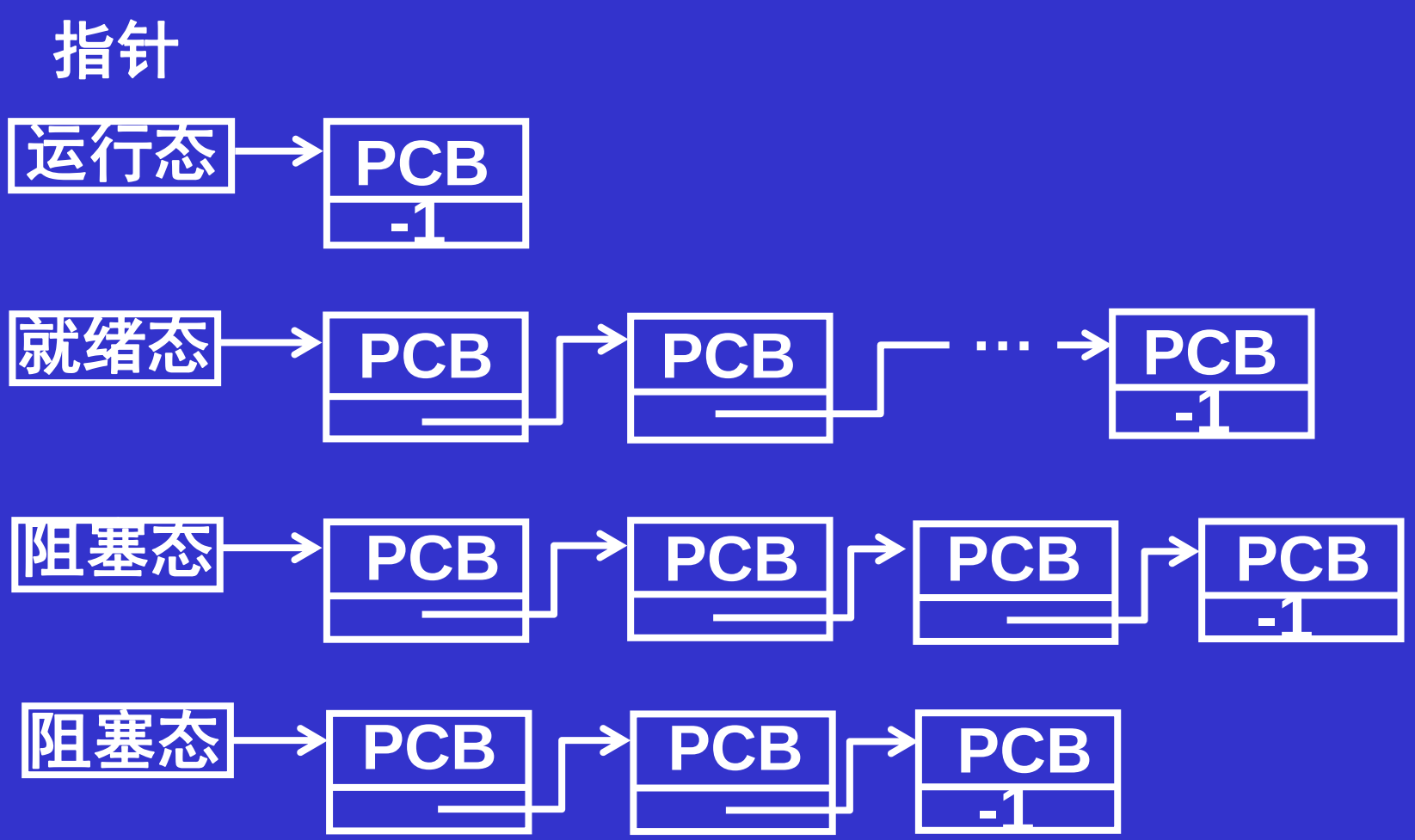
Fields of a process table entry

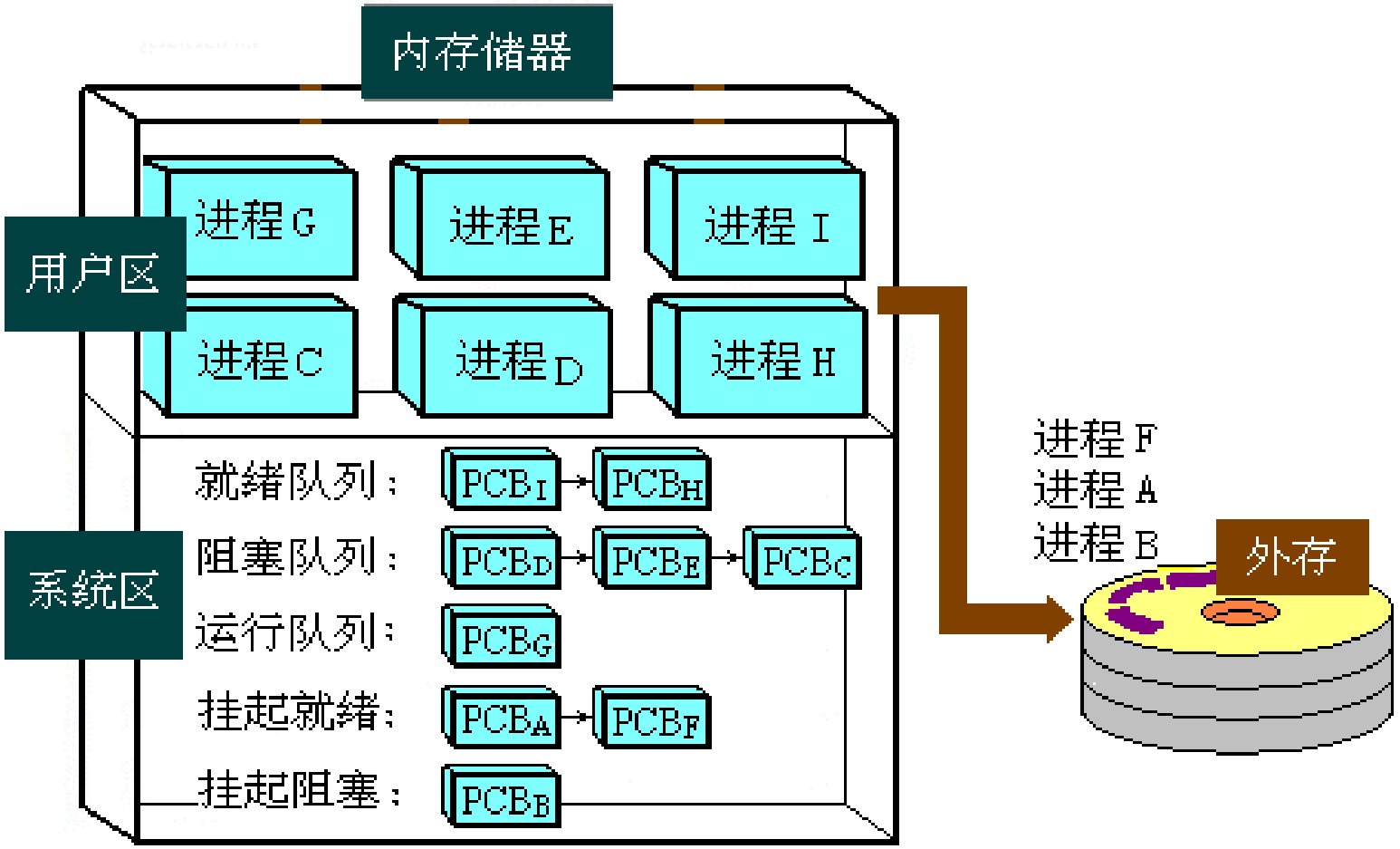
- 为了便于对进程进行管理，系统把各个进程的PCB集中存放在内存的指定区域，并组织在一起形成PCB表。
- 不同的操作系统采用不同的PCB表组织结构，PCB表的物理组织结构直接关系到系统的效率。
- 常用的有三种：线性表结构、索引表结构和链接表结构。

PCB表

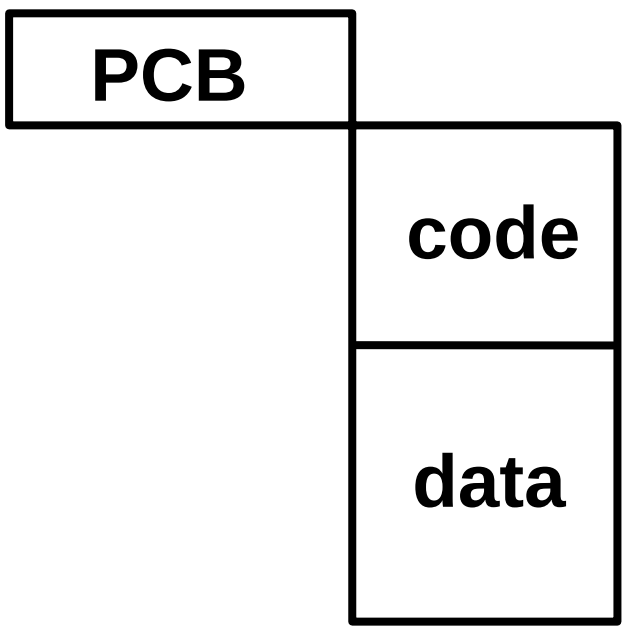



指针
索引表
PCB表
运行态
就绪态
阻塞态
PCB1
PCB2
PCB3
PCB4
PCB5
PCBn-1
PCBn

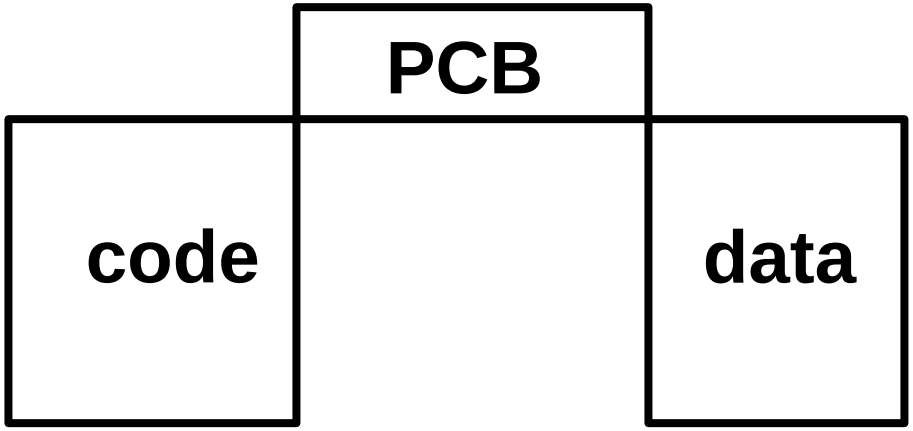

指针
运行态
PCB
-1
就绪态
PCB
PCB
…
PCB
-1
阻塞态
PCB
PCB
PCB
PCB
-1
阻塞态
PCB
PCB
PCB
-1


内存储器
用户区
进程G
进程E
进程I
进程C
进程D
进程H
系统区
就绪队列：
PCBI
PCBH
阻塞队列：
PCBD
PCBE
PCBC
运行队列：
PCBG
挂起就绪：
PCBA
PCBF
挂起阻塞：
PCBB
进程F
进程A
进程B
外存

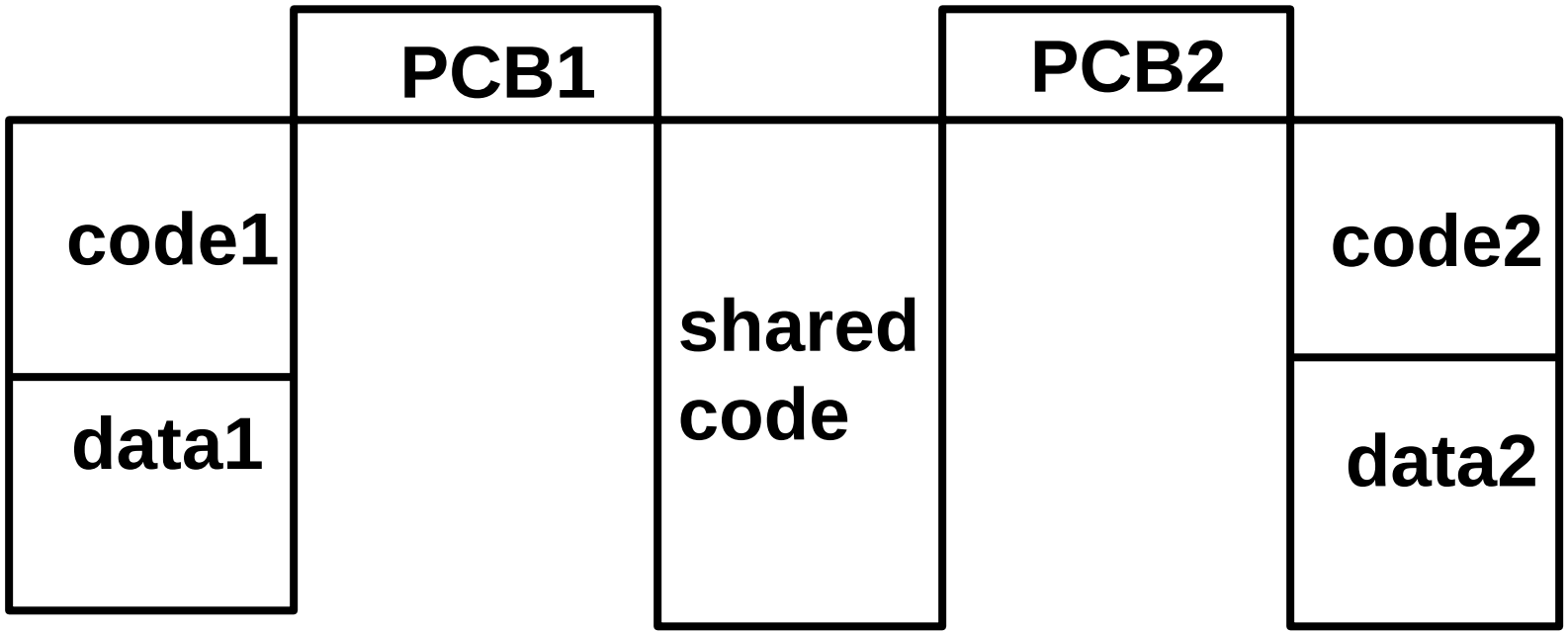
Space of Processes



(A)

