Chapter 2
Processes and Threads

2.1 Processes
2.2 Threads
2.3 Inter-process communication
2.4 Classical IPC problems
2.5 Scheduling

Inter-process Communication

- Three issues are involved in inter-process communication (IPC)
  1. How one process can pass information to another.
  2. Resource shared (e.g. printer)
     - How to make sure two or more processes do not get into each other's way when engaging in critical activities. (Mutual Exclusion)
  3. Process cooperation
     - Proper sequencing when dependencies are present. (Synchronization)

Process Synchronization

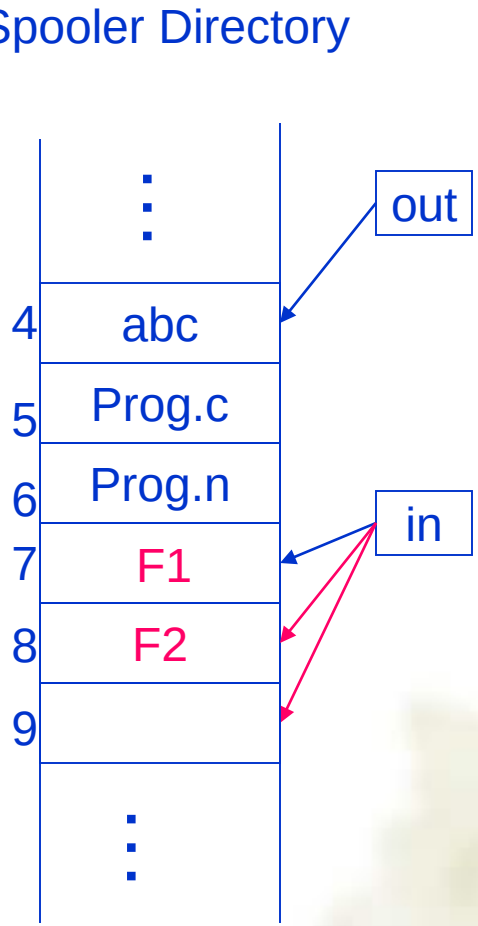
- 进程同步：对多个相关进程在执行次序上的协调，用于保证这种关系的相应机制称为进程同步。（或相互合作的一组并发进程在一些关键点上可能需要互相等待与互通消息，这种相互制约的等待与互通消息称为进程同步。）
- 例：医生为病员诊病，认为需要化验，开出化验单。病员取样送到化验室，等化验完毕交给医生化验结果，继续诊病。医生诊病是一个进程，化验室的化验工作是另一个进程，它们各自独立的活动，但又共同完成医疗任务。化验进程只有在接收到诊病进程的化验单后才开始工作；而诊病进程只有获得化验结果后才能继续为该病员诊病，并根据化验结果确定医疗方案。
- 例：计算进程与打印进程共享一个单缓冲区的问题。

Spooling Example: Correct

Process 1

int next_free;

1. next_free = in;
2. Stores F1 into next_free;
3. in=next_free+1



Process 2

int next_free;

4. next_free = in
5. Stores F2 into next_free;
6. in=next_free+1

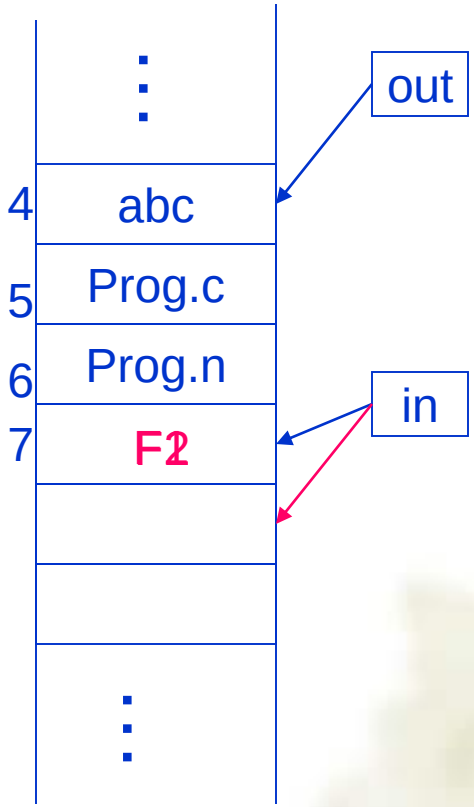
Spooling Example: Races

Process 1

int next_free;

1. next_free = in;
   /* value: 7 */

5. Stores F1 into next_free;
6. in=next_free+1



Process 2

int next_free;

2. next_free = in
   /* value: 7 */
3. Stores F2 into next_free;
4. in=next_free+1

Race conditions

Race conditions are situations in which several processes access shared data and the final result depends on the order of operations.

The key to avoid race conditions is to prohibit more than one process from reading and writing the shared data at the same time.

