操作系统

第5章 并发性：互斥和同步 Concurrency: Mutual Exclusion and Synchronization

孙承杰
哈工大计算学部

E-mail： sunchengjie@hit.edu.cn

2025年秋季学期

Learning Objectives

- Discuss basic concepts related to concurrency
  - race conditions
  - OS concerns
  - mutual exclusion requirements
- Understand hardware approaches to supporting mutual exclusion
- Define and explain semaphores
- Define and explain monitors
- Explain the readers/writers problem

Outline

- **Mutual Exclusion: software approaches**
- **Principles of Concurrency**
  - A simple example
  - Race condition
  - Operating system concerns
  - Process interaction
  - Requirements for mutual exclusion
- **Hardware Support**
  - Interrupt disabling
- **Semaphores**
  - Mutual exclusion
  - The Producer/Consumer problem
  - Implementation of semaphores
- **Monitors**
  - Monitor with signal
- **Message Passing**
  - Synchronization
  - Addressing
  - Message format
  - Queueing discipline
  - Mutual exclusion
- **Readers/Writers Problem**
  - Readers have priority
  - Writers have priority

Designing correct routines for controlling concurrent activities proved to be one of the most difficult aspects of systems programming. The ad hoc techniques used by programmers of early multiprogramming and real-time systems were always vulnerable to subtle programming errors whose effects could be observed only when certain relatively rare sequences of actions occurred. The errors are particularly difficult to locate, since the precise conditions under which they appear are very hard to reproduce.

-- WHAT CAN BE AUTOMATED?: THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY, MIT Press, 1980

Concurrency

- **The central themes of OS design are all concerned with the management of processes and threads**
  - **Multiprogramming**
    - The management of multiple processes within a **uniprocessor** system
  - **Multiprocessing**
    - The management of multiple processes within a **multiprocessor** system
  - **Distributed processing**
    - The management of multiple processes executing on **multiple**, **distributed** computer systems.
    - clusters

Fundamental to all of these areas, and fundamental to OS design, is **concurrency**

Concurrency Arises in Three Different Contexts

- **Multiple Applications**
  - Multiprogramming was invented to allow processing time to be dynamically shared among a number of active applications
- **Structured Applications**
  - As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes
- **Operating System Structure**
  - OS are themselves often implemented as a set of processes or threads

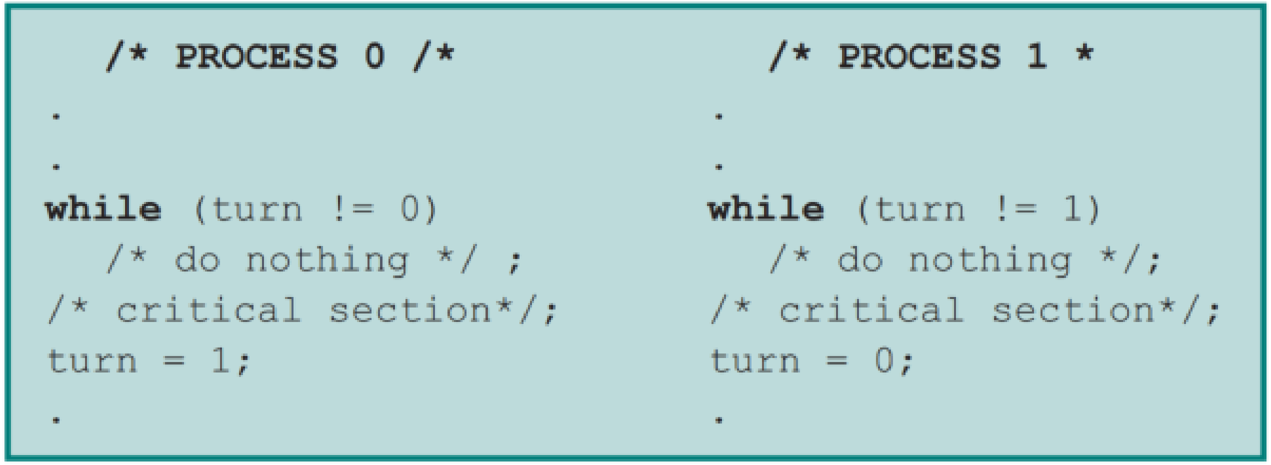
Some Key Terms Related to Concurrency

Atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
Critical section	A section of code within a process that requires access to shared resources, and that must not be executed while another process is in a corresponding section of code.
Deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
Livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
Mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
Race condition	A situation in which multiple threads or processes read and write a shared data item, and the final result depends on the relative timing of their execution.
Starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Mutual Exclusion: software approaches