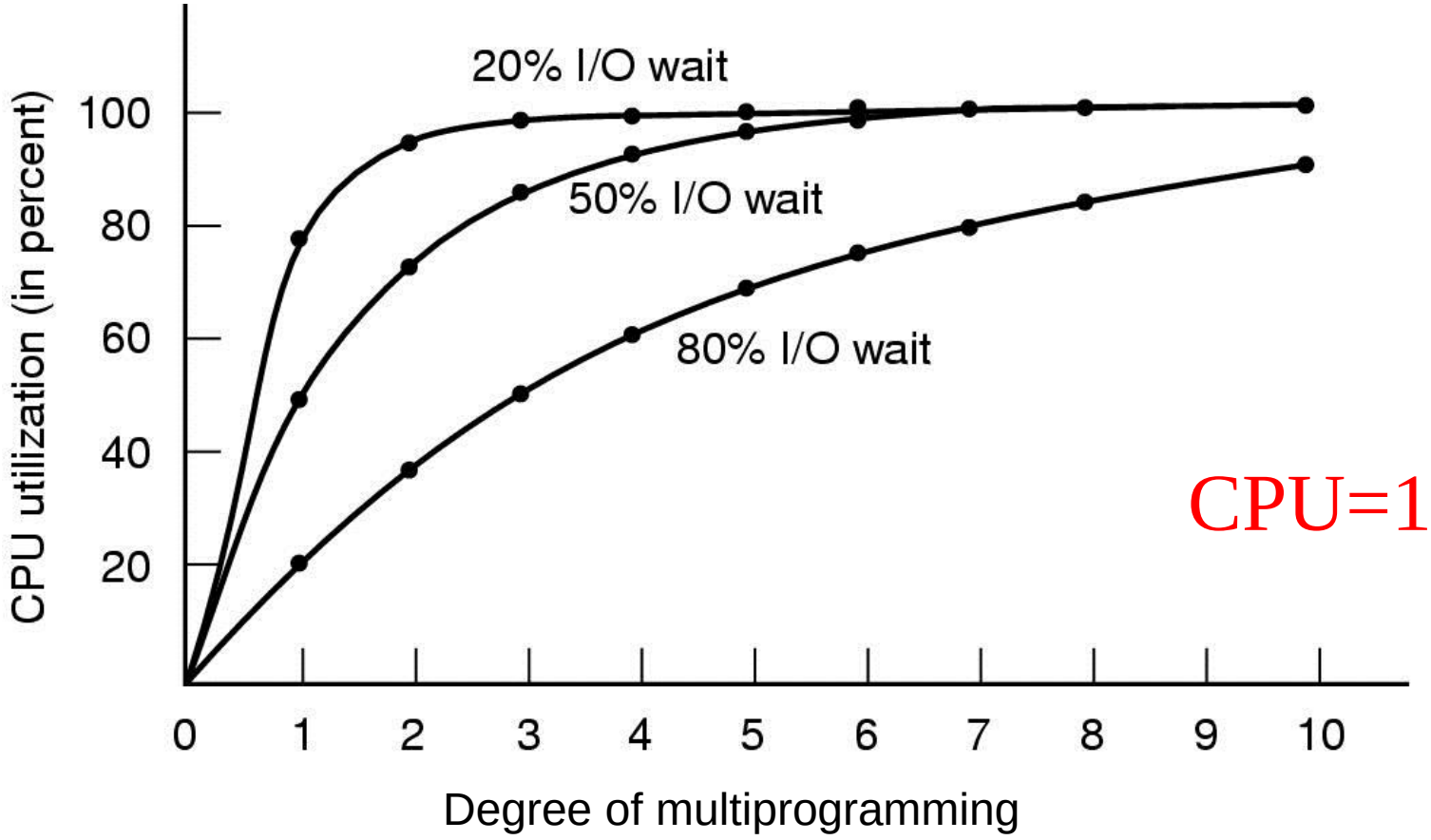


(B)



(C)

Modeling Multiprogramming



$CPU=1-p^n$

CPU utilization as a function of number of processes in memory

Example:
A computer has 8GB of memory, with OS taking up 2GB and each user program also taking up 2GB. Three user program to be in memory at once. With 80% average I/O wait, we have a CPU utilization of $1 - 0.8^3 = 49\%$.

- Adding another 8GB of memory allows the system to go from 3 to 7, thus raising the CPU utilization to $1 - 0.8^7 = 79\%$.

  In another word, the additional 8GB will raise the throughput by 30%.

- Adding yet another 8GB would increase CPU utilization only from 79% to $1 - 0.8^{11} = 91\%$.

  Thus raising the throughput by only another 12%.

Analysis of Multiprogramming System Performance

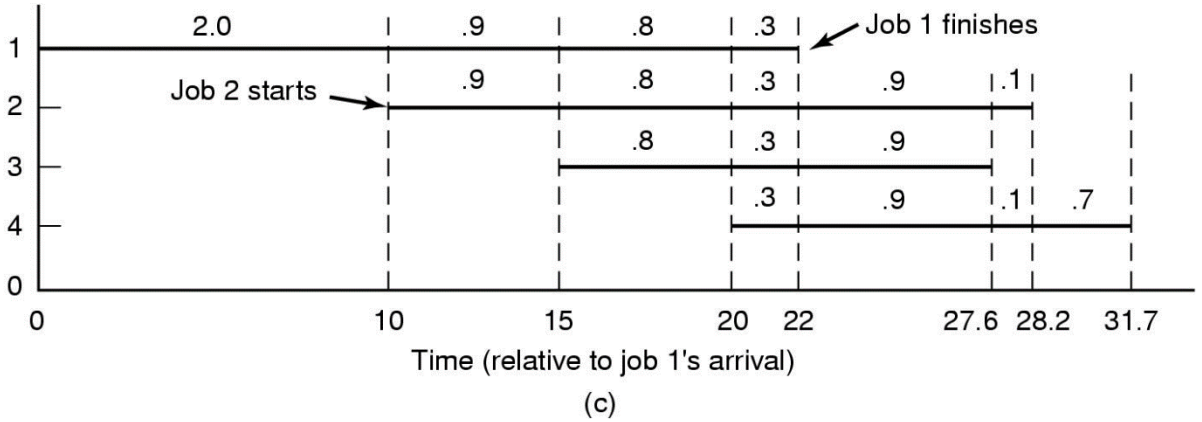
Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes			
	1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)

总CPU利用率提高明显，单个进程降低幅度不大。



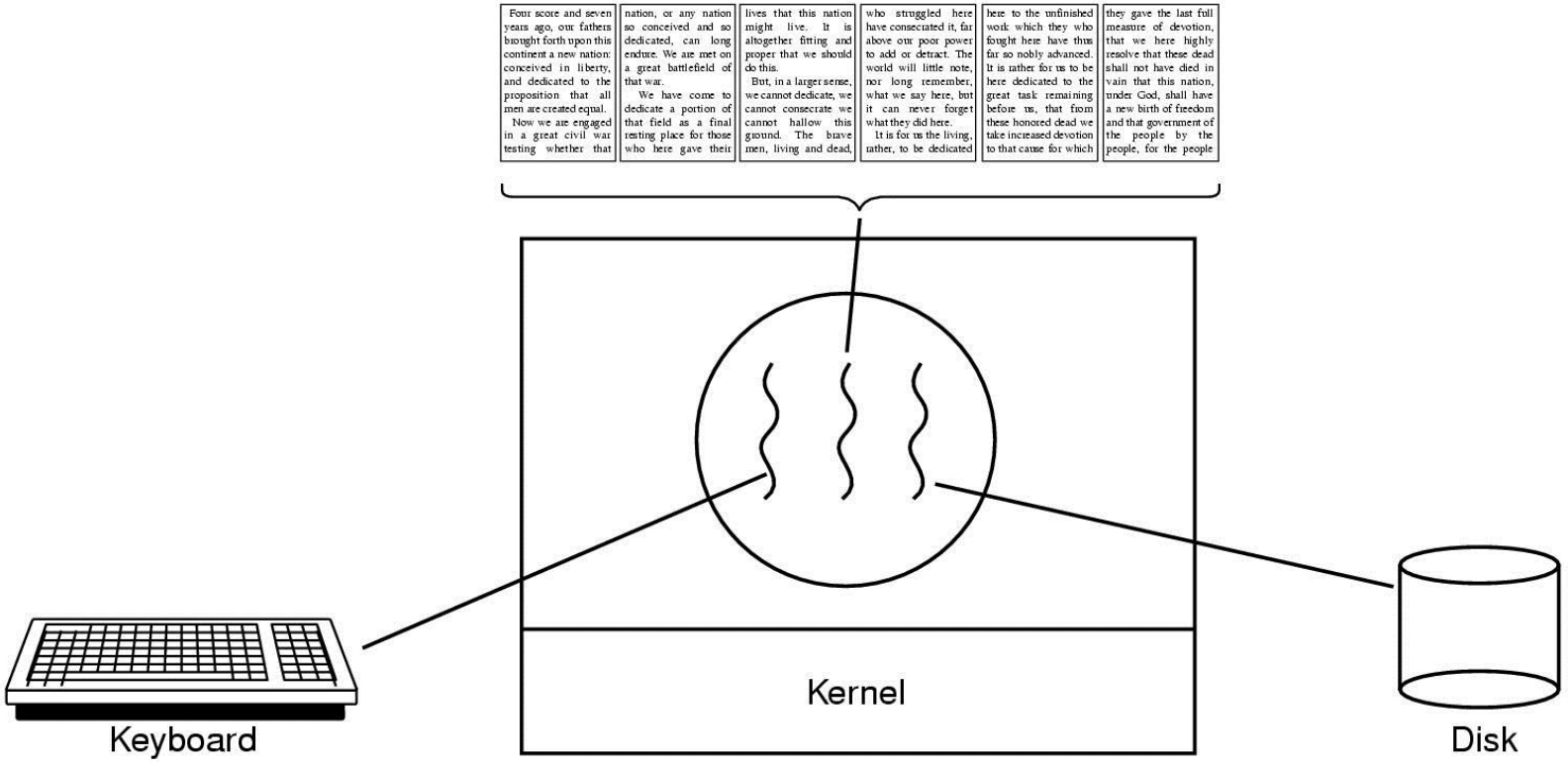
(c)

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - note numbers show amount of CPU time jobs get in each interval

THREADS

- The Thread Model
- Thread Usage
- Implementing Threads in User Space
- Implementing Threads in the Kernel
- Hybrid Implementations
- Scheduler Activations
- Pop-Up Threads
- Making Single-Threaded Code Multithreaded

Threads



A word processor with three threads

现代os：一个进程可以创建多个线程

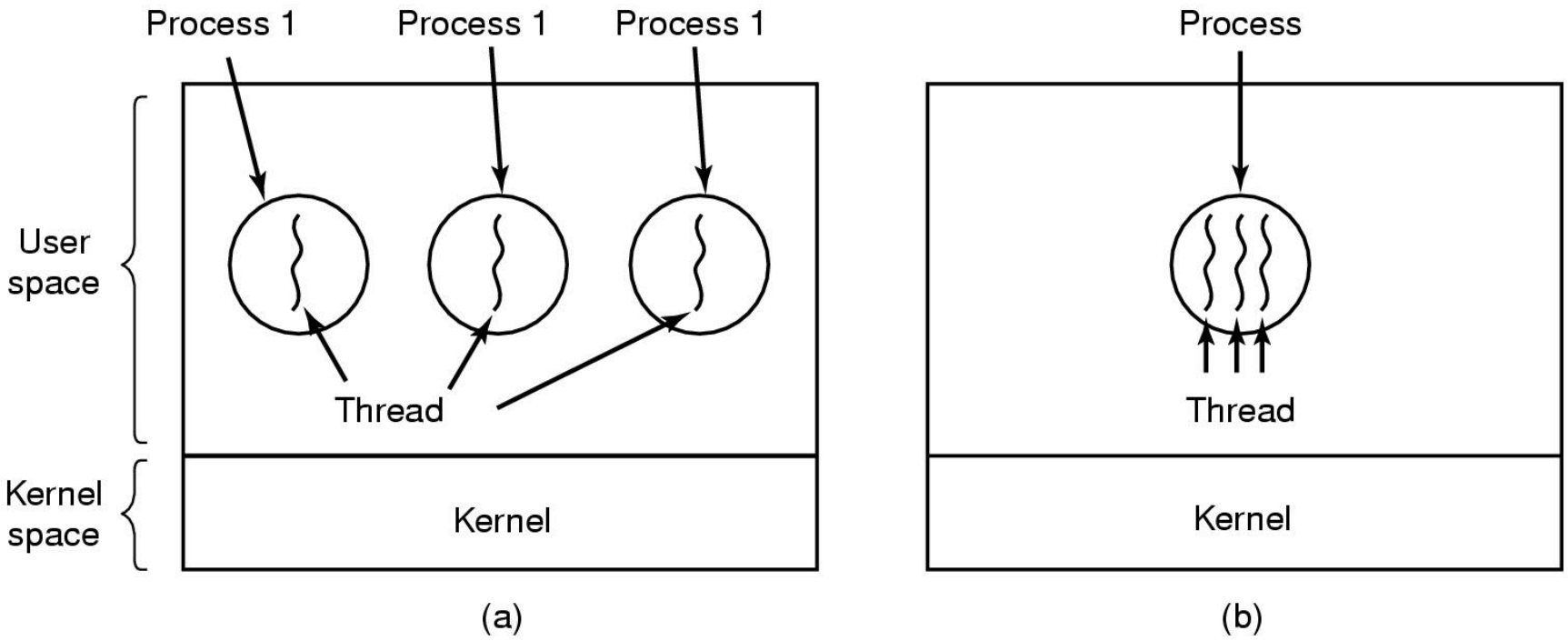
- **线程：**“轻量级进程（Lightweight Process）”，是进程的一个实体，是被独立调度和分派的基本单位，表示进程中的一个控制点（执行体），执行一系列指令。