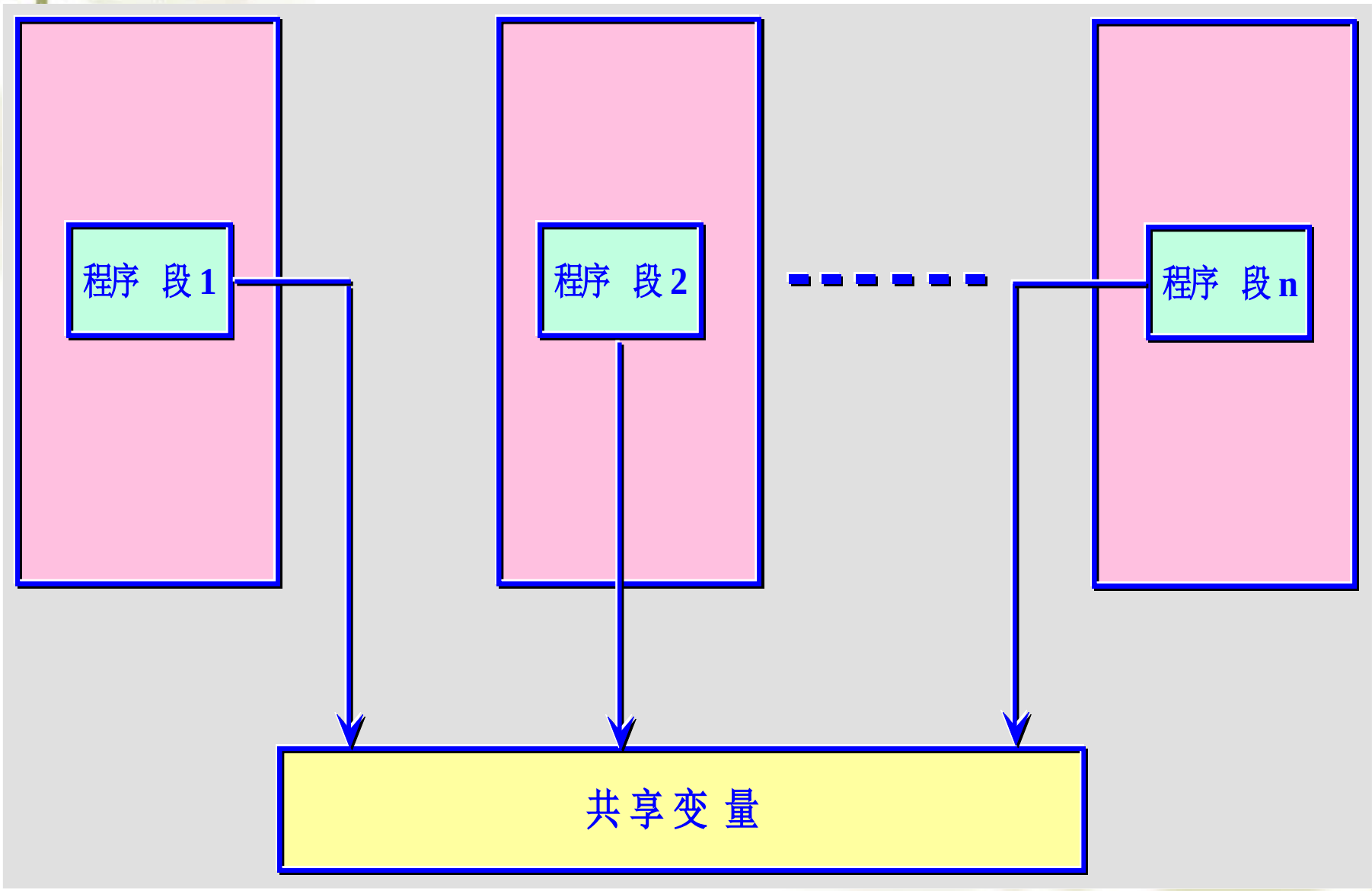
Mutual exclusion

Some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

- Critical Resource(临界资源)
  - 一次仅允许一个进程访问的资源称为临界资源
  - 临界资源包括：
    - 硬件资源：输入机、打印机等
    - 软件资源：变量、表格、队列、文件等

- Critical Region (Critical Section)
  - The part of the program where the critical resource is accessed is called critical region or critical section.

If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races.


程序 段1
程序 段2
程序 段n
共享变量

P_1

```
a := a +1
print (a)
```

P_2

```
a := a -1
print (a)
```

P_3

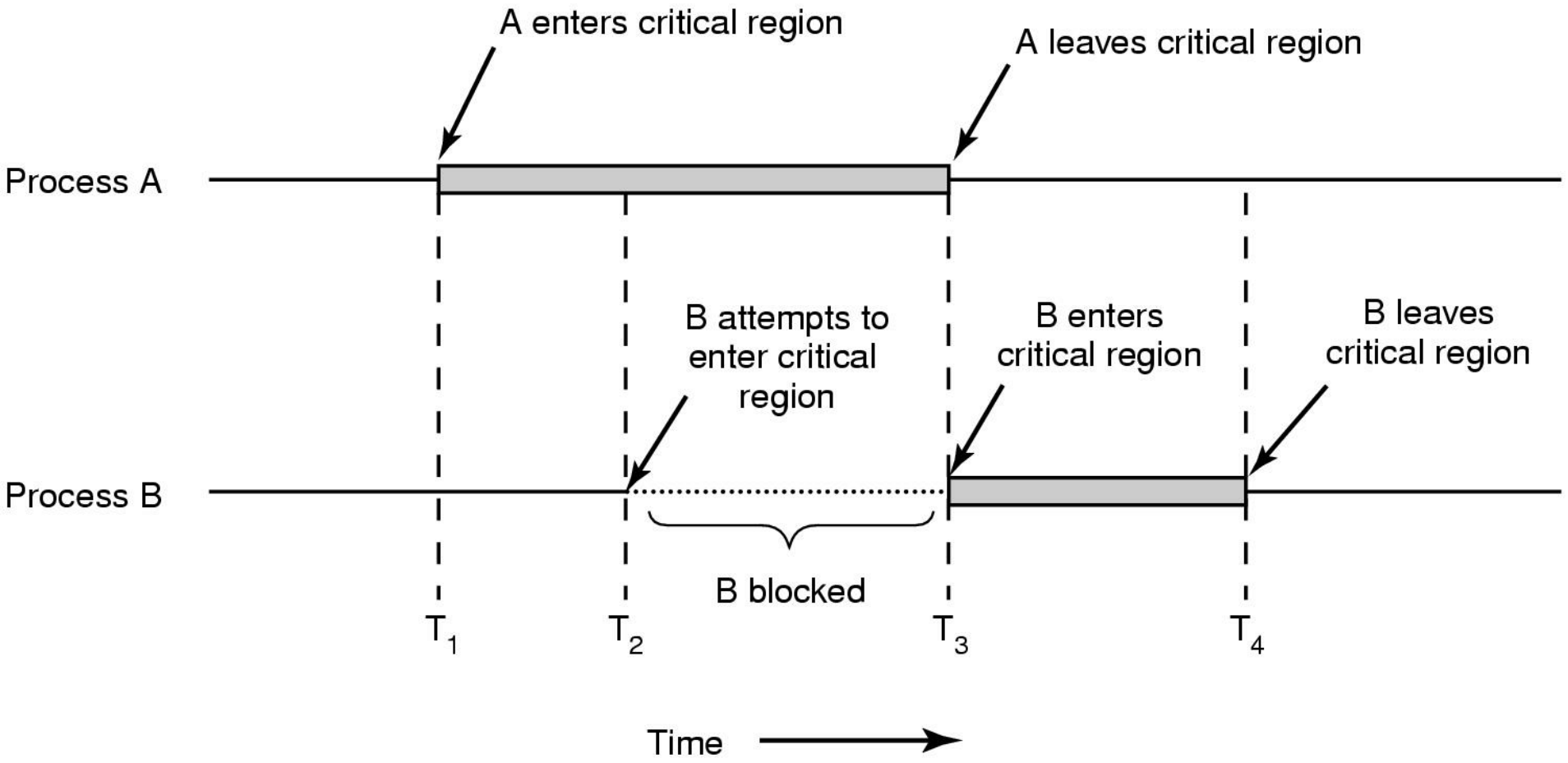
```
If a < 0
then
a := a +1
else
a:= a-1
```