- **Dekker’ s Algorithm**
  - Dijkstra [DIJK65] reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker.

```
int turn = 0;
```



```
   /* PROCESS 0 /*                    /* PROCESS 1 *
.                                  .
.                                  .
while (turn != 0)                  while (turn != 1)
   /* do nothing */ ;                 /* do nothing */;
/* critical section*/;             /* critical section*/;
turn = 1;                          turn = 0;
.                                  .
```

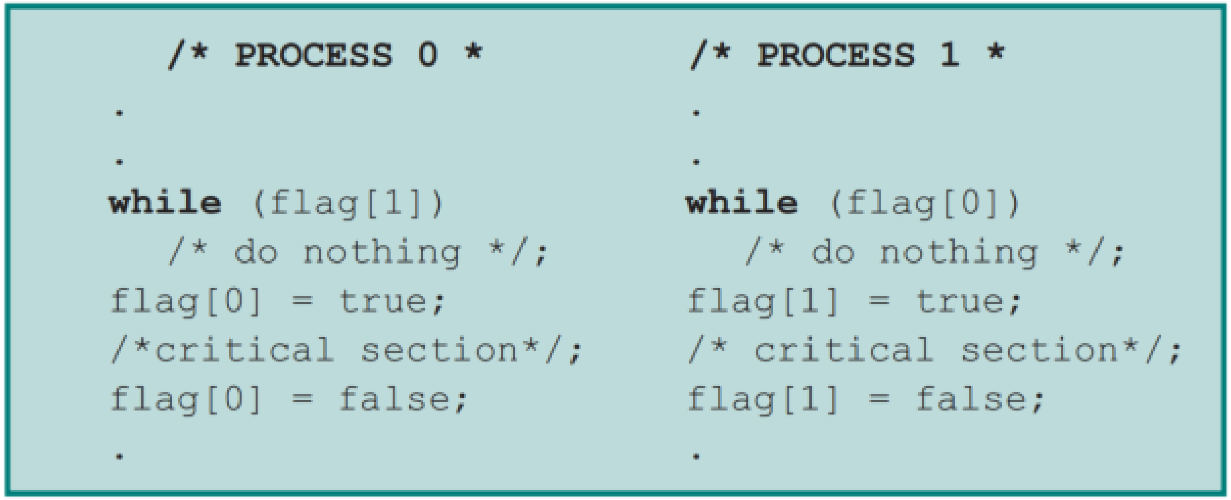
(a) First attempt

Mutual Exclusion: software approaches

- Dekker’ s Algorithm

enum boolean (false = 0; true = 1);

boolean flag[2] = 0, 0



```
    /* PROCESS 0 *               /* PROCESS 1 *
  .                            .
  .                            .
  while (flag[1])              while (flag[0])
     /* do nothing */;            /* do nothing */;
  flag[0] = true;              flag[1] = true;
  /*critical section*/;        /* critical section*/;
  flag[0] = false;             flag[1] = false;
  .                            .
```