进程作为其他资源分配单位

线程作为CPU调度单位

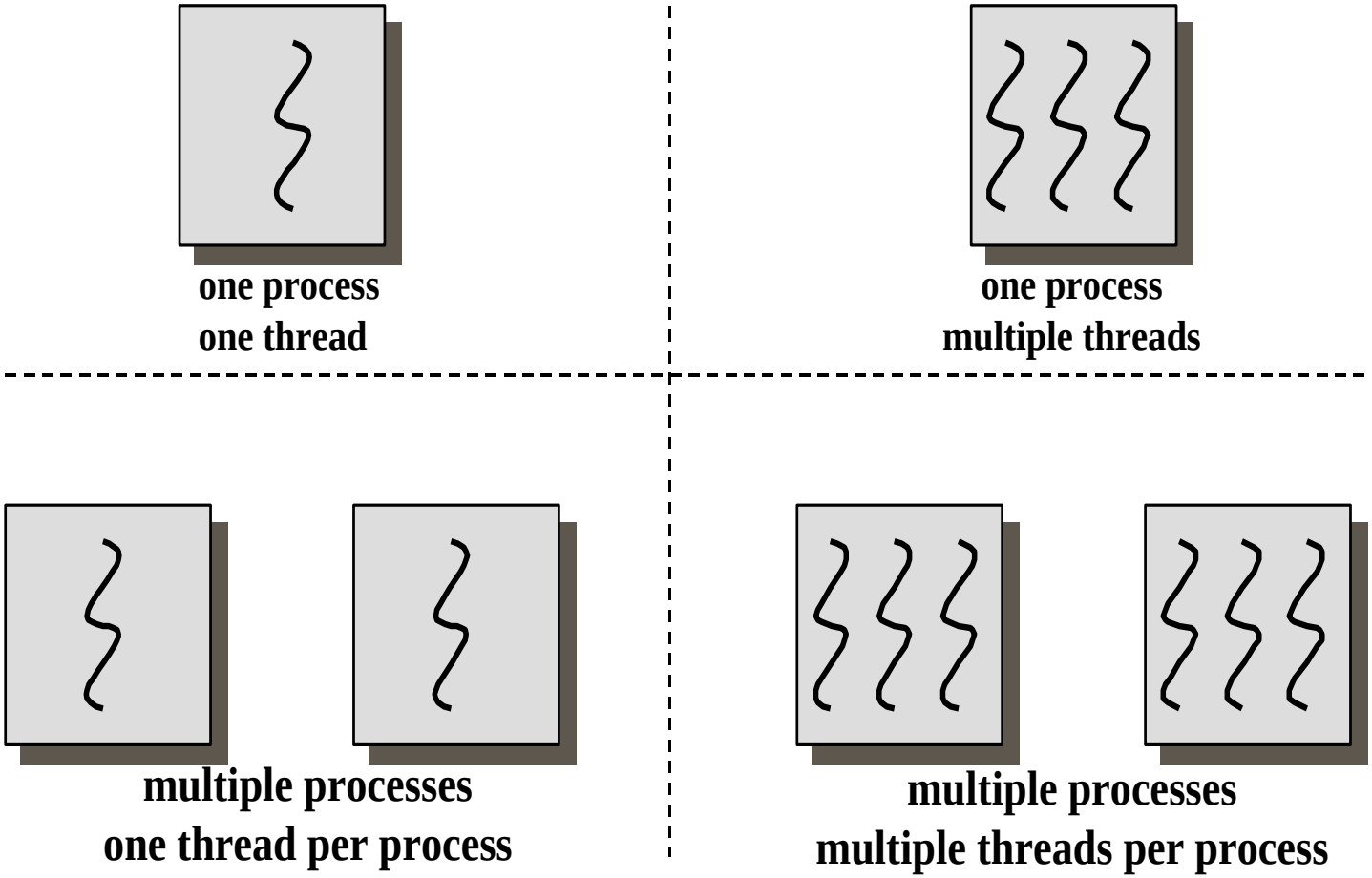
只拥有必不可少的资源，如：PC、寄存器上下文和栈

The Thread Model (1)



(a) Three processes each with one thread

(b) One process with three threads

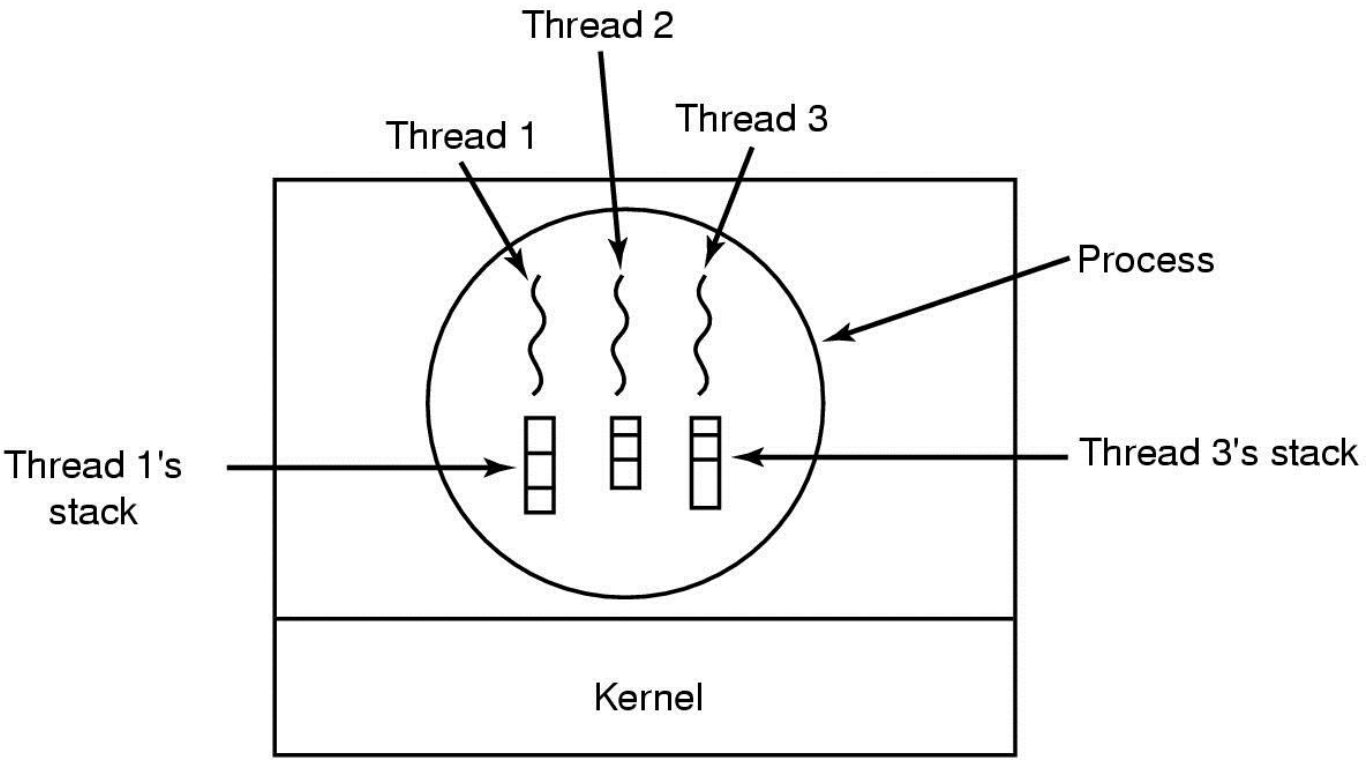

one process
one thread
one process
multiple threads
multiple processes
one thread per process
multiple processes
multiple threads per process

The Thread Model (2)

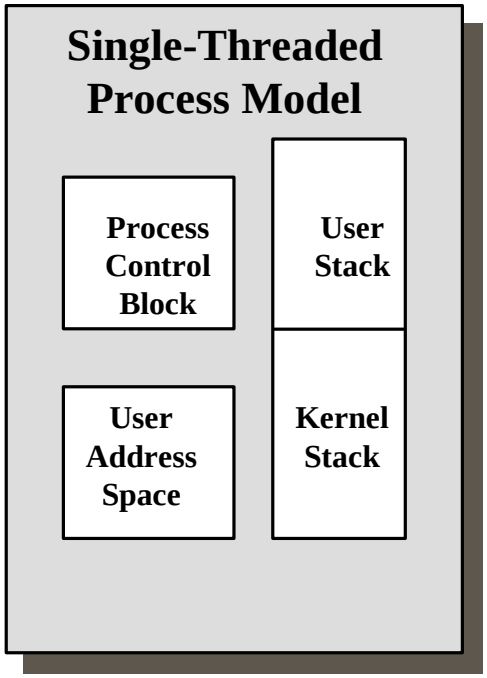
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

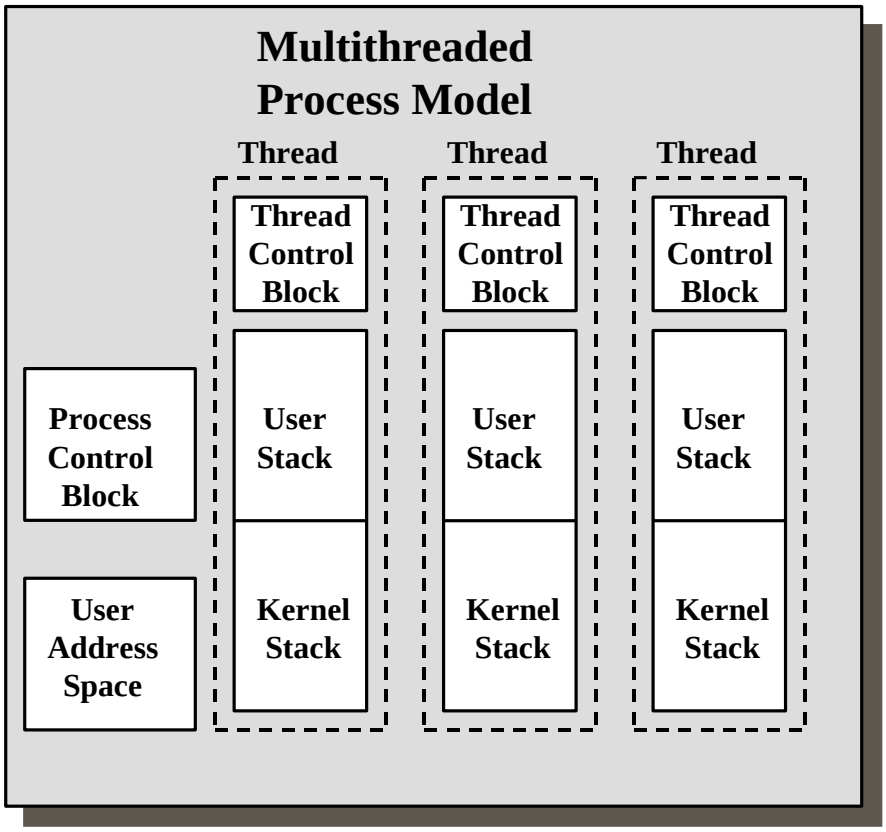
- Items shared by all threads in a process
- Items private to each thread

The Thread Model (3)



Each thread has its own stack


Single-Threaded
Process Model
Process
Control
Block
User
Stack
User
Address
Space
Kernel
Stack


Multithreaded
Process Model
Thread
Thread
Thread
Thread
Control
Block
Thread
Control
Block
Thread
Control
Block
Process
Control
Block
User
Stack
User
Stack
User
Stack
User
Address
Space
Kernel
Stack
Kernel
Stack
Kernel
Stack

POSIX Threads

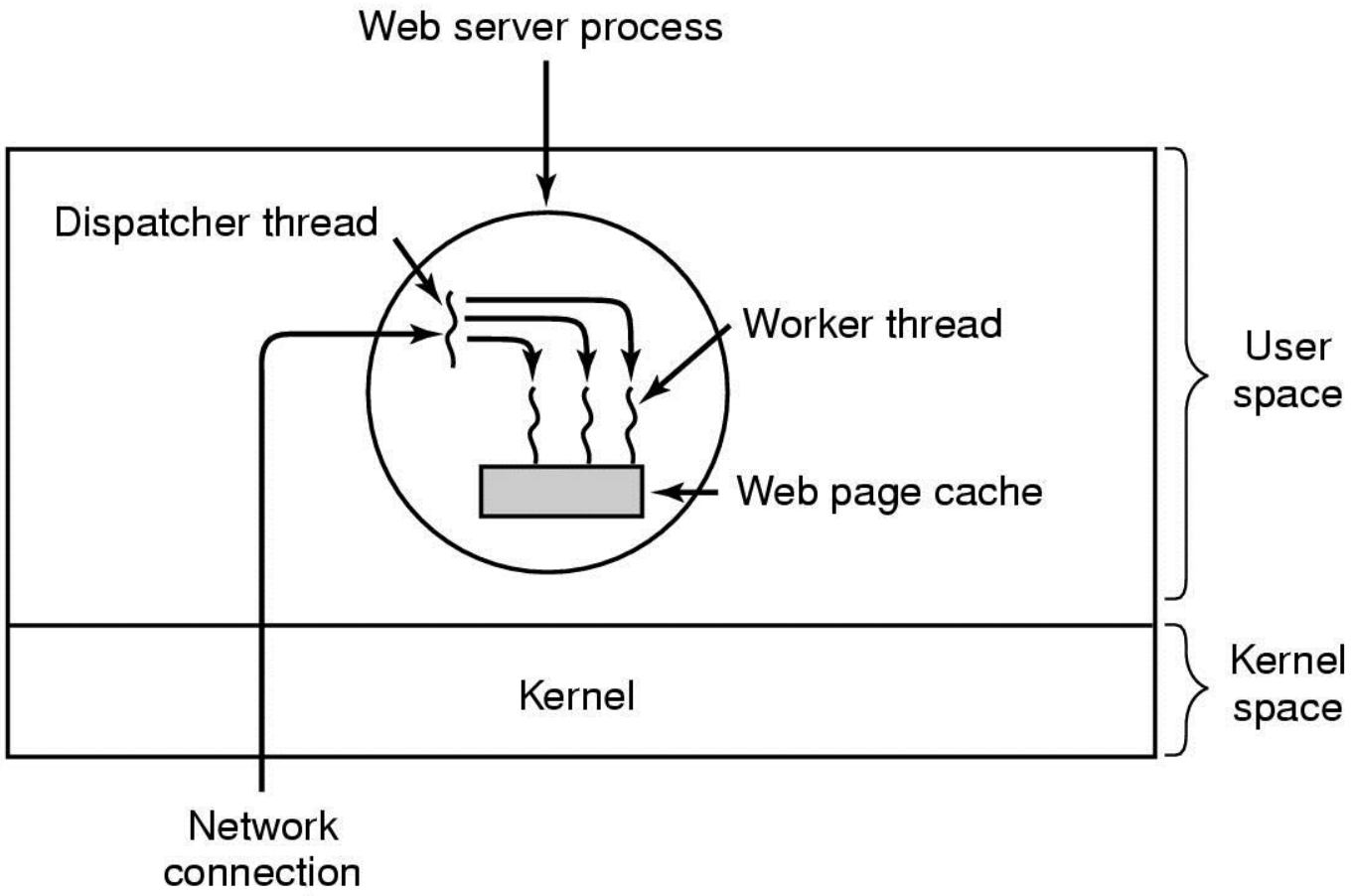
Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure 2-14. Some of the Pthreads function calls.