Mutual exclusion

```
entry section
    critical section
exit section
    remainder section
```

Critical Region Requirement

- 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



Mutual exclusion using critical regions

Mutual Exclusion

- Possible Solutions
  - Disabling Interrupts
  - Lock Variables
  - Strict Alternation
  - Peterson’s solution
  - TSL
  - Sleep and Wakeup

Solution 1 - Disabling Interrupts

- How does it work?
  - Disable all interrupts just after entering a critical section and re-enable them just before leaving it.
- Why does it work?
  - With interrupts disabled, no clock interrupts can occur. (The CPU is only switched from one process to another as a result of clock or other interrupts, and with interrupts disabled, no switching can occur.)
- Problems:
  - What if the process forgets to enable the interrupts?
  - Multiprocessor? (disabling interrupts only affects one CPU)
- Only used inside OS

Solution 2 - Lock Variable

```
shared int lock = 0;
  /* entry_code: execute before entering critical section */
  while (lock != 0) /* do nothing */ ;
  lock = 1;

      - critical section -

  /* exit_code: execute after leaving critical section */
  lock = 0;
```

What happens if there is a context switch here?

Two (or more) processes may be able to enter their critical sections at the same time. This may violate property 1.

These sequence needs to be atomic.

Atomic means that the code cannot be interrupted during execution.

Solution 3 - Strict Alternation

```
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)

This solution may violate property 3. Since the processes must strictly alternate entering their critical sections, a process wanting to enter its critical section twice in a row will be blocked until the other process decides to enter (and leave) its critical section.

Solution 4 - Peterson's

process i

```
enter_region ( i );
Critical region
leave_region ( i );
Noncritical region
```

Solution 4 - Peterson's

```
#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);
{
      int other;

      other = 1 – 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)
{
      interested[process] = FALSE;
}
```

Mutual exclusion: Enter critical section if and only if

- Other process does not want to enter.
- Other process wants to enter, but your turn.

- A process can enter its critical section twice.
- The process running outside its critical region can not block another process.

Peterson's solution for achieving mutual exclusion

Hardware solution 5: Test-and-Set Locks (TSL)

The hardware must support a special instruction, tsl, which does 2 things in a single atomic action:

tsl register, flag

(a) copy a value in memory (flag) to a CPU register

(b) set flag to 1.

Test-and-Set Locks (TSL)

```
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

返回

Test-and-Set Locks (TSL)

XCHG instruction

- Exchange the contents of two locations atomically.
- All Intel x86 CPUs use XCHG instruction for low-level synchronization.

```
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.

Mutual Exclusion with Busy Waiting

- The last two solutions, 4 and 5, require BUSY-WAITING; that is, a process executing the entry code will sit in a tight loop using up CPU cycles, testing some condition over and over, until it becomes true.

- Busy-waiting may lead to the PRIORITY-INVERSION PROBLEM if simple priority scheduling is used to schedule the processes.

Mutual Exclusion with Busy Waiting

- Example: Test-and-set Locks:

P0 (low) - in cs -x
|
context
switch
|
P1 (high) -----tsl-cmp-jne-tsl... x-tsl-cmp... x-... forever.

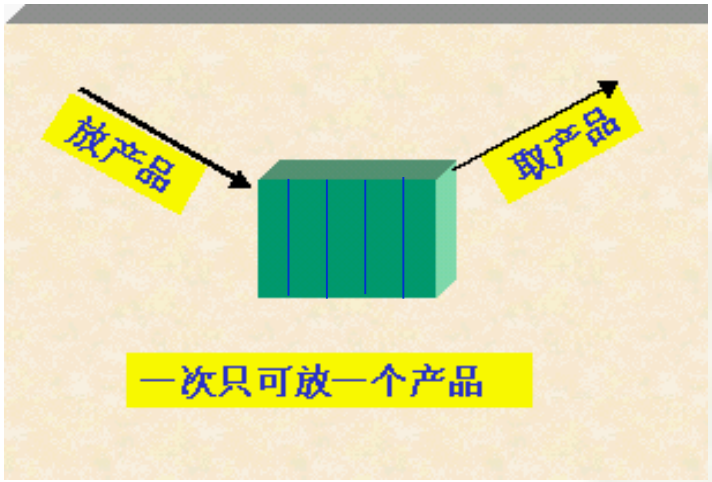
- Note, since priority scheduling is used, P1 will keep getting scheduled and waste time doing busy-waiting. :-(
- Thus, we have a situation in which a low-priority process is blocking a high-priority process, and this is called PRIORITY-INVERSION.

Sleep and Wakeup

- Solution of previous problems: sleep and wakeup (busy waiting)
  - Block instead of busy waiting when it is not allowed to enter the Critical Region (sleep)
  - Wakeup when it is OK to retry entering the critical region

Producer-Consumer Problem (Bounded-buffer problem)

- Consider two processes share a circular buffer that can hold N items.
- Producers add items to the buffer and Consumers remove items from the buffer.
- The Producer-Consumer Problem is to restrict access to the buffer so correct executions result.



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 */
     }
}
```

The wakeup signal is lost. Both will sleep forever.

What happens if consumer is interrupted after reading count as 0?

Producer-consumer problem with fatal race condition

Sleep and Wakeup

- Solution: add a wakeup waiting bit
  - When a wake up is sent to a process that is still awake, this bit is set to 1.
  - Later, when the process tries to go to sleep, if the wakeup bit is on, it will be turn off, but the process is still awake.

Semaphores

- Semaphores introduced by E.W.Dijkstra in 1965 to count the number of “wakeups” saved. Its value could be
  - 0: no wakeups were saved
  - Some positive value: one or more wakeups were pending
- Higher-level synchronization mechanism
- Higher than disabling interrupts.
  - Fine for short sequences of code in kernel.
  - Not fine for application-level use.