(b) Second attempt

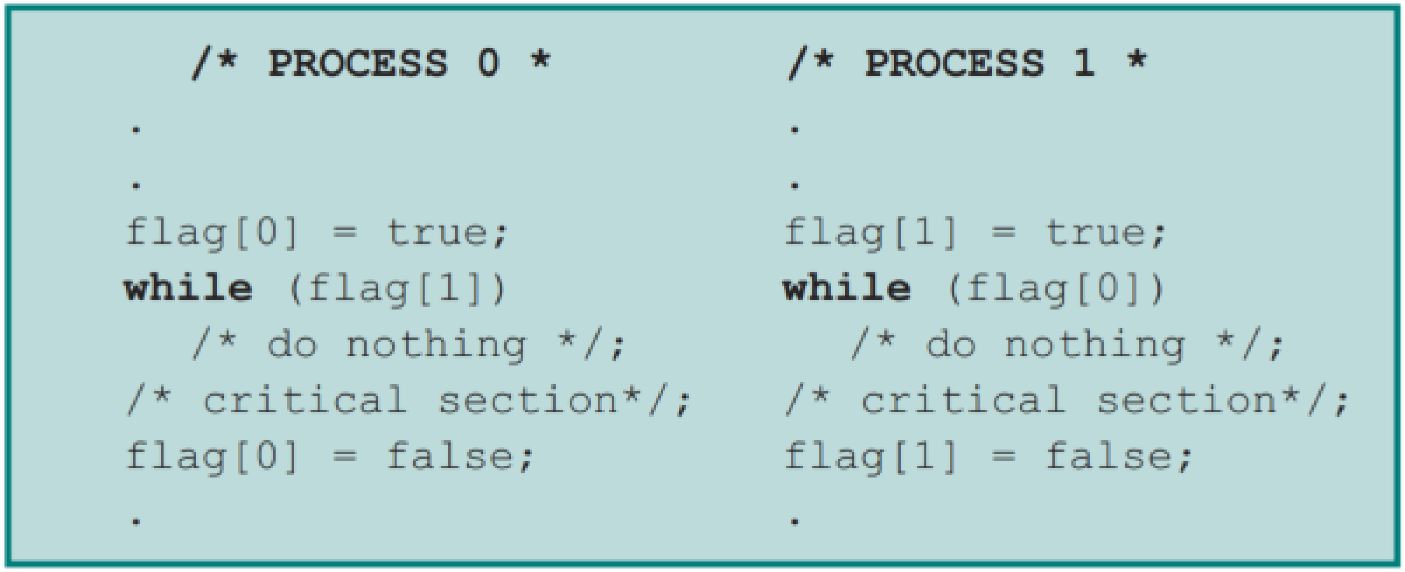
P0 executes the **while** statement and finds `flag[1]` set to `false`
Pl executes the **while** statement and finds `flag[0]` set to `false`
P0 sets `flag[0]` to `true` and enters its critical section
Pl sets `flag[1]` to `true` and enters its critical section

Mutual Exclusion: software approaches

- Dekker’ s Algorithm

  enum boolean (false = 0; true = 1);

  boolean flag[2] = 0, 0



```
   /* PROCESS 0 *              /* PROCESS 1 *
 .                            .
 .                            .
 flag[0] = true;              flag[1] = true;
 while (flag[1])              while (flag[0])
    /* do nothing */;            /* do nothing */;
 /* critical section*/;       /* critical section*/;
 flag[0] = false;             flag[1] = false;
 .                            .
```

(c) Third attempt

Mutual Exclusion: software approaches

- **Dekker’ s Algorithm**

enum boolean (false = 0; true = 1);

boolean flag[2] = 0, 0

```
  /* PROCESS 0 *            /* PROCESS 1 *
.                         .
.                         .
flag[0] = true;           flag[1] = true;
while (flag[1]) {         while (flag[0]) {
   flag[0] = false;          flag[1] = false;
   /*delay */;               /*delay */;
   flag[0] = true;           flag[1] = true;
}                         }
/*critical section*/;     /* critical section*/;
flag[0] = false;          flag[1] = false;
.
```

(d) Fourth attempt

```
P0 sets flag[0] to true.
Pl sets flag[1] to true.
P0 checks flag[1].
Pl checks flag[0].
P0 sets flag[0] to
false.
Pl sets flag[1] to
false.
P0 sets flag[0] to true.
Pl sets flag[1] to true.
```