Thread Usage

- Parallel entities
- Easy to create and destroy
- Substantial calculation & substantial I/O
- Useful on systems with CPUs

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)

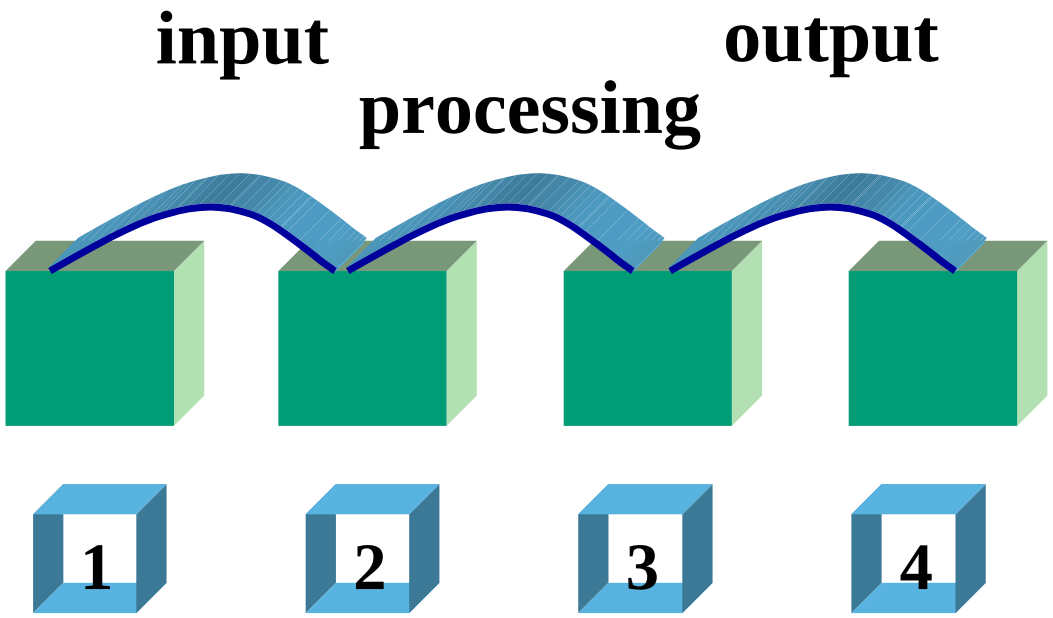
- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread

Thread Usage (4)

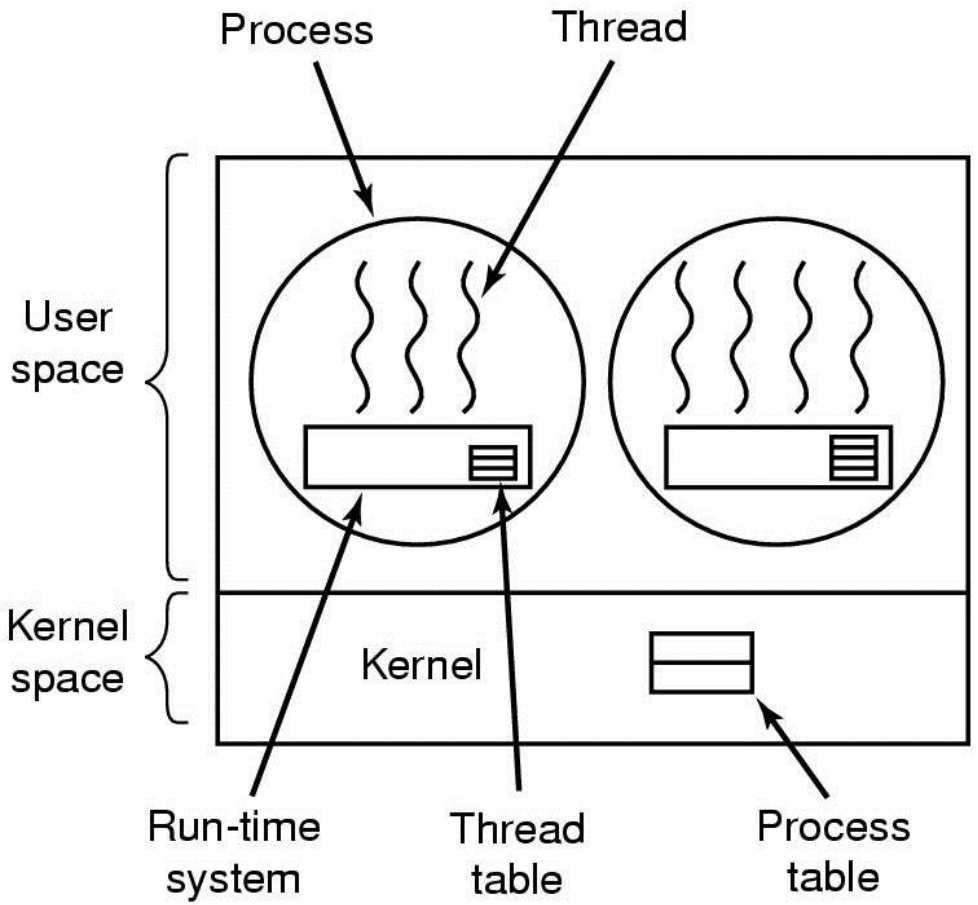
Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Three ways to construct a server

Thread Usage (5)

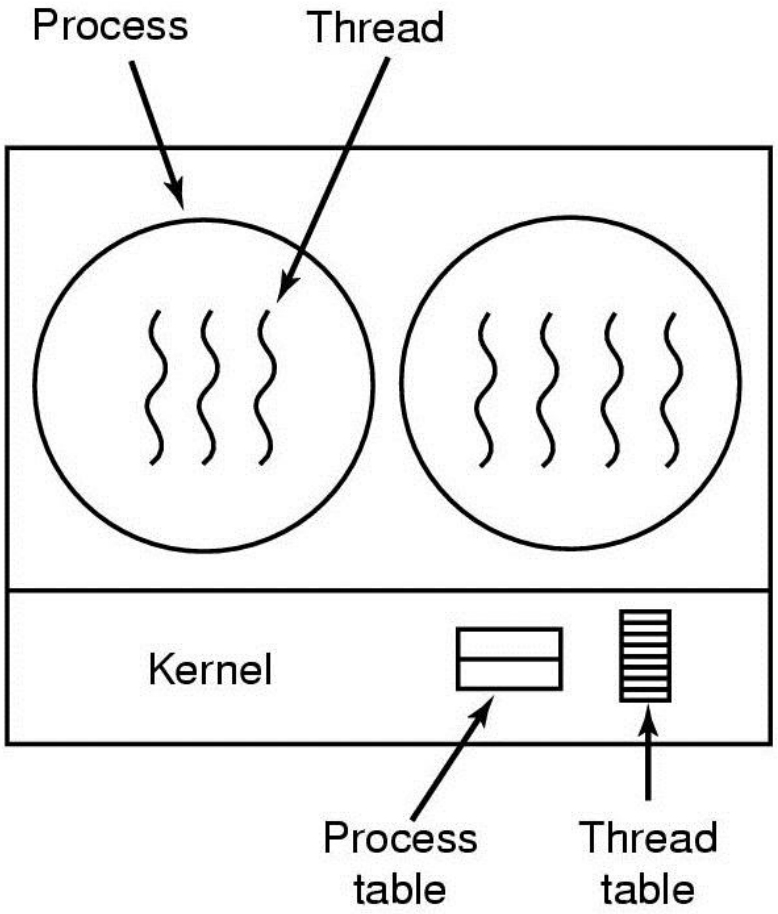


Implementing Threads in User Space



A user-level threads package

Implementing Threads in the Kernel



A threads package managed by the kernel

进程和线程的比较

- **地址空间和其他资源（如打开文件）**：进程间相互独立，同一进程的各线程间共享。
- **调度**：线程上下文由于其小而创建、撤销及切换时间比进程上下文切换要快得多。
- **并发性**：不仅进程之间可并发执行，一个进程中的多个线程之间也可并发执行，从而系统具有更好的并发行。
- **切换**：同一进程中，线程的切换不会引起进程的切换，只有从一进程中的线程切换到另一进程中的线程时，才会引起进程的切换。

INTERPROCESS COMMUNICATION

- Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- Semaphores
- Mutexes
- Monitors
- Message Passing
- Barriers

Interprocess Communication

Three Issues

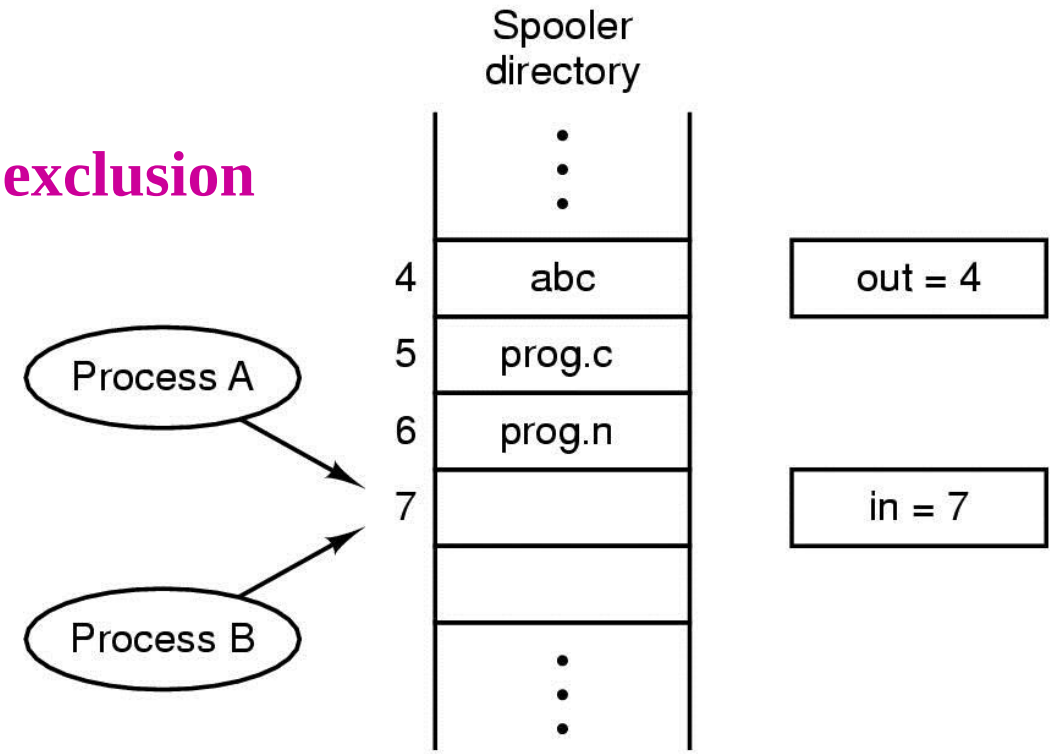
Communication

Mutual exclusion

Synchronous

Interprocess Communication
Race Conditions

Mutual exclusion

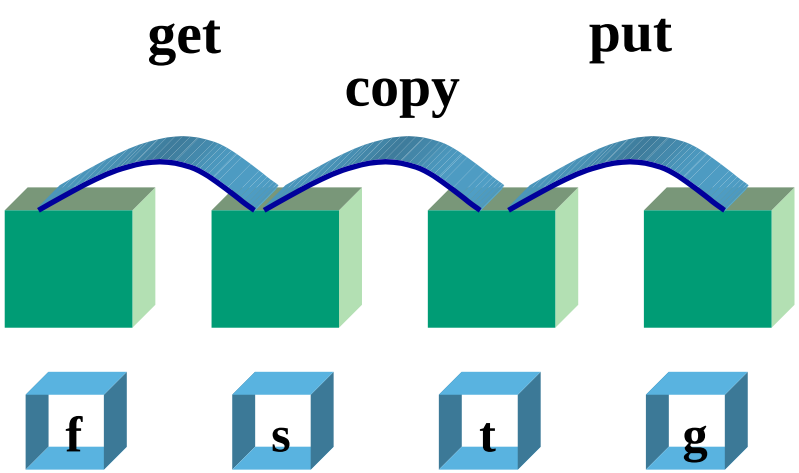


- Two processes want to access shared memory at same time

- 竞争：两个或多个进程对同一共享数据同时进行访问，而最后的结果是不可预测的，它取决于各个进程对共享数据访问的相对次序。这种情形叫做竞争。

- 竞争条件：多个进程并发访问和操作同一数据且执行结果与访问的特定顺序有关。

Synchronization



	f	s	t	g	结果
初始状态	3,4,...,m	2	2	(1,2)	
g,c,p	4,5,...,m	3	3	(1,2,3)	正确
g,p,c	4,5,...,m	3	3	(1,2,2)	错误
c,g,p	4,5,...,m	3	2	(1,2,2)	错误
c,p,g	4,5,...,m	3	2	(1,2,2)	错误
p,c,g	4,5,...,m	3	2	(1,2,2)	错误
p,g,c	4,5,...,m	3	3	(1,2,2)	错误

Coming data blocks

- 进程间的制约关系

  – 间接制约：进行竞争——独占分配到的部分或全部共享资源，“互斥”

  – 直接制约：进行协作——等待来自其他进程的信息，“同步”

Critical Regions (0)

临界资源（Critical Resources）：一种一次只能为一个进程服务的资源。

临界区（Critical Section）：进程中访问临界资源的程序。每个使用该资源的进程都要包含一个临界区。

```
    …
critical section
    …   ; remainder section
```