Semaphores

- A semaphore, S, is a structure consisting of two parts:
  (a) an integer counter, COUNT
  (b) a queue of pids of blocked processes, Q

- That is,

```
struct sem_struct {
        int count;
        queue Q;
} semaphore;

semaphore S;
```

Semaphores

- A semaphore count represents count number of abstract resources
  - Counting semaphores: 0..N (initialized to N)
  - Binary semaphores: 0,1(initialized to 1)
- There are 2 operations on semaphores
  - P(sem S) or wait(sem S) or down (sem S): used to acquire a resource and decrements count.
  - V(sem S) or signal(sem S) or up(sem S): used to release a resource and increments count.
- Any semaphore operation is indivisible, must be executed atomically. (what is primitive?)

Semaphores

- Suppose that P is the process making the down and up system calls. The operations are defined as follows:

P操作 DOWN(S):

```
S.count = S.count - 1;   //apply for a resource
if (S.count < 0)         //if  no resource
      block(P);
```

(a) enqueue the pid of P in S.Q,
(b) block process P (remove the pid from the ready queue), and
(c) pass control to the scheduler.

执行一次P操作后，若S.count<0，则|S.count|等于Q队列中等待S资源的进程数。

Semaphores

V操作 UP(S):

```
S.count = S.count + 1;      //release a resource
if ( S.count<= 0 )          // have process blocked
      wakeup(P) for some process P in S.Q;
```

(a) remove a pid from S.Q (the pid of P),
(b) put the pid in the ready queue, and
(c) pass control to the scheduler.

执行一次V操作后，若S.count≤0，则表明Q队列中仍有因等待S资源而被阻塞的进程，所以需要唤醒(wakeup)Q队列中的进程。

Semaphores

- How do we ensure that the semaphore primitives is atomic?
  - Approach 1
    - Make them system calls, and ensure only one P() or V() operation can be executed by any process at a time.
    - This effectively puts a lock around the P() and V() operations themselves!
    - Easy to do by disabling interrupts in the P() and V() calls.
  - Approach 2
    - Use hardware support.
    - Say your CPU had atomic P and V instructions.

The producer-consumer problem using 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 */
    }
}
```

exchange

- In conclusion, we use semaphores in two different ways:
  - mutual exclusion (mutex);
  - process synchronization (full, empty).
- Is it easy to use semaphores?

Mutex: Binary Semaphore

- Simplified version of semaphore
- Mutex is used for mutual exclusion problem
- A variable with only 2 states
  - Lock
  - Unlock

Mutual Exclusion Problem using Mutex