Dekker's Algorithm

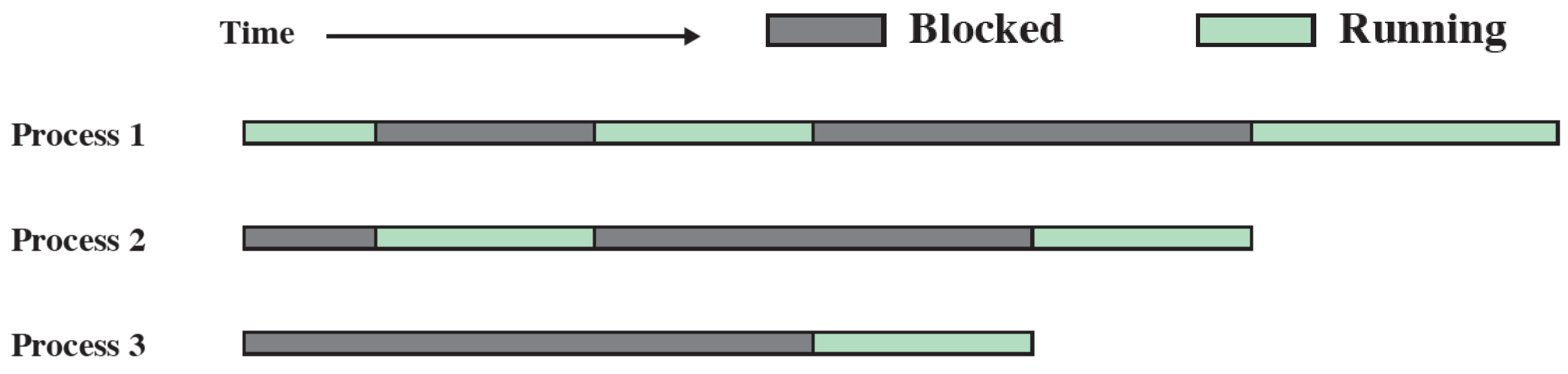
```
boolean flag [2];
int turn;
void P0()
{
      while (true) {
           flag [0] = true;
           while (flag [1]) {
                   if (turn == 1)
                        flag [0] = false;
                         while (turn == 1) /* do nothing */;
                         flag [0] = true;
                   }
           }
           /* critical section */;
           turn = 1;
           flag [0] = false;
           /* remainder */;
      }
}
void P1( )
{
      while (true) {
           flag [1] = true;
           while (flag [0]) {
                   if (turn == 0) {
                         flag [1] = false;
                         while (turn == 0) /* do nothing */;
                         flag [1] = true;
                   }
           }
           /* critical section */;
           turn = 0;
           flag [1] = false;
           /* remainder */;
      }
}
void main ()
{
      flag [0] = false;
      flag [1] = false;
      turn = 1;
      parbegin  (P0, P1);
}
```

Mutual Exclusion: software approaches

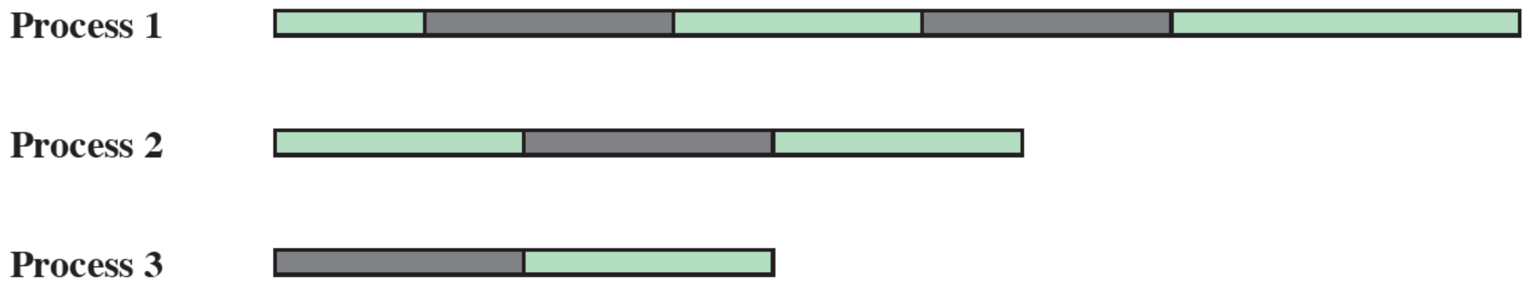
- **Peterson’ s Algorithm for Two Processes**

```
boolean flag [2];
int turn;
void P0()
{
      while (true) {
              flag [0] = true;
              turn = 1;
              while (flag [1] && turn == 1) /* do nothing */;
              /* critical section */;
              flag [0] = false;
              /* remainder */;
      }
}
 void P1()
{
      while (true) {
              flag [1] = true;
              turn = 0;
              while (flag [0] && turn == 0) /* do nothing */;
              /* critical section */;
              flag [1] = false;
              /* remainder */
      }
}
void main()
{
      flag [0] = false;
      flag [1] = false;
      parbegin  (P0, P1);
}
```