两个进程不能同时进入访问同一临界资源的临界区，这称为**进程互斥**。

一个飞机订票系统，两个终端，运行T1、T2进程

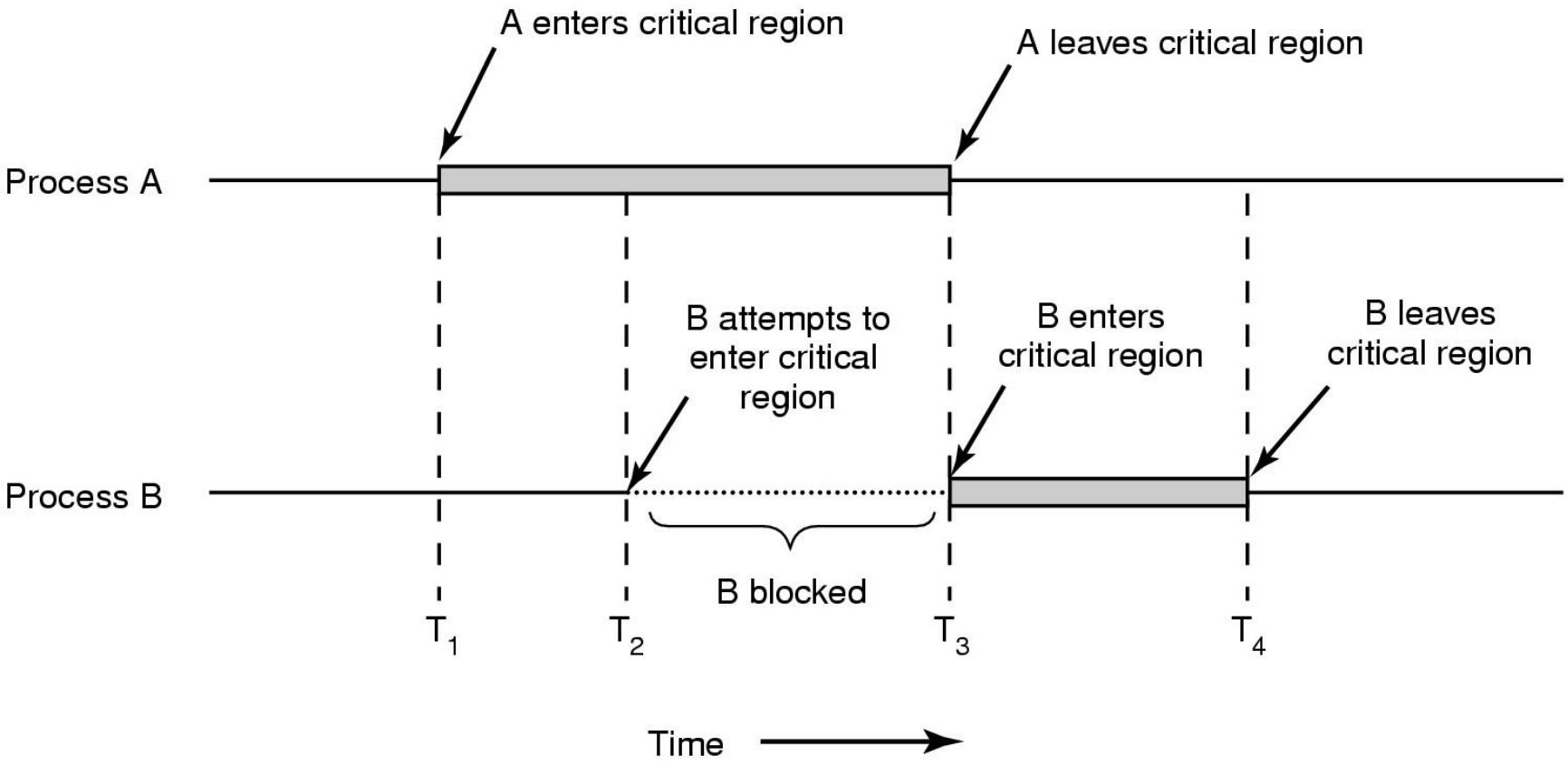
```
T1 :                    T2:
...                     ...
Read(x);                Read(x);
if x>=1 then            if x>=1 then
    x:=x-1;                 x:=x-1;
write(x);               write(x);
...                     ...
```

Critical Regions (1)

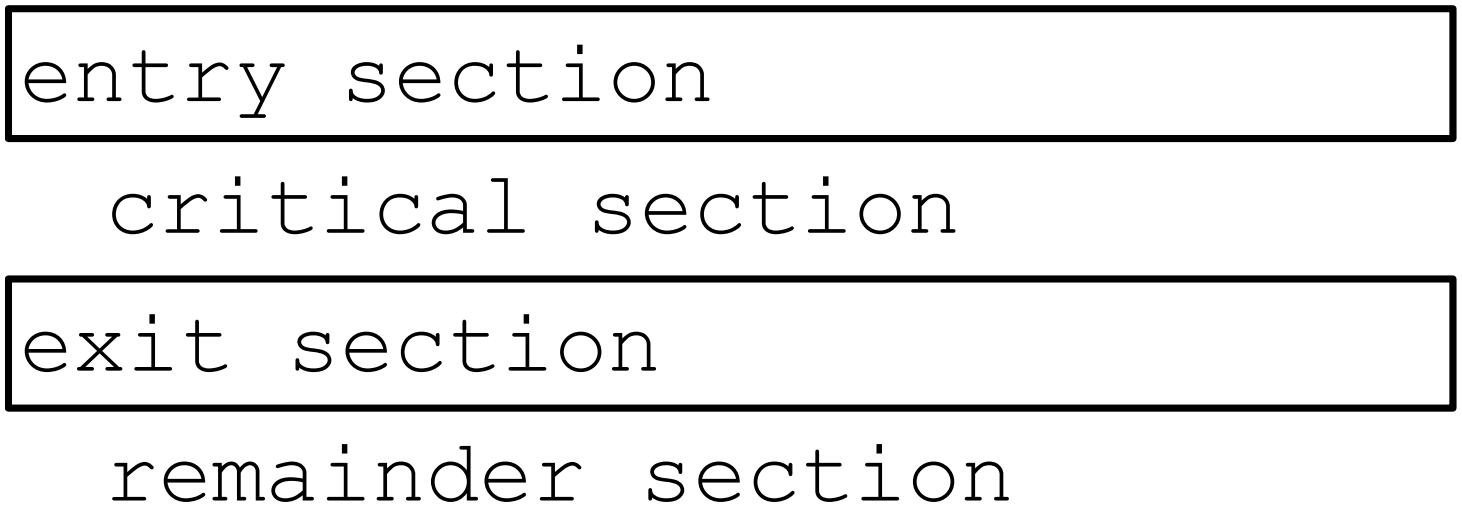
Four conditions to provide mutual exclusion

1. No two processes simultaneously in critical region（空闲则入）
2. No assumptions made about speeds or numbers of CPUs（忙则等待）
3. No process running outside its critical region may block another process（让权等待）
4. No process must wait forever to enter its critical region（有限等待）

Critical Regions (2)



- Mutual exclusion using critical regions


entry section
critical section
exit section
remainder section

Mutual Exclusion with Busy Waiting

Proposals for achieving mutual exclusion:

- Disabling interrupts
- Lock variables
- Strict alternation
- Peterson's solution
- The TSL instruction

Mutual Exclusion with Busy Waiting (1)

```
while (lock!=0);          <a>
lock= 1;                  <b>
   critical section
lock= 0;
   remainder section
```

Lock Variables

Mutual Exclusion with Busy Waiting (2)

```
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)

Strict Alternation

(a) Process 0. (b) Process 1.

Mutual Exclusion with Busy Waiting (3)

```
#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

Mutual Exclusion with Busy Waiting (4)

```
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

```
enter_region:
        MOVE REGISTER,#1            | put a 1 in the register
        XCHG REGISTER,LOCK          | swap the contents of the register and lock variable
        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 XCHG instruction.

- 优点
  - 适用于任意数目的进程，在单处理器或多处理器上更显其优越
  - 简单，容易验证其正确性
  - 可以支持进程内存在多个资源对应的临界区，只需为每个临界资源对应的临界区设立一个布尔变量
- 缺点
  - 等待要耗费CPU时间，即“忙等”，不能实现“让权等待”
  - 可能产生“饿死”现象：有的进程可能永远执行不了

试考虑以下进程P_A和P_B反复使用临界区的情况：

P_A

A：lock(key［S］)

〈S〉

unlock(key［S］)

Goto A

P_B

B：lock(key ［S］)

＜S＞

unlock(key ［S］)

Goto B

Semaphores

前面的互斥算法缺乏公平性，只有获得处理机的进程才可进行进入临界区的测试，且测试结果具有偶然性。

需要一个地位高于进程的管理者来解决公有资源的使用问题。OS可从进程管理者的角度来处理互斥的问题，信号量就是OS提供的管理公有资源的有效手段。

信号量代表可用资源实体的数量。

- 信号量具有以下特性：

1) 信号量是一个整形变量。

2) 每一个信号量表示一种系统资源的状况，其值表示该资源当前可用的数量，初值为非零。

3) 每一个信号量都对应一个空或非空的等待队列。该队列就是信号量所代表的资源的等待队列。

4) 对信号量只能实施down、up（ P、V ）操作，只有P、V操作原语才能改变其值。

- 信号量只能通过**初始化**和**两个标准的原语**来访问——作为OS核心代码执行，不受进程调度的打断

- **初始化**资源信号量为指定一个非负整数值，表示**空闲资源总数**——若为非负值表示**当前的空闲资源数**，若为负值其绝对值表示**当前等待临界区的进程数**

原语 *down(s); P(s)*

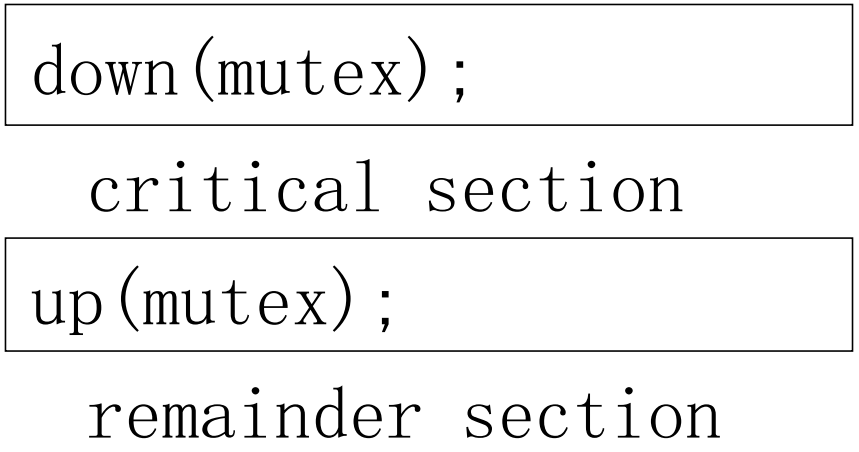
```
procedure down(var s:semaphore)
    begin
       s:=s-1;
       if s<0 then W(s);      //将调用P操作原语的进程置
                              成等待信号量s的状态。
    end
```

原语 *up(s); V(s)*

```
Procedure up(var s:semaphore)
  begin
    s:=s+1;
    if s≤0 then R(s);          //从信号量s的等待队列中释放一个进程。
  end
```

Mutexes

- ✓ 在实现进程互斥时，信号量的初值设为1，表示中只允许一个进程进入临界区。
- ✓ 在进程执行过程中，当进入临界区时执行down操作，在离开临界区时执行up操作，使临界区位于对同一个信号量的down操作和up操作之间。



- **mutex**为**互斥信号量**，其**初值为1**；在每个进程中将临界区代码置于down(mutex)和up(mutex)原语之间
- 必须**成对使用**down和up原语：遗漏down原语则不能保证互斥访问，遗漏up原语则不能在使用临界资源之后将其释放（给其他等待的进程）；down、up原语**不能次序错误、重复或遗漏**

- 在进程互斥中使用的信号量，每个进程都可以对它实施down操作，这样的信号量又称为公用信号量。

用信号量实现两并发进程的互斥

s:semaphore;

s:=1;

进程A	**进程B**
. . . . . .	**. . . . . .**
down(s);	**down(s);**
临界区CRA;	**临界区CRB;**
up(s);	**up(s);**
. . . . . .	**. . . . . .**

s=1　　0　　-1　　0　　1

进程 A　s:=s-1　down(s)　CRA　s:=s+1 [唤醒] up(s)

进程 B　s:=s-1 [阻塞] down(s)　CRB　s:=s+1 up(s)

--------t1----------t2----------t3----------t4---->

Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
        CMP REGISTER,#0             | was mutex zero?
        JZE ok                      | if it was zero, mutex was unlocked, so return
        CALL thread_yield           | mutex is busy; schedule another thread
        JMP mutex_lock              | try again later
ok:     RET | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0               | store a 0 in mutex
        RET | return to caller
```

- Implementation of mutex_lock and mutex_unlock

Synchronous

P_C （计算进程）

:

A: 计算

得到计算结果

Buf ← 计算结果

Goto A

P_P （打印进程）

:

B: 打印 Buf中的数据

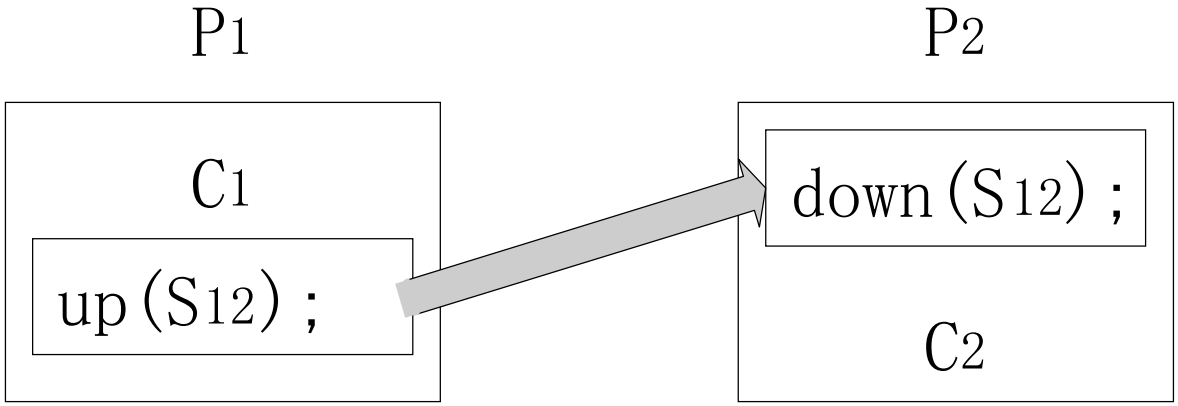
清除 Buf中的数据

Goto B

一组并发进程，各自的执行结果互为对方的执行条件，则需要互相发送消息以便互相合作、互相等待，按一定的速度执行的过程称为进程间的同步。

一般来说，可以把各进程之间发送的消息作为信号量看待。

利用信号量来描述前趋关系



- 前趋关系：并发执行的进程P1和P2中，分别有代码C1和C2，要求C1在C2开始前完成；
- 为该前趋关系设置一个信号量 S12，初值为0

与进程互斥时不同的是，这里的信号量只与制约进程及被制约进程有关而不是与整组并发进程有关。因此，称该信号量为私用信号量（**Private Semaphore**）。

如运算和打印进程分别为进程T1和T2，要求T1和T2同步运行，则

设定两个信号量s1和s2，s1的初值为1，s2的初值为0。

```
s1,s2:semaphore;
s1=1,s2=0;
cobegin
    repeat T1;
    repeat T2;
coend;
```

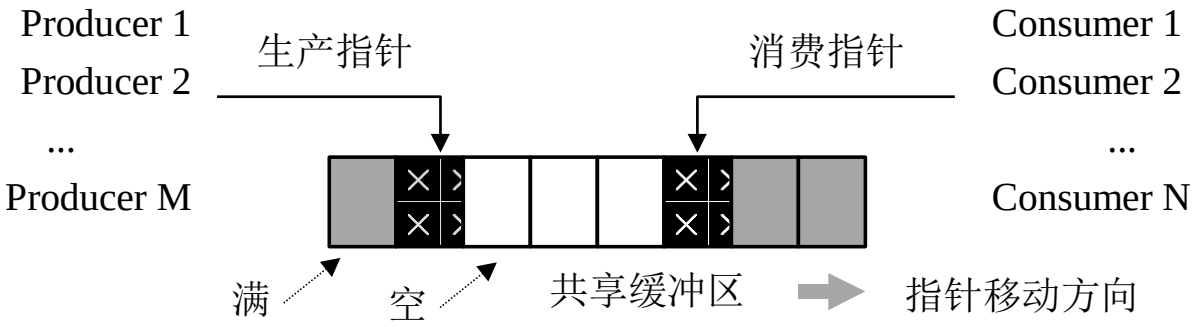
```
procedure T1:
  begin
      . . .
      down(s1);
      m1;
      up(s2);
      . . .
  end;
```