```
semaphore mutex = 1;   /* set mutex.count = 1 */

DOWN(mutex);
        - critical section -
UP(mutex);
```

- Binary semaphore has an initial value of 1.
- DOWN(mutex) is called before a critical section.
- UP(mutex) is called after the critical section.

- For two parallel processes, the value of mutex is 1, 0, -1
  For N parallel processes, the value range of mutex is 1~ -(N-1)

Quiz

If the initial value of semaphore S is 2 in a down( ) and up( ) operation, its current value is -1, that means there are ___ processes are waiting.

A.0

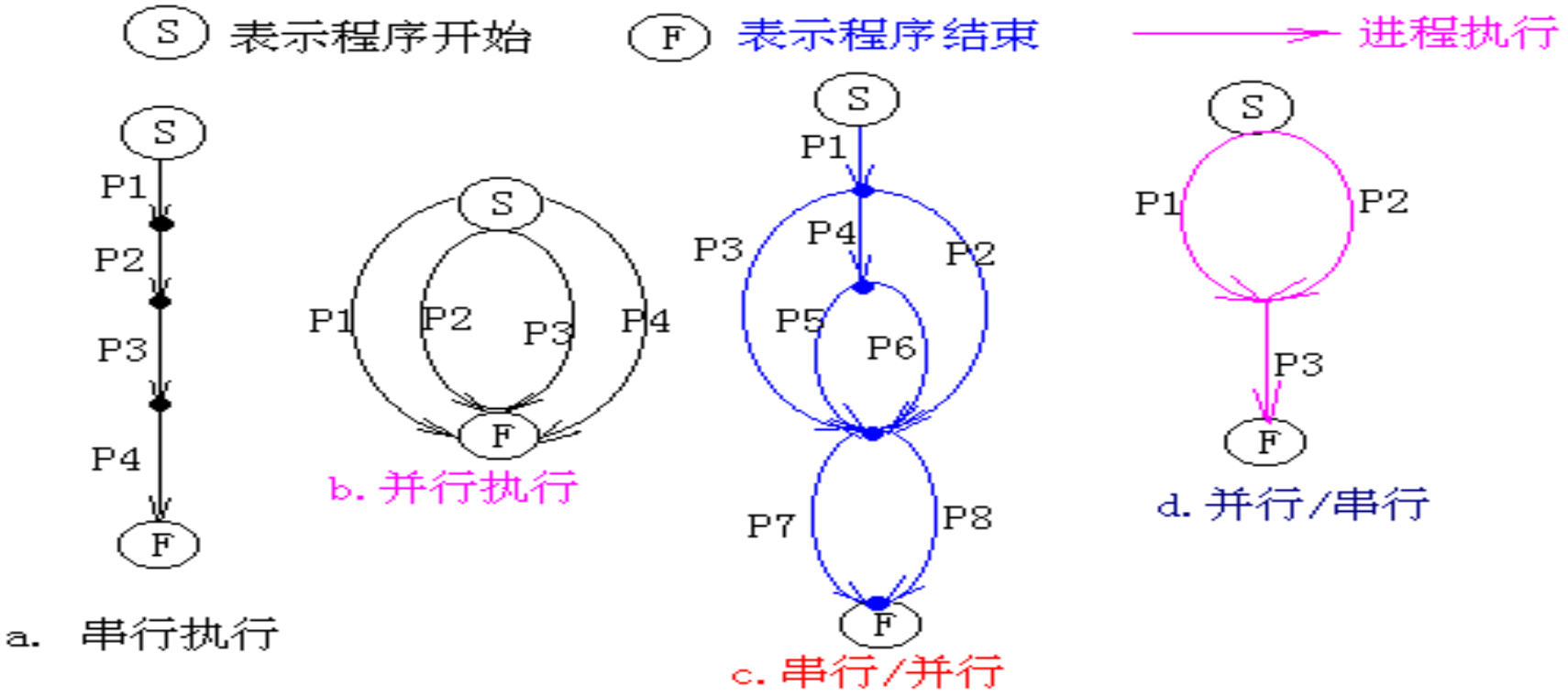
B.1

C.2

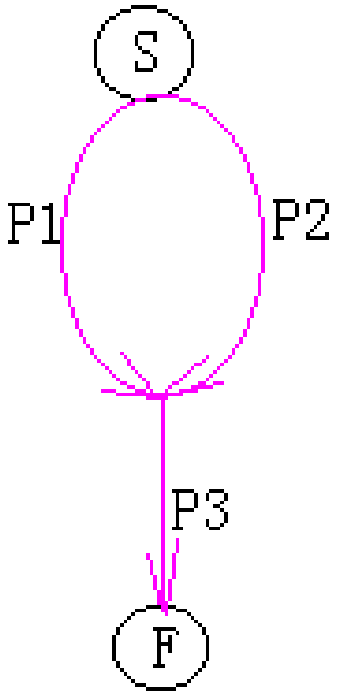
D.3

Using Semaphores

- Process Synchronization: Order process execution

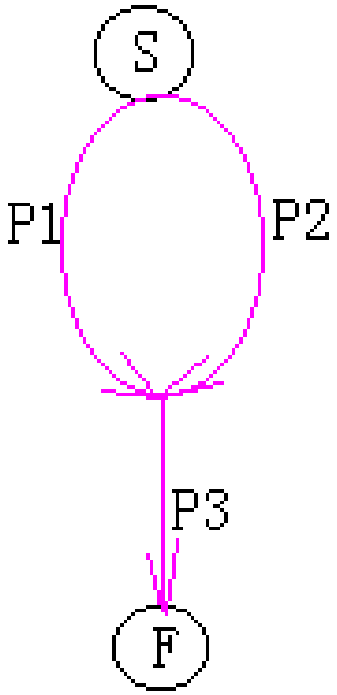


Using Semaphores



- Suppose we have 3 processes: P1, P2, and P3. P1 and P2 must finish executing before P3.
- The processes may be synchronized using semaphores:
  - S1=0: P1 still unfinished;
  - S2=0: P2 still unfinished;

Using Semaphores

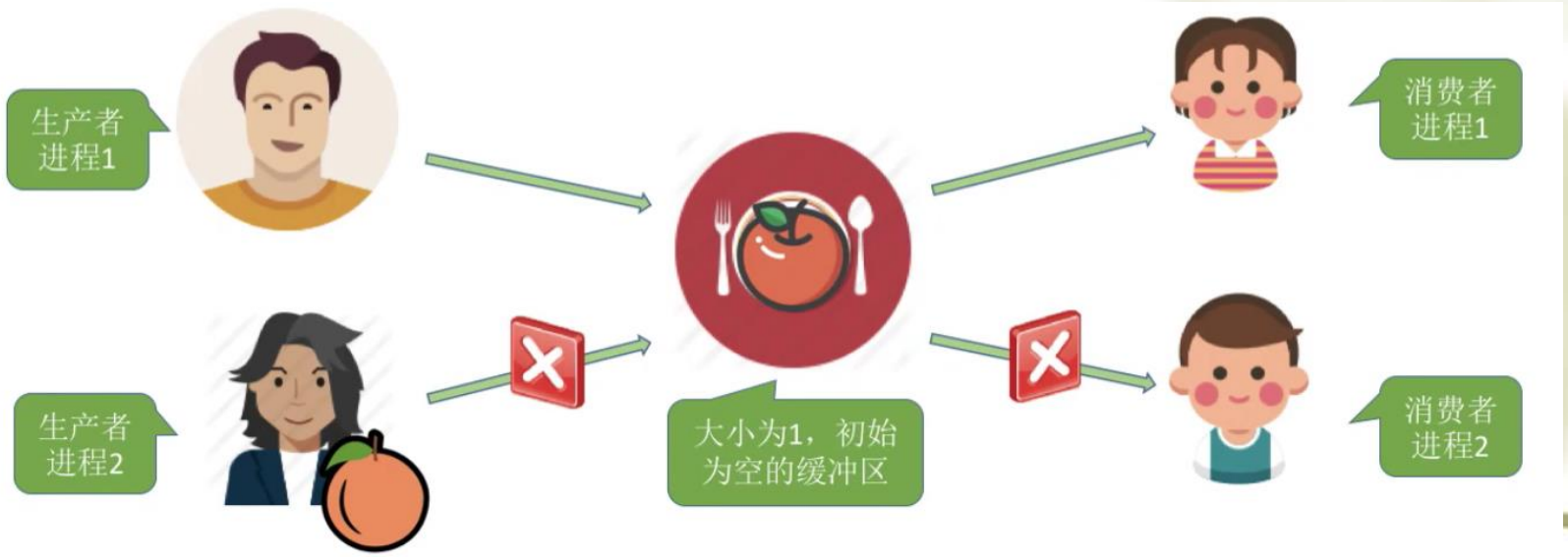


P1()	P2()	P3()
{	{	{
……	……	DOWN(S1);
UP(S1);	UP(S2);	DOWN(S2);
}	}	……
		}

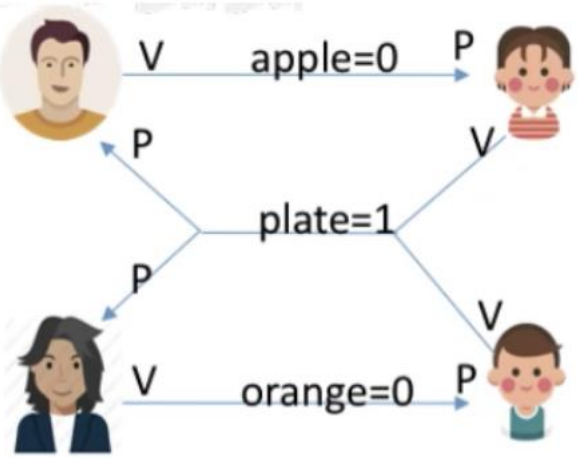
Example

桌上有一只盘子，只能放一个水果，爸爸专向盘中放苹果，妈妈专向盘中放橙子，儿子专等吃盘里的橙子，女儿专等吃盘里的苹果。只要盘子空，则爸爸或妈妈可向盘中放水果，仅当盘中有自己需要的水果时，儿子或女儿可从中取出。

请给出四人之间的同步关系，并用P、V操作实现四人正确活动的程序。



- 由于父亲和母亲都可向盘子放水果，设置信号量plate=1，表示可放一个水果;
- 父亲和女儿、母亲和儿子存在同步关系，分别设置信号量apple, orange=0。



Father:	**Daughter:**
P(plate); 向盘中放苹果; V(apple);	P(apple); 取盘中苹果; V(plate);
Mother:	**Son:**
P(plate); 向盘中放橙子; V(orange);	P(orange); 取盘中橙子; V(plate);

为什么不用互斥信号量

如果儿子或女儿进程先被调度执行也会因为没有水果可取而被阻塞；

如果父亲先执行，则P(plate)，可以放水果(苹果)，而母亲如果此时也执行，则P(plate)时会被阻塞以等待盘子；

父亲放入苹果后通过V(apple)唤醒女儿，其他进程还是阻塞，不会访问临界资源盘子；

女儿P(apple)访问盘子，取走苹果V(plate)，唤醒母亲，母亲访问盘子放橙子，其他进程阻塞……

为什么不用互斥信号量

主要原因：

缓冲区大小为1，任何时刻，apple、orange和plate三个同步信号量中最多只有一个是1。

因此，在任何时刻，最多只有一个进程的P操作不会被阻塞，并顺利地进入临界区。

Quiz1

桌子上有一只盘子，最多可容纳两个水果，每次只能放入或取出一个水果。爸爸专向盘子放苹果，妈妈专向盘子中放橙子；两个儿子专等吃盘子中的橙子，两个女儿专等吃盘子中的苹果。

请用PV操作来实现爸爸、妈妈、儿子、女儿之间的同步与互斥关系。

- 设置信号量plate=2, 对允许向盘中放入水果的个数进行计数。
- 每次只能放入或取出一个水果，因此需要对盘子进行互斥访问。

Father: P(plate); P(mutex); 向盘中放苹果； V(mutex); V(apple); **Mother:** P(plate); P(mutex); 向盘中放橙子； V(mutex); V(orange);	**Daughter i (i=1,2):** P(apple); P(mutex); 取盘中苹果； V(mutex); V(plate); **Son i (i=1,2):** P(orange); P(mutex); 取盘中橙子； V(mutex); V(plate) ;

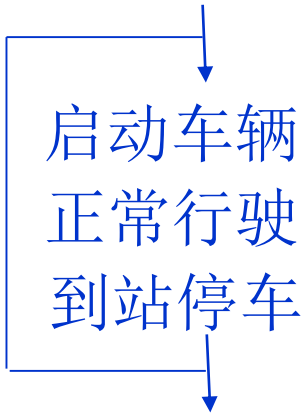
- 如果缓冲区大小大于1，就必须专门设置一个互斥信号量来保证互斥访问缓冲区。
- 如果缓冲区大小为1，有可能不需要设置互斥信号量就可以互斥访问缓冲区。

Quiz2

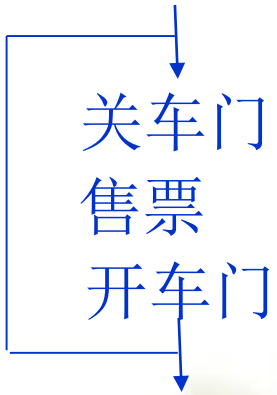