Examples of concurrent processing



(a) Interleaving (multiprogramming, one processor)



(b) Interleaving and overlapping (multiprocessing; two processors)

Principles of Concurrency

- **Interleaving and overlapping**
  - can be viewed as examples of concurrent processing
  - both present the same problems

- **Uniprocessor – the relative speed of execution of processes cannot be predicted**
  - depends on activities of other processes
  - the way the OS handles interrupts
  - scheduling policies of the OS

Difficulties of Concurrency

- **Sharing of global resources**
  - global variables, read/write
- **Difficult for the OS to manage the allocation of resources optimally**
  - may lead to a deadlock condition
- **Difficult to locate programming errors**
  - results are not deterministic and reproducible

A simple example

```
void echo()
{
        chin = getchar();
        chout = chin;
        putchar(chout);
}
```

shared global variable

An interrupt can stop instruction execution anywhere in a process

How to control access to the shared resource?

Race Condition

- **Occurs when multiple processes or threads read and write data items**
- **A example**
  - P1 and P2, share the global variable a
    - P1 updates a to the value 1
    - P2 updates a to the value 2
  - The final result depends on the order of execution
    - the “loser” of the race is the process that updates last and will determine the final value of the variable

Operating System Concerns

- **What design and management issues are raised by the existence of concurrency?**
- **The OS must**
  - be able to keep track of various processes
  - allocate and de-allocate resources for each active process
    - Processor time, Memory, Files, I/O devices
  - protect the data and physical resources of each process against interference by other processes
  - ensure that the processes and outputs are independent of the processing speed

How to understand?