```
procedure T2:
  begin
      . . .
      down(s2);
      m2;
      up(s1);
      . . .
  end;
```

- 只能由一个进程对其实施down操作的信号量称为该进程的私有信号量。
- s1是T1的私有信号量，s2是T2的私有信号量。
- 在两个进程相互推进的运行过程中，哪个进程的私有信号量为1，就表示它可以向前推进。

生产者—消费者问题
(the producer-consumer problem)

问题描述：若干进程通过有限的共享缓冲区交换数据。其中，“生产者”进程不断写入，而“消费者”进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。



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

引入：信号量——用来累计唤醒次数
（对方的执行许可）

当产生一个唤醒信号执行**up**操作；
用到一个唤醒信号执行**down**操作

Semaphores

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item( );                 /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item( );                  /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

The producer-consumer problem using semaphores

- 采用信号量机制：
  - full是“满”数目，初值为0，empty是“空”数目，初值为N。实际上，full和empty是同一个作用，始终有full + empty = N
  - mutex用于访问缓冲区时的互斥，初值是1

- 进程中down操作的次序是重要的：先检查资源数目，再检查是否互斥——否则可能死锁(为什么？)

PROBLEMS

- The bridge cross a river. A person wants to walk through the bridge from north to south, another person wants to walk through the bridge from south to north. The bridge can pass through one person at a time. Please gave a algorithm to solve such problem.

- At a bus, the driver and the conductor have situation as follows:
- Driver activity: start bus
- drive bus
- stop bus
- Conductor activity: close door
- sell tickets
- open door
- When the bus drives on, it should not open the door, and if the door do not closed, it should not drive bus.
- Please, describe a procedure which can keep the driver and conductor synchronization.

Monitors

信号量同步的缺点：

- **同步操作分散：**信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）；
- **易读性差：**要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；
- **不利于修改和维护：**各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；
- **正确性难以保证：**操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误；

Monitors

管程是一种具有面向对象程序设计思想的同步机制。它提供了与信号量机制相同的功能。

管程（Monitor）概念是著名学者Hore于1974年提出的，并在很多系统中得到实现，诸如Pascal、Java、Modula-3等。

管程是由局部数据结构、多个处理过程和一套初始化代码组成的模块。

```
monitor example
        integer i;
        condition c;

        procedure producer( );

        .
        .
        .
        end;

        procedure consumer( );

        .
        .
        .
        end;
end monitor;
```

Example of a monitor

Monitors

管程的特征：

（1）管程内的数据结构只能被管程内的过程访问，任何外部访问都是不允许的；

（2）进程可通过调用管程的一个过程进入管程；

（3）任何时间只允许一个进程进入管程，其他要求进入管程的进程统统被阻塞到等待管程的队列上。

实现方法：

- **将所有的临界区转换成管程中的过程——互斥；**
- **引入条件变量full、empty及相关的操作wait、signal——同步。**

Monitors

```
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

- 管程实现了临界区互斥的自动化，使并行编程更容易保证正确性。
- **管程是一个编程语言中用到的概念，必须用支持管程的语言才能实现**。

Message Passing

之前介绍的方法多用于有公共内存的进程间通信，当涉及局域网相连的机器之间的通信——消息传递。

消息传递中的管理机制由操作系统提供，主要体现为以下两个原语。

发送原语：send(destination, message)，其中，destination为消息的目的地（接收进程名）。该原语表示发送消息到进程destination。

接收原语：receive(source, message)，其中，source是消息发出地（发送进程名）。该原语表示从进程source接收消息。

Producer-Consumer Problem with Message Passing (1)

```
#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 */
     }
}
```

. . .

The producer-consumer problem with N messages.

Producer-Consumer Problem with Message Passing (2)

. . .

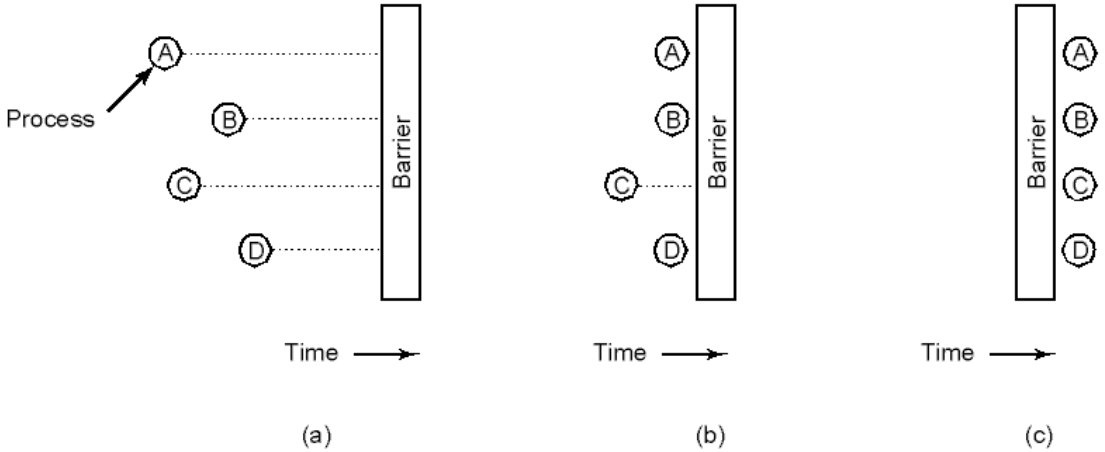
```
void consumer(void)
{
        int item, i;
        message m;

        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

针对一组进程的同步机制。



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

补充：进程间通信—进程之间交换信息

- **低级通信：**只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量。
  - 主要用于控制进程执行速度的作用。
  - 优点：速度快。
  - 缺点：传送信息量小。效率低，每次通信传递的信息量固定，若传递较多信息则需要进行多次通信。

- **高级通信：**能够传送任意数量的数据，目的主要是用于交换信息。

高级通信常用方式

- 共享存储区方式
  - 不要求数据移动，可以任意读写和使用任意数据结构，需要进程互斥和同步的辅助来确保数据一致性
- 消息或邮箱机制
- 管道、文件、文件映射

共享存储区

这种通信需要在内存中开辟一个共享的存储空间，供进程之间进行数据传递。比如计算进程将所得的结果送入内存共享区的缓冲区环中，打印进程从中将结果取出来，就是一个利用共享存储器进行通信的例子。这种通信方式在UNIX、Linux、Windows及OS/2等系统中都有具体的实现。

共享存储器系统中，共享的空间一般应当是需要互斥访问的临界资源。诸多进程为了避免丢失数据或重复取数，需要执行特定的同步协议。

在利用共享存储器进行通信之前，信息的发送者和接收者都要将共享空间纳入到自己的虚地址空间中，让它们都能访问该区域。存储器管理模块将共享空间映射成实际的内存空间。

（1）创建或删除共享存储区
（2）共享存储区的附接与断接
（3）共享存储区状态查询
（4）共享存储区管理

共享存储器系统的特点：

- 利用共享存储器系统进行通信的效率特别高，适用于通信速度要求特别高的场合。
- 这种同步与互斥机制的实现一般要由程序员来承担，系统仅仅提供一个共享内存空间的管理机制。

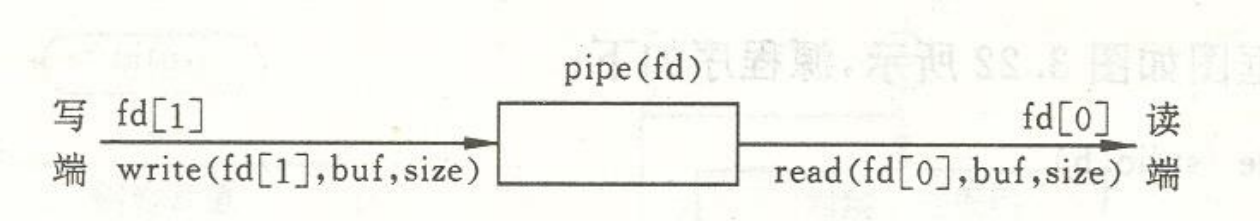
Linux的共享存储区

- 创建或打开共享存储区(shmget)：依据用户给出的整数值key，创建新区或打开现有区，返回一个共享存储区ID。
- 连接共享存储区(shmat)：连接共享存储区到本进程的地址空间，可以指定虚拟地址或由系统分配，返回共享存储区首地址。父进程已连接的共享存储区可被fork创建的子进程继承。
- 拆除共享存储区连接(shmdt)：拆除共享存储区与本进程地址空间的连接。
- 共享存储区控制(shmctl)：对共享存储区进行控制。如：共享存储区的删除需要显式调用shmctl(shmid, IPC_RMID, 0)。

管道(pipe)

管道是一条以字节流方式传送的信息通道，
适用于大容量数据的通信。

管道是可供进程共享的一种特殊文件，是单向的，主要用于连接一个写入进程和一个读出进程，以实现它们之间的数据通信。在使用管道前要建立相应的管道，然后才可使用。



利用UNIX提供的系统调用pipe，可建立一条同步通信管道。
其格式为：　pipe(fd)
intfd［2］；
这里，fd［1］为写入端，fd［0］为读出端。

示例：

用C语言编写一个程序，建立一个pipe，同时父进程生成一个子进程，子进程向pipe中写入一字符串，父进程从pipe中读出该字符串。

解： 程序如下：

```
#include 〈stdio.h〉
main()
 {
    intx，fd［2］；
    char buf［30］，s［30］；
    pipe(fd)；/*创建管道*/
    while((x=fork())==-1)；/*创建子进程失败时，循环*/
    if(x==0)
```

```
        {
          sprintf(buf，″This is an example \ n″);
          write(fd［1］，buf，30)；/*把buf中字符写入管道
*/
          exit(0)；
        }
        else/*父进程返回*/
        {
          wait(0)；
          read(fd［0］，s，30)；/*父进程读管道中字符*/
          printf(″%s″，s)；
        }
 }
```

Linux管道

- 通过pipe系统调用创建无名管道，得到两个文件描述符，分别用于写和读。
  - int pipe(int fildes[2]);
  - 文件描述符fildes[0]为读端，fildes[1]为写端；
  - 通过系统调用write和read进行管道的写和读；
  - 进程间双向通信，通常需要两个管道；
  - 只适用于父子进程之间或父进程安排的各个子进程之间；
- Linux中的命名管道，可通过mknod系统调用建立：指定mode为S_IFIFO
  - int mknod(const char *path, mode_t mode, dev_t dev);

*Windows NT*管道

无名管道：类似于Linux管道，CreatePipe可创建无名管道，得到两个读写句柄；利用ReadFile和WriteFile可进行无名管道的读写；