【问题描述】：

设公共汽车上有一位司机和一位售票员，它们的活动如下：

司机：



售票员：

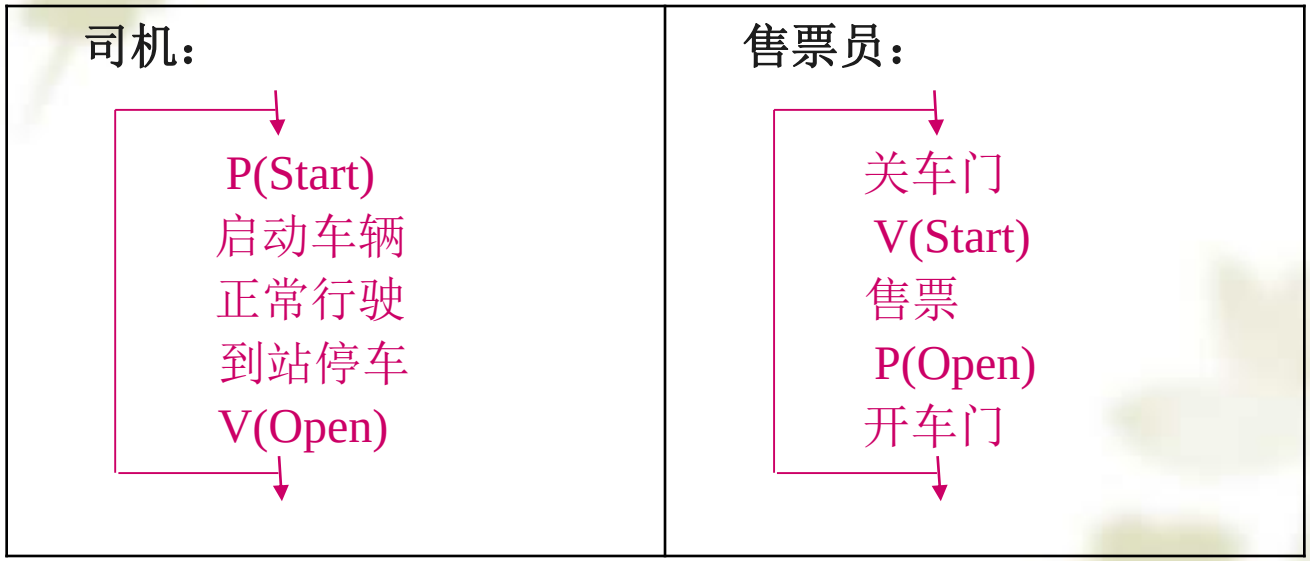


请分析汽车在不断的到站、停车、行驶过程中司机与售票员之间的同步关系，并用PV操作实现。

【分析】

- 第一步，确定进程间的关系。
  - 只有当售票员关车门之后司机才能启动车辆，只有当司机到站停车后售票员才能开车门。所以司机与售票员之间是一种同步关系。
- 第二步，确定信号量及其初值。
  - 由于司机与售票员之间要互通消息，司机进程设置一个信号量是start，用于判断司机能否启动车辆，初值为0。
  - 售票员进程设置一个信号量open，售票员是否能够开车门，初值为0。

【解答】



PV操作题目的解题思路

1. 关系分析

找出题目中描述的各个进程，分析它们之间的同步、互斥关系

2. 设置信号量

设置需要的信号量，并根据题目条件确定信号量初值

(互斥信号量初值一般为1，同步信号量初值要看对应资源的初始值是多少)

讨论：

1) 信号量的物理含义

S>0表示有S个资源可用

S=0表示无资源可用

S<0则|S|表示S等待队列中的进程个数

P(S)表示申请一个资源，V(S)表示释放一个资源

信号量的初值应该大于等于0

2) P,V操作必须成对出现，有一个P操作就一定有一个V操作

- 当为互斥操作时，它们同处于同一进程；
  当为同步操作时，则不在同一进程中出现。
- 如果P(S1)和P(S2)两个操作在一起，那么P操作的顺序至关重要，一个同步P操作与一个互斥P操作在一起时同步P操作在互斥P操作前；
- 而两个V操作的顺序无关紧要。

3) 信号量同步的缺点

用信号量可实现进程间的同步，但由于信号量的控制分布在整个程序中，其正确性分析很困难。

4) 引入管程

- 1973年，Hoare和Hanson提出一种高级同步原语——管程；其基本思想是把信号量及其操作原语封装在一个对象内部。
- 管程是管理进程间同步的机制，它保证进程互斥地访问共享变量，并方便地阻塞和唤醒进程。
- 管程可以函数库的形式实现。相比之下，管程比信号量好控制。

Monitors

- A **monitor** is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- An example of a monitor in Pidgin Pascal

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

- Monitors are a programming language construct.
  - Java, C#, etc., have built-in support for monitors.
  - C does not have.

Monitors

- Rules to Follow with Monitors
  - A process/thread enters the monitor by invoking one of its procedures.
  - Only one process/thread can be active within the monitor at a time.(mutual exclusion).
    - Any other process that has invoked the monitor is suspended, waiting for the monitor to become available.
  - No process can directly access a monitor's local variables.
  - A monitor may only access its local variables.

Monitors

- Mutual exclusion
  - A monitor can only be accessed by one process at a time.
  - The **compiler** use a mutex or a binary semaphore to implement mutual exclusion on monitor entries.
- Process synchronization
  - Done by the programmer by using **condition variables**, which allow a process to wait within the monitor.
  - Condition variable can only be used with the operations **wait** and **signal**.
    - The operation **wait(x)** means that the process invoking this operation is suspended until another process invokes **signal(x).**
    - The signal(x) operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect.

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

Recall: 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 */
     }
}
```