Process Interaction

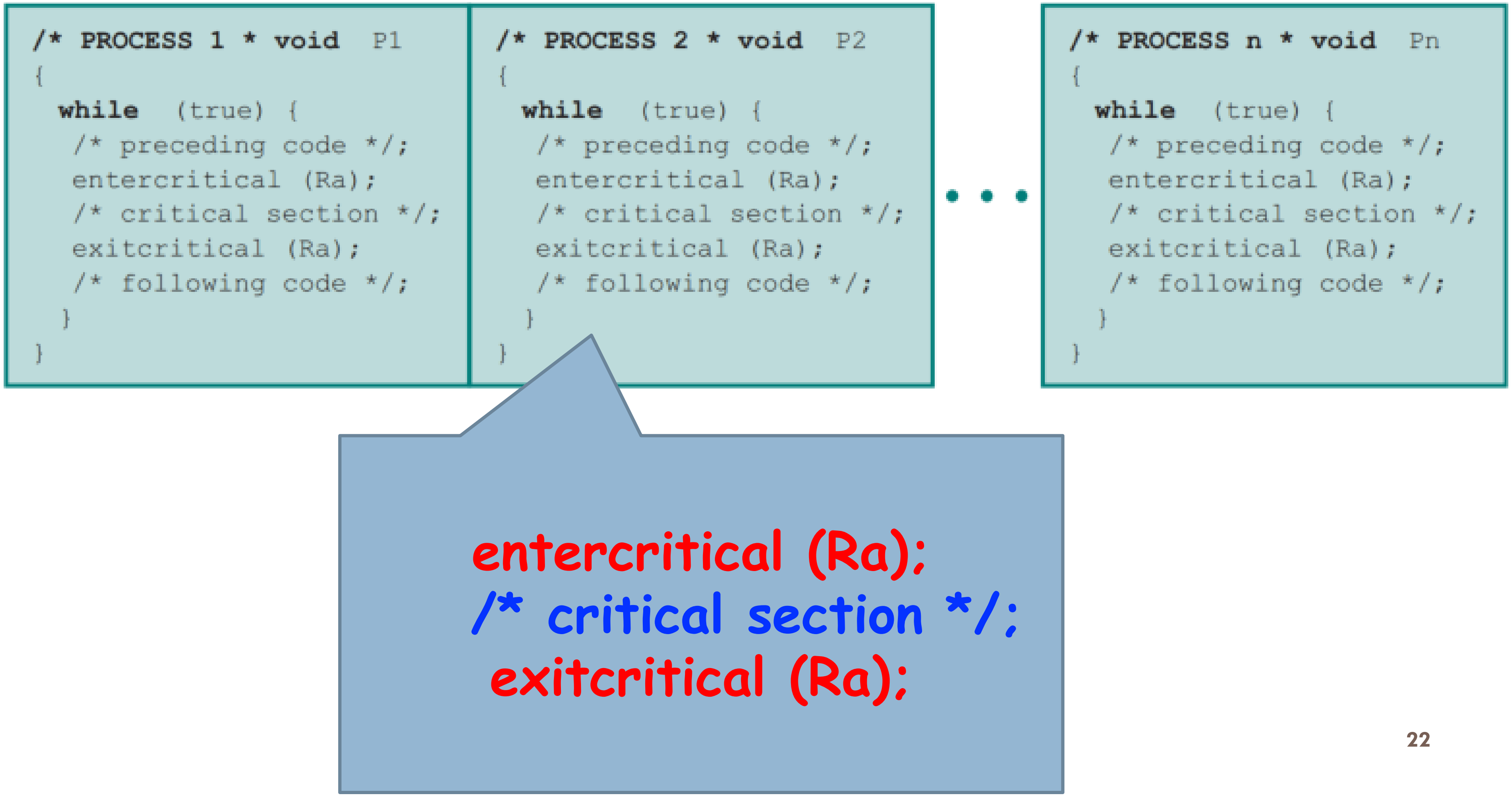
Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - processor time
  - memory
  - I/O devices
- In the case of competing processes three control problems must be faced
  - the need for mutual exclusion
  - deadlock
  - starvation

Mutual Exclusion

Illustration of Mutual Exclusion



```
/* PROCESS 1 * void   P1
{
  while   (true) {
    /* preceding code */;
    entercritical  (Ra);
    /* critical section */;
    exitcritical  (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 * void   P2
{
  while   (true) {
    /* preceding code */;
    entercritical  (Ra);
    /* critical section */;
    exitcritical  (Ra);
    /* following code */;
  }
}
```

• • •

```
/* PROCESS n * void   Pn
{
  while   (true) {
    /* preceding code */;
    entercritical  (Ra);
    /* critical section */;
    exitcritical  (Ra);
    /* following code */;
  }
}
```

```
entercritical (Ra);
/* critical section */;
exitcritical (Ra);
```

Requirements for Mutual Exclusion

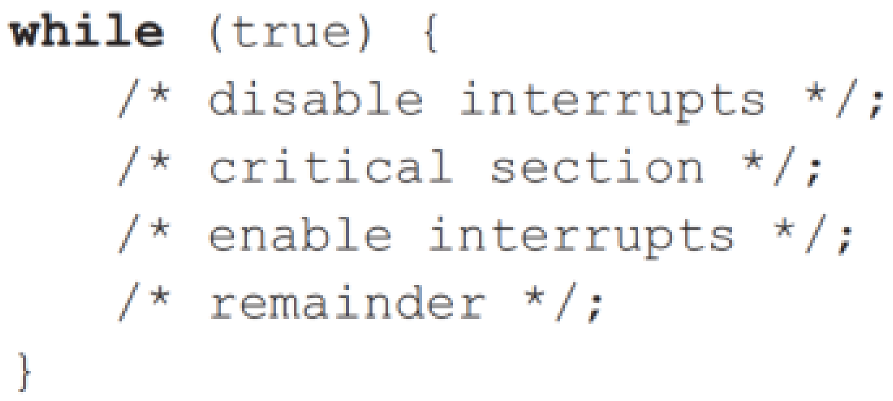
- **Mutual exclusion must be enforced**
  - Only one process at a time is allowed into its critical section
- **A process that halts must do so without interfering with other processes**
- **No deadlock or starvation**
- **A process must not be denied access to a critical section when there is no other process using it**
- **No assumptions are made about relative process speeds or number of processes**
- **A process remains inside its critical section for a finite time only**

What is the solution ?