```
BOOL CreatePipe( PHANDLE hReadPipe,
                  // address of variable for read handle
                 PHANDLE hWritePipe,
                  // address of variable for write handle
                 LPSECURITY_ATTRIBUTES lpPipeAttributes,
                  // pointer to security attributes
                 DWORD nSize      // number of bytes reserved for pipe
);
```

小结

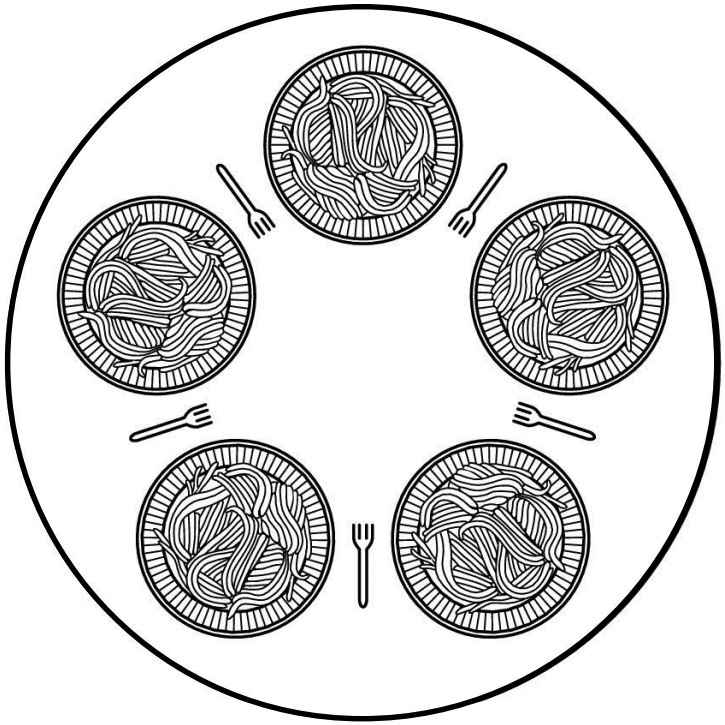
- 进程间通信的关系
  - 竞争：互斥（临界资源；临界区）
  - 协作：同步
- 信号量的含义、操作
- 信号量实现互斥同步
- 管程等其他方式

CLASSICAL IPC PROBLEMS

- The Dining Philosophers Problem
- The Readers and Writers Problem

Dining Philosophers (1)

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



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;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

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

Solution to dining philosophers problem (part 1)

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)

The Readers and Writers Problem

- 问题描述：对共享资源的读写操作

  任一时刻“写者”最多只允许一个，而“读者”则允许多个

  – “读－写”互斥，
  – “写－写”互斥，
  – “读－读”允许

The Readers and Writers Problem

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to 'rc' */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to 'rc' */
        use_data_read( );               /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

A solution to the readers and writers problem

- 采用信号量机制：
  - **db**互斥信号量:实现读者与写者、写者与写者之间的互斥，初值是1。
  - **rc**读者共享的变量:表示“正在读”的进程数，初值是0；
  - **mutex**表示对**rc**的互斥操作，初值是1。

读者优先还是写者优先 

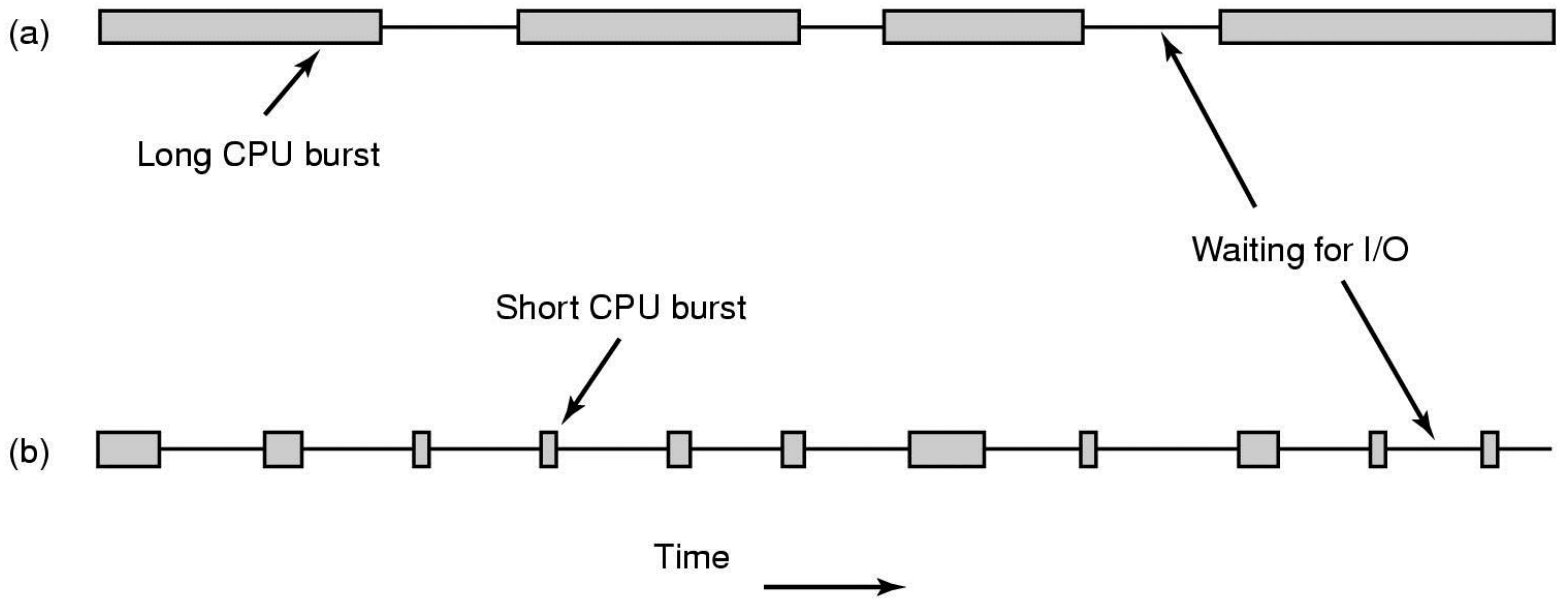
PROBLEMS

- 假定一个阅览室最多可同时容纳100个人阅读，读者进入和离开阅览室时，都必须在阅览室门口的一个登记表上登记。假定每次只允许一个人登记和去掉登记，设阅览室内有100个座位。请用down,up原语编写读者进程的同步算法。

SCHEDULING

- Introduction to Scheduling
- Scheduling in Batch Systems
- Scheduling in Interactive Systems
- Scheduling in Real-Time Systems
- Policy versus Mechanism
- Thread Scheduling

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

Introduction to Scheduling (2)

- When to schedule
  - a new process is created
  - a process exits
  - a process blocks
  - an I/O interrupt occurs
  - preemptive/nonpreemptive
- Categories of Scheduling Algorithms
  - Batch
  - Interactive
  - Real-Time

Introduction to Scheduling (3)

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

平均周转时间

$$T = \frac{1}{n}\sum_{i=1}^{n}(t_{fi} - t_{bi})$$

平均带权周转时间

$$W = \frac{1}{n}\sum_{i=1}^{n}(t_{fi} - t_{bi}) / t_{si}$$

其中：

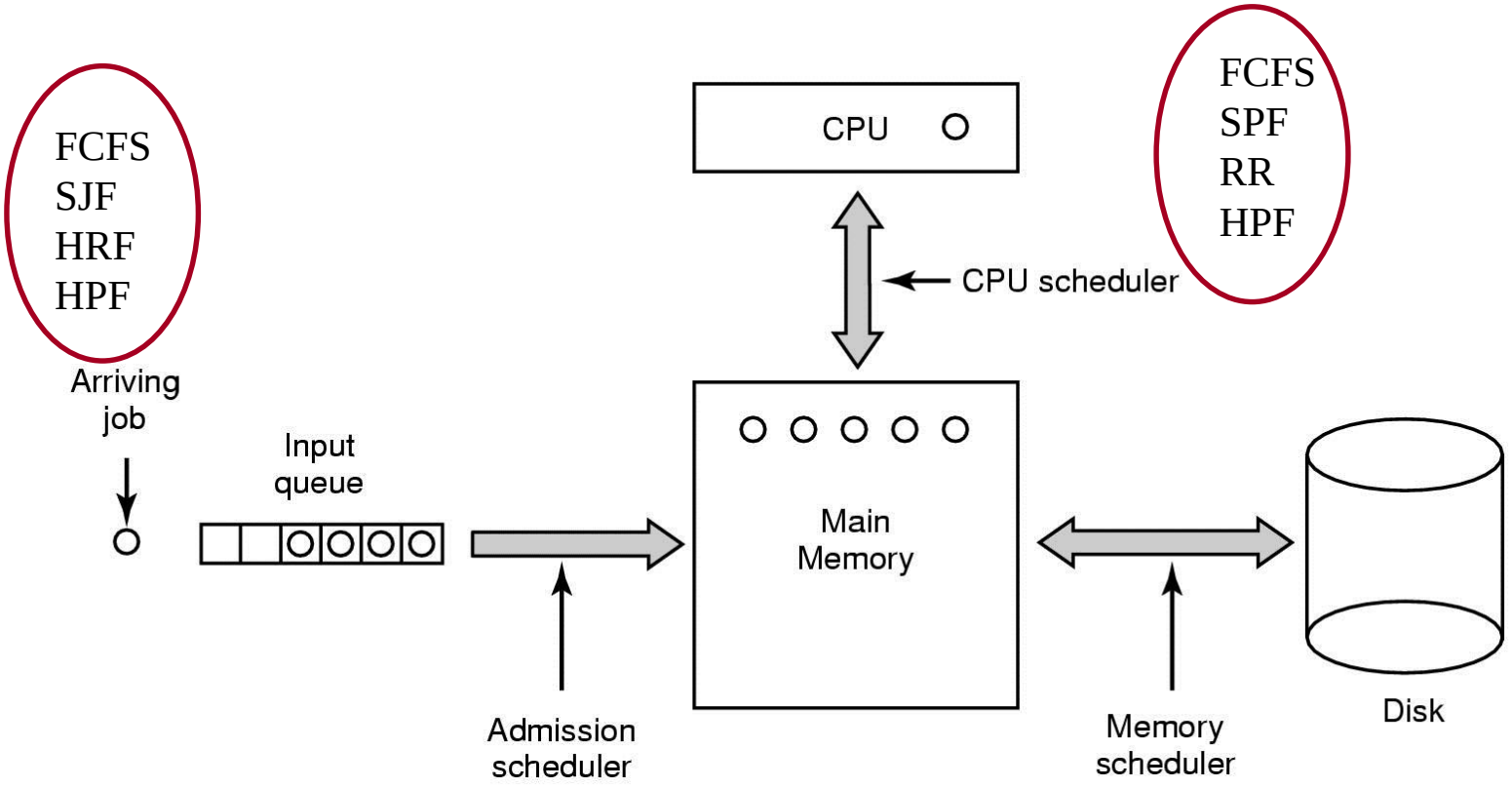
*n*为单位时间内的作业数量。

*tfi*为作业*i*的完成时间。

*tbi*为作业的开始时间。

*tsi*为作业*i*的运行时间。

Scheduling in Batch Systems (1)



Three level scheduling

Scheduling in Batch Systems (2)

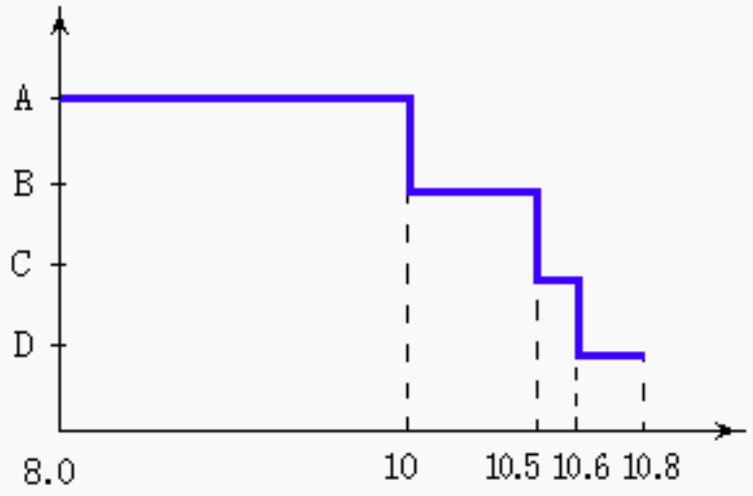
- FCFS
- SJF
- HRF 响应比=（等待时间+估计运行时间）/ 估计运行时间
- HPF
- SRF

An example of shortest job first scheduling