What happens if consumer is interrupted after reading count as 0?

Message passing

Possible Approaches of IPC:

1) Shared memory

2) Shared file mode

pipe: is a shared file

- One end is for reading and one end is for writing.

3) Message passing:

primitive: send and receive

- Assign each process a unique address such as addr. Then, send messages directly to the process:

send(addr, msg);

recv(addr, msg);

- Use mailboxes:

send(mailbox, msg);

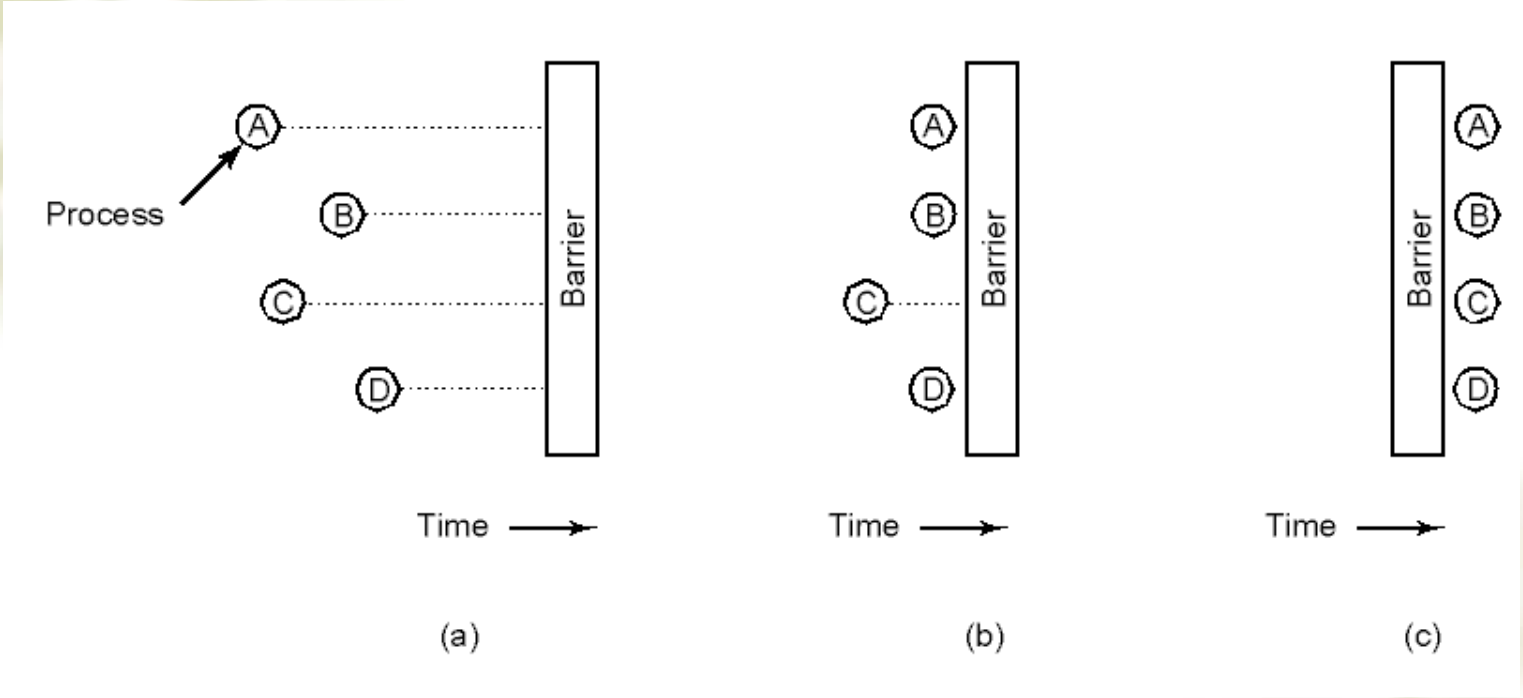
recv(mailbox, msg);

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
  a) processes approaching a barrier
  b) all processes but one blocked at barrier
  c) last process arrives, all are let through
- Example: Parallel matrix multiplication