OS or a program language
Software
Hardware

Hardware Support

- **Interrupt Disabling**
  - uniprocessor system
  - disabling interrupts guarantees mutual exclusion



```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

- **Disadvantages**
  - the efficiency of execution could be noticeably degraded
    - because the processor is limited in its ability to interleave processes
  - this approach will not work in a multiprocessor architecture

Special Machine Instructions

- **Compare&Swap Instruction**
  - also called a compare and exchange instruction, can be defined as follows [HERL90]

```
int compare_and_swap (int *word, int testval, int newval)
{
    int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
}
```

Special Machine Instructions

- **Exchange Instruction**
  - exchanges the contents of a register with that of a memory location.

```
void exchange (int *register, int *memory)
{
    int temp;
    temp = *memory;
    *memory = *register;
    *register = temp;
}
```

Special Machine Instructions

```
/* program  mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int  i)
{
    while (true)  {
      while (compare_and_swap(bolt, 0, 1) == 1)
           /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */;
    }
}
void main() {
    bolt = 0;
    parbegin (P(1), P(2), ... ,P(n));
}
```

(a) Compare and swap instruction

```
/* program mutualexclusion */
int const n = /* number of processes */;
int bolt;
void P(int  i)
{
    while (true)
        int  keyi = 1;
        do  exchange (&keyi, &bolt)
        while  (keyi != 0);
        /* critical section */;
        bolt = 0;
        /* remainder */;
    }
}
void main() {
    bolt = 0;
    parbegin  (P(1), P(2), ..., P(n));
}
```

(b) Exchange instruction

Hardware Support for Mutual Exclusion

Special Machine Instructions

- **Advantages**
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - Simple and easy to verify
  - It can be used to support multiple critical sections
    - each critical section can be defined by its own variable

- **Disadvantages**
  - Busy-waiting is employed
    - thus while a process is waiting for access to a critical section it continues to consume processor time
  - Starvation is possible
    - when a process leaves a critical section and more than one process is waiting
  - Deadlock is possible

Common Concurrency Mechanisms

Semaphore	An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a **counting semaphore** or a **general semaphore**.
Binary semaphore	A semaphore that takes on only the values 0 and 1.
Mutex	Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to 0) must be the one to unlock it (sets the value to 1).
Condition variable	A data type that is used to block a process or thread until a particular condition is true.
Monitor	A programming language construct that encapsulates variables, access procedures, and initialization code within an abstract data type. The monitor's variable may only be accessed via its access procedures and only one process may be actively accessing the monitor at any one time. The access procedures are critical sections. A monitor may have a queue of processes that are waiting to access it.
Event flags	A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
Mailboxes/messages	A means for two processes to exchange information and that may be used for synchronization.
Spinlocks	Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability

Semaphore

- **The first major advance in dealing with the problems of concurrent processes came in 1965 with Dijkstra's treatise**

The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals

Edsger Wybe Dijkstra
Netherlands – 1972

Semaphore

- May be initialized to a nonnegative integer value
- The semWait operation decrements the value
- The semSignal operation increments the value

A variable that has an integer value upon which only three operations are defined:

There is no way to inspect or manipulate semaphores other than these three operations

Consequences

- **[DOWN16] points out three interesting consequences of the semaphore definition**
  - There is no way to know before a process decrements a semaphore whether it will block or not
  - There is no way to know which process, if either, will continue immediately on a uniprocessor system
  - You don’ t necessarily know whether another process is waiting, so the number of unblocked processes may be zero or one

A Definition of Semaphore Primitives

```
struct semaphore {
int count;
queueType queue;
};
void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0) {
          /* place this process in s.queue */;
          /* block this process */;
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count<= 0) {
          /* remove a process P from s.queue */;
          /* place process P on ready list */;
     }
}
```

Binary Semaphore Primitives

```
struct binary_semaphore {
enum {zero, one} value;
queueType queue;
};
void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Strong/Weak Semaphores

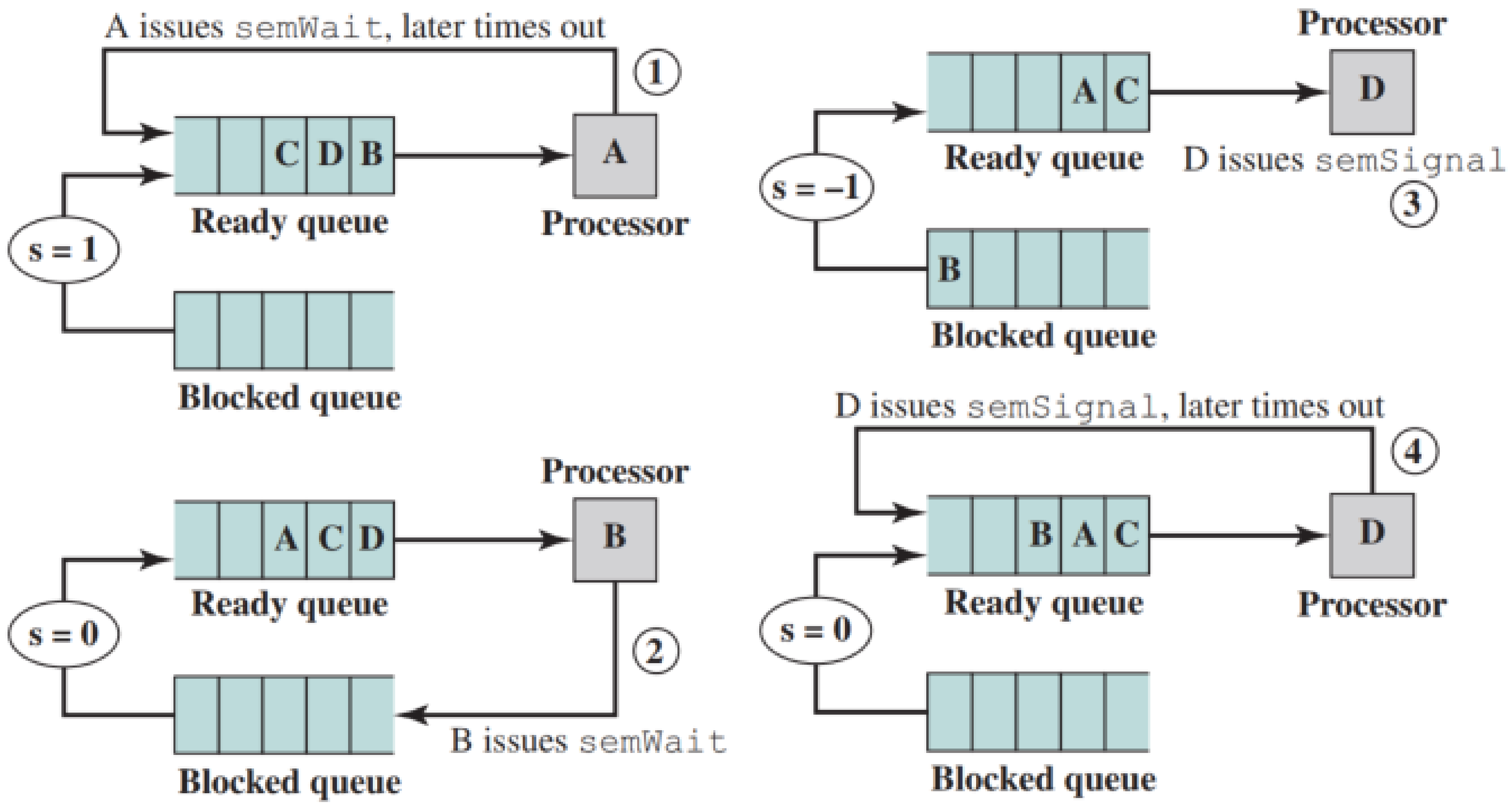
- **A queue is used to hold processes waiting on the semaphore**

Strong Semaphores