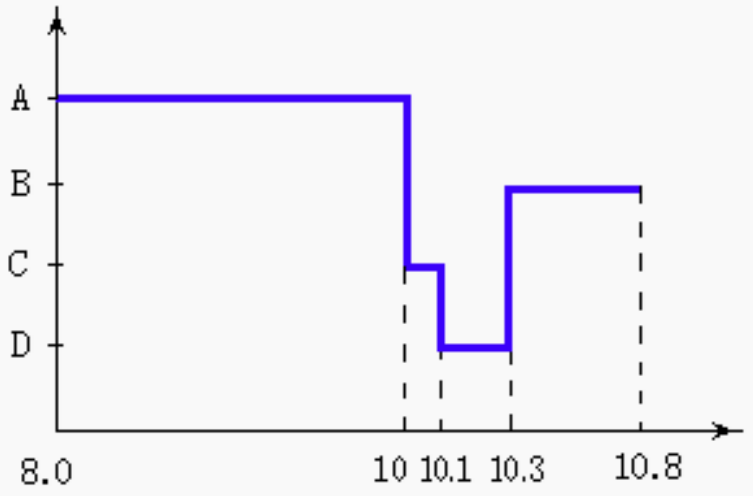
例子：

通过作业平均周转时间T和平均带权周转时间W来分析一下作业调度中的4种算法的性能。为了便于分析，我们设想的是一个单道批处理系统。选用的例子中有4个作业，按照下图所示。

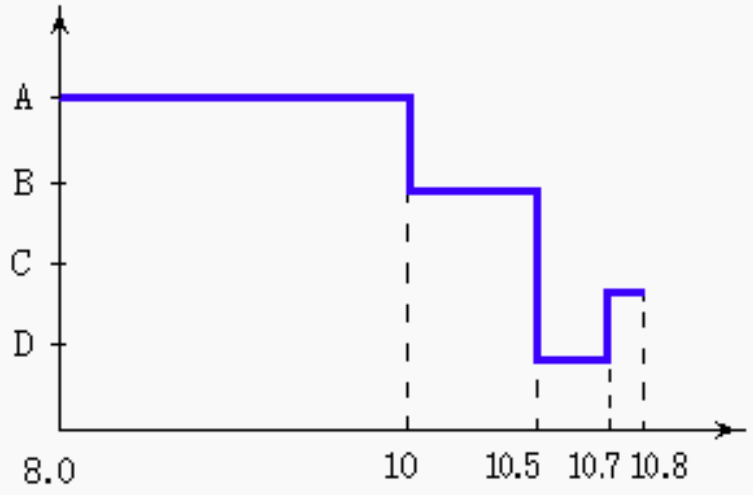
其中, 作业的到达顺序为A,B,C,D。时间为：8, 8.5, 9, 9.5。运行时间：2, 0.5, 0.1, 0.2。优先级为：10，25，7，18。



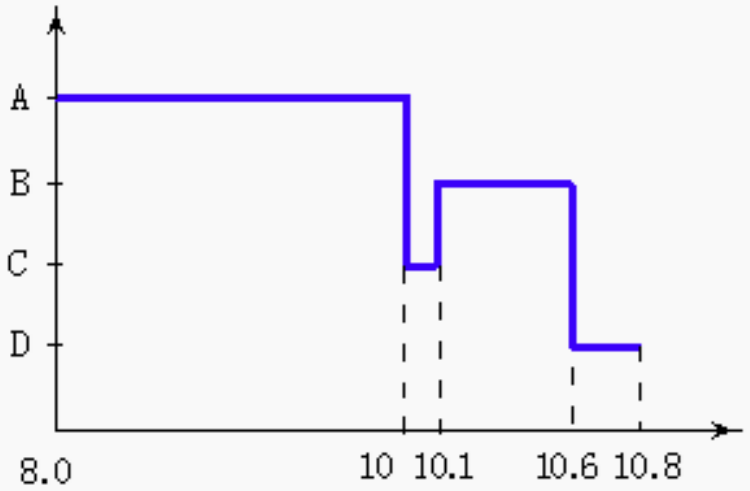
(a) FCFS 算法



(b) SJF 算法



(c) HPF 算法

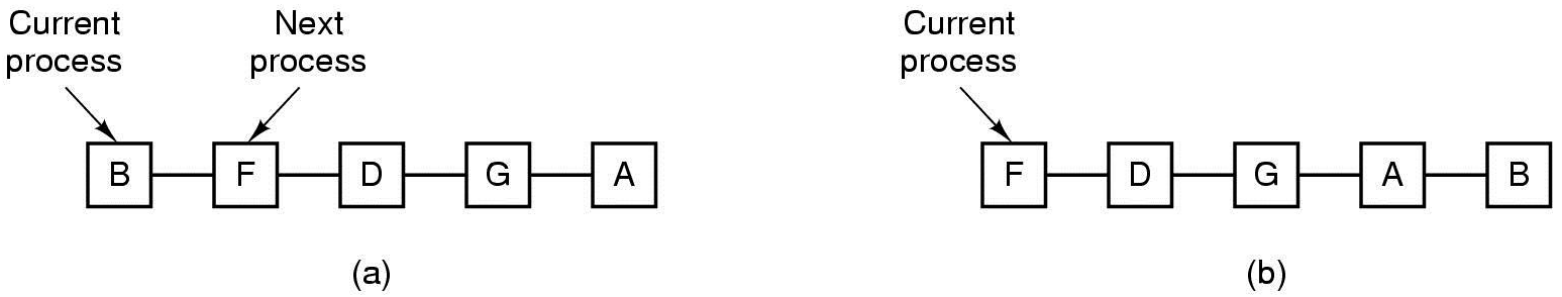


(d) HRF 算法

Scheduling in Interactive Systems

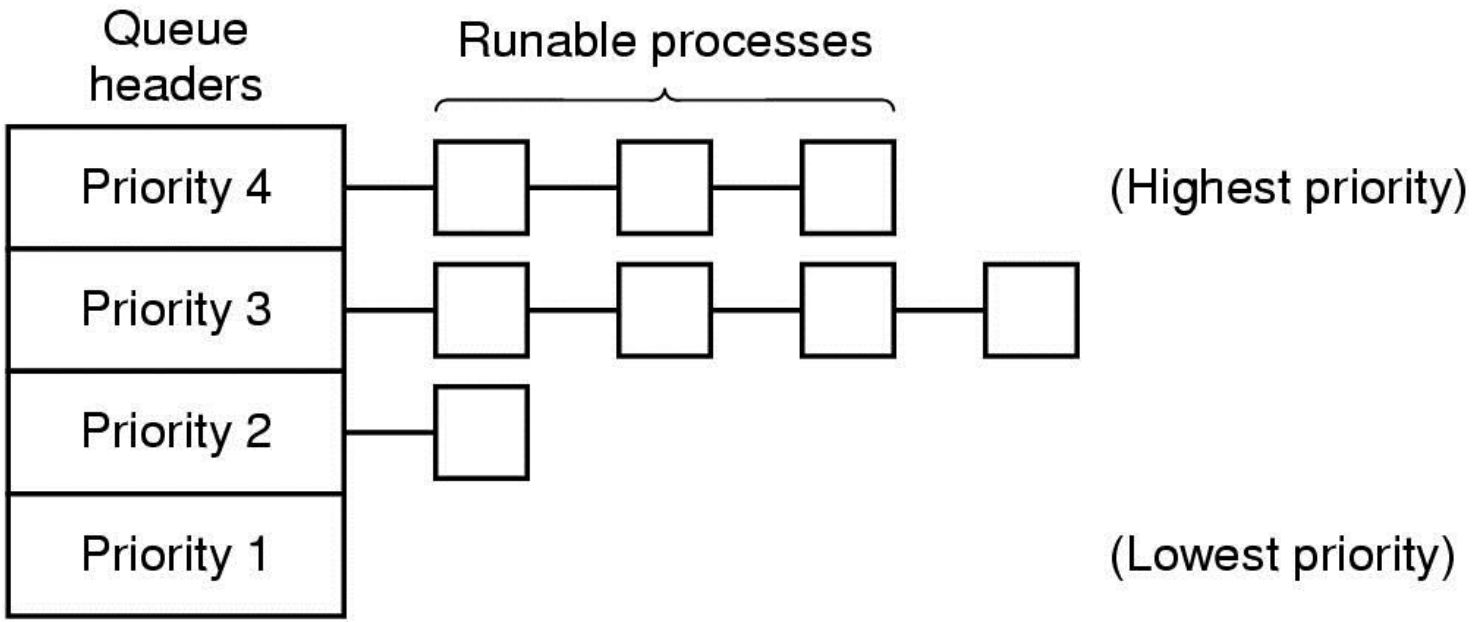
- Round-robin scheduling
- Priority scheduling
- Multiple queues
- Shortest process next
- Guaranteed scheduling
- Lottery scheduling
- Fair-share scheduling

Scheduling in Interactive Systems (1)



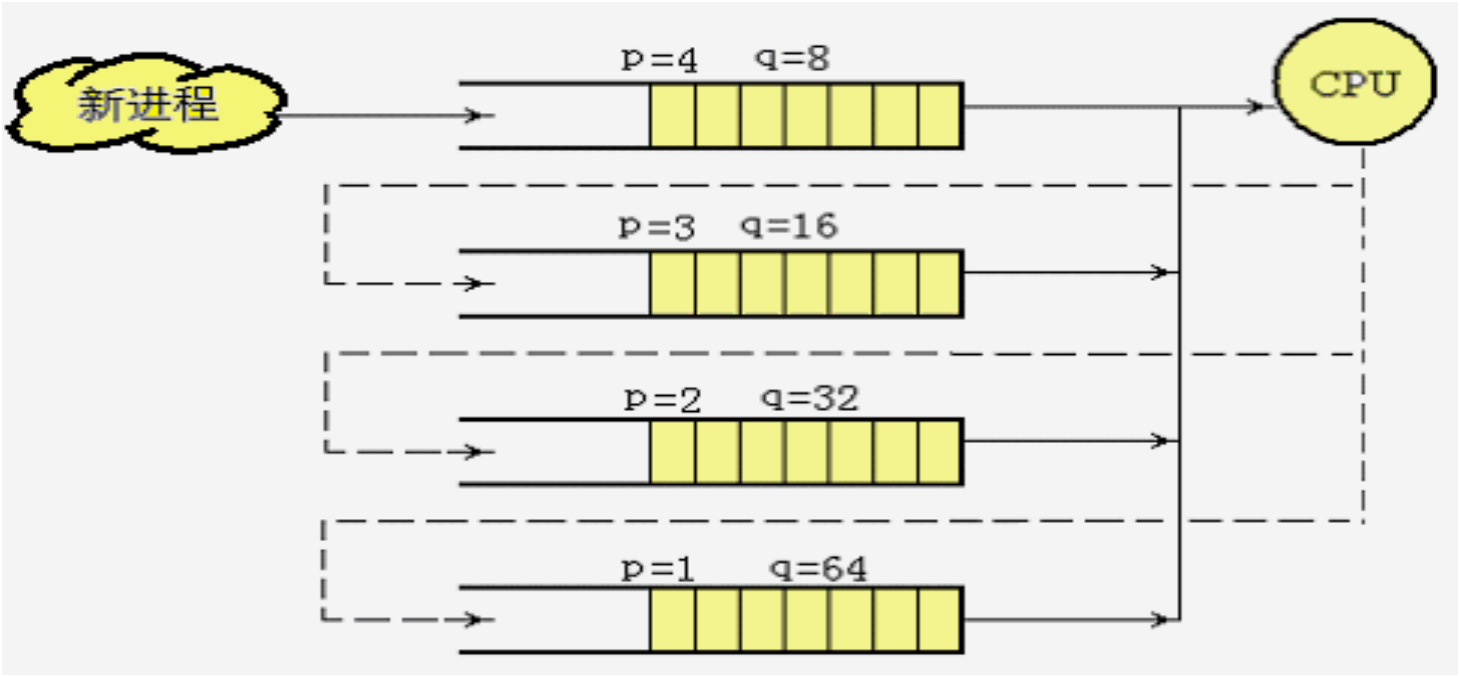
- Round Robin Scheduling
  - list of runnable processes
  - list of runnable processes after B uses up its quantum

Scheduling in Interactive Systems (2)



A scheduling algorithm with four priority classes

Scheduling in Interactive Systems (3)



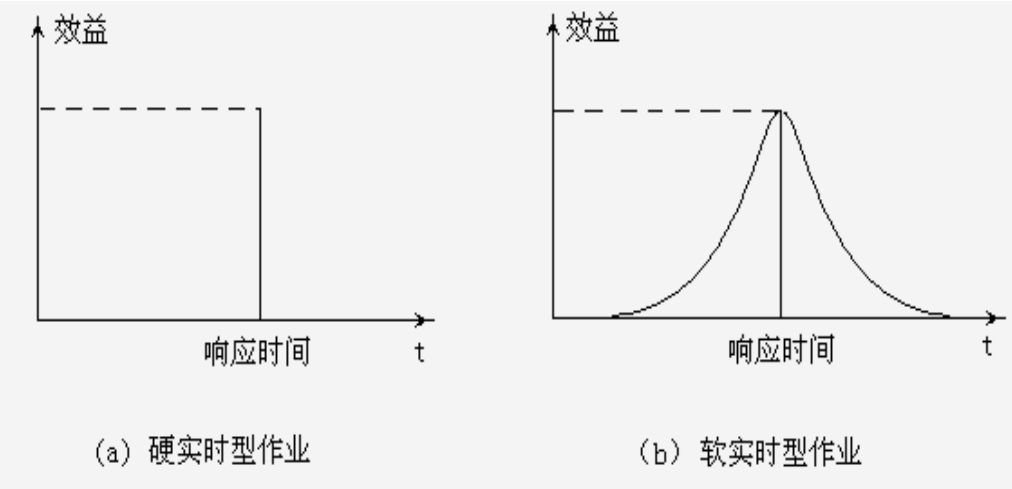
Multiple Queues

Scheduling in Real-Time Systems

- 要求更详细的调度信息：如，就绪时间、开始或完成截止时间、处理时间、资源要求、绝对或相对优先级（硬实时或软实时），优先级用户可控。
- 采用抢先式调度
- 快速中断响应：在中断处理时（硬件）关中断的时间尽量短。
- 快速上下文切换：相应地采用较小的调度单位（如线程）。提高响应时间。

Scheduling in Real-Time Systems

1、硬实时和软实时



(a) 硬实时型作业　　(b) 软实时型作业

2、周期性和非周期性

Scheduling in Real-Time Systems

周期性任务调度

- 在一些信号检测和过程控制系统中，有许多任务呈现周期性的运行规律。比如，气象信息检测中每隔2小时要读取一次数据，窑炉控制中每隔5分钟需检测一次炉温。这些任务的共同特点是，周期性强，而且有固定的时间间隔和相同的工作流程。通常，我们将这类任务称为周期性任务。目前，生产过程中的大多数数据信号采集系统，精密的过程检测与控制系统都属于此类。

- 一个周期性任务进入时，需要向系统提交的信息有：代码长度和资源需求，还要包括间隔周期和每个周期内的执行时间等。

Scheduling in Real-Time Systems

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

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

Scheduling in Real-Time Systems

非周期性任务

- 紧迫型实时任务调度
  - 立即抢占式优先级调度
- 普通型的实时任务调度
  - 基于时钟中断的抢占式优先级调度
- 宽松型的实时任务调度
  - 非抢占的HPF调度算法
  - RR算法

Scheduling in Real-Time Systems

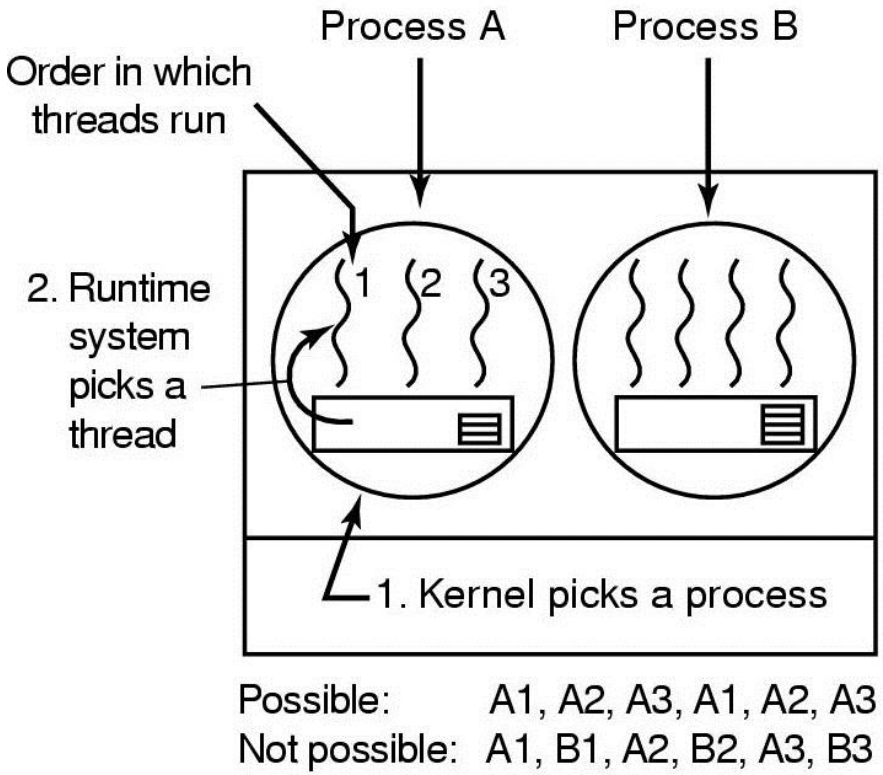
最早截止时间优先调度算法

根据任务的截止时间来确定任务的优先级，截止时间越早，其优先级越高，称为最早截止时间优先算法。

在系统中保持一个实时任务就绪队列，该队列按各任务截止时间的早晚排序，具有最早截止时间的任务排在队列的最前面。调度程序在选择任务时，总是选择队列中的第一个任务，为之分配处理机，使之投入运行。

该算法既可用于抢占式调度，也可用于非抢占式调度。

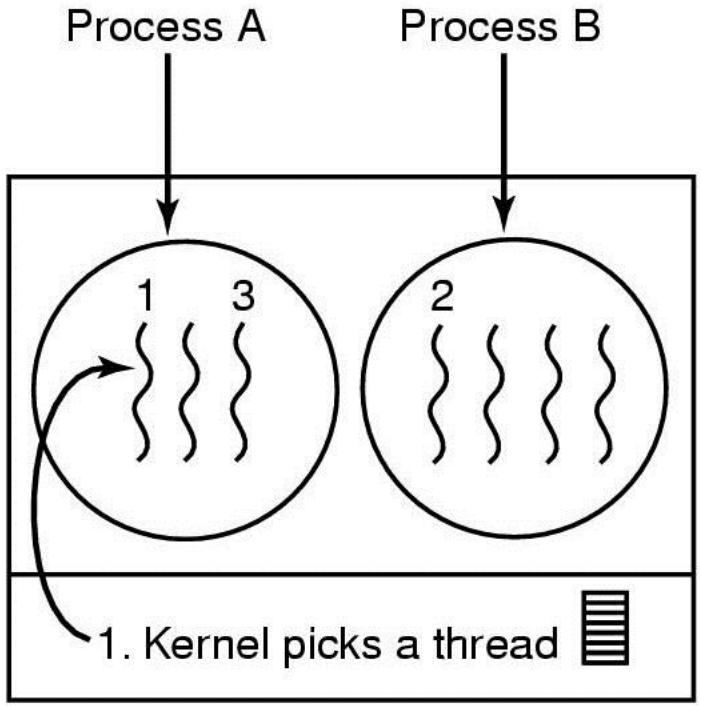
Thread Scheduling (1)



Possible scheduling of user-level threads

- 50-msec process quantum
- threads run 5 msec

Thread Scheduling (2)



Possible scheduling of kernel-level threads

- 50-msec process quantum
- threads run 5 msec

Policy versus Mechanism

- Separate the **scheduling mechanism** from the **scheduling policy**
  - a process knows which of its children threads are important and need priority
- Scheduling algorithm parameterized
  - mechanism in the kernel
- Parameters filled in by user processes
  - policy set by user process

PROBLEMS(HOMEWORKS)

1、7、12、16、24、43、45、50