Classical IPC Problems

- These problems are used for testing every newly proposed synchronization scheme:
  - Bounded-Buffer (Producer-Consumer) Problem
  - Dining-Philosophers Problem
  - Readers and Writers Problem

Dining Philosophers

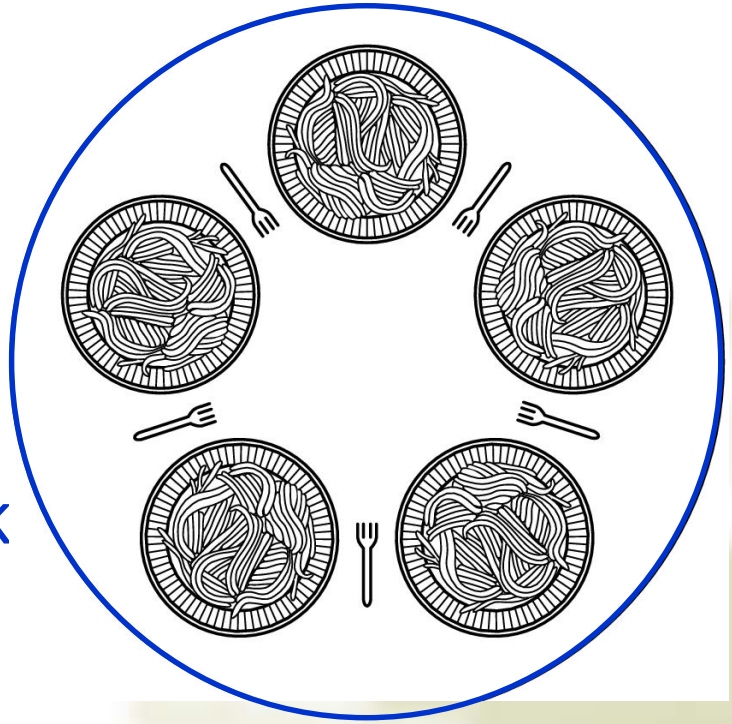
- Dining Philosophers Problem [Dijkstra, 1965]:

  Problem:

  - Five philosophers are seated around a table. There is one fork between each pair of philosophers.
  - Each philosopher needs to grab the two adjacent forks in order to eat.
  - Philosophers alternate between eating and thinking.
  - They only eat for finite periods of time.

Dining Philosophers

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



Dining Philosophers

```
#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

- Problem:
  - Suppose all philosophers execute the first DOWN operation, before any have a chance to execute the second DOWN operation; that is, they all grab one fork. Then, deadlock will occur and no philosophers will be able to proceed. This is called a CIRCULAR WAIT.
- Other Solutions:
  - Only allow up to four philosophers to try grabbing their forks.
  - Asymmetric solution: Odd numbered philosophers grab their left fork first, whereas even numbered philosophers grab their right fork first.
  - Protect the five statements following *think()* by mutex
  - Pick-up the forks only if both are available. See Fig. 2-46.

Dining Philosophers

```
#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

```
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)

Readers and Writers Problem

- The readers and writers problem models access to a shared database.
- Rules:
  - Multiple readers can read the data simultaneously
  - Only one writer can write the data at any time
  - A reader and a writer cannot in critical section together.

	Reader	Writer
Reader	OK	No
Writer	No	No

Readers and Writers Problem

- One solution (reader preference):
  - Suppose new reader come: if writer is waiting and some reader is reading, then read ok.
  - Suppose new writer come: if no reader or writer, it can write, else wait.
- One variation of the problem, called weak readers preference, is to suspend the incoming readers as long as a writer is waiting.

Readers and Writers Problem

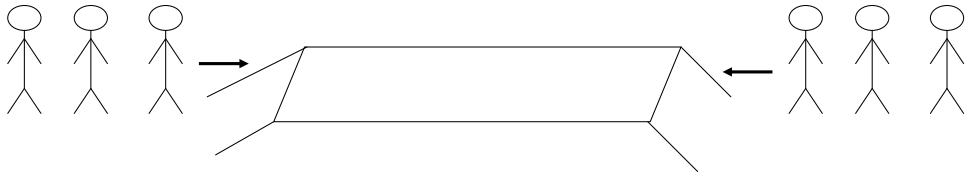
```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to 'rc' */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

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 */
    }
}
```

entry

exit

A solution to the readers and writers problem

As illustrated in the figure, on the two sides of a single-plank bridge (独木桥), there are two groups of soldiers that are composed of *m* and *n* people respectively and need to cross the bridge, but the narrow bridge allows only one group of the soldiers in the same direction to cross at the same time. One group of the soldiers is permitted to cross as long as there are no people on the bridge. Once one group of the soldiers begins walking on the bridge, the other group should be waiting to start crossing until all members of the first group have passed the bridge.



Please design two semaphore-based processes to describe the crossing actions of the soldiers in the two groups. It is required to
(1)Define the semaphores and variables needed and give their initial values;
(2)Illustrate the structures of processes for the soldiers in each group.

该问题可归结为读写者问题，但两队士兵均为读者。

（1）定义两个整型变量count1、count2，分别用于计数每组士兵过桥人数，初值都为0。

定义两个二元信号量mutex1、mutex2，用于控制对count1、count2的互斥访问，初值都为1。

定义二元信号量bridge，用于两队士兵间的互斥，初值为1。

（2）

```
P1()
{
down (mutex1);
count1 ++;
if count1 = =1 down (bridge);  // 该组第1个士兵过桥时，阻塞另一组过桥
up (mutex1);
……
crossing the bridge;
……
down (mutex1);
count1 --;
if count1 = =0 up (bridge);  // 该组最后1个士兵过桥后，允许另一组过桥
up (mutex1);
}
```

```
P2()
    {
    down (mutex2);
    count2 ++;
    if count2 = =1 down (bridge);  // 该组第1个士兵过桥时，阻塞另一组过桥
    up (mutex2);
    …….
    crossing the bridge;
    ……
    down (mutex2);
    count2 --;
    if count2 = =0 up (bridge);  // 该组最后1个士兵过桥后，允许另一组过桥
    up (mutex2);
    }
```

Summary

- Race condition, Critical region and mutual exclusion
- Mutual exclusion using busy waiting
  - Peterson's Solution
  - TSL
- Sleep and Wakeup
- Semaphores
- Monitor
- Barrier
- Passing Message
- Classic synchronization problems
  - Dining-Philosophers Problem
  - Readers and Writers Problem

Homework

- 27、34、39、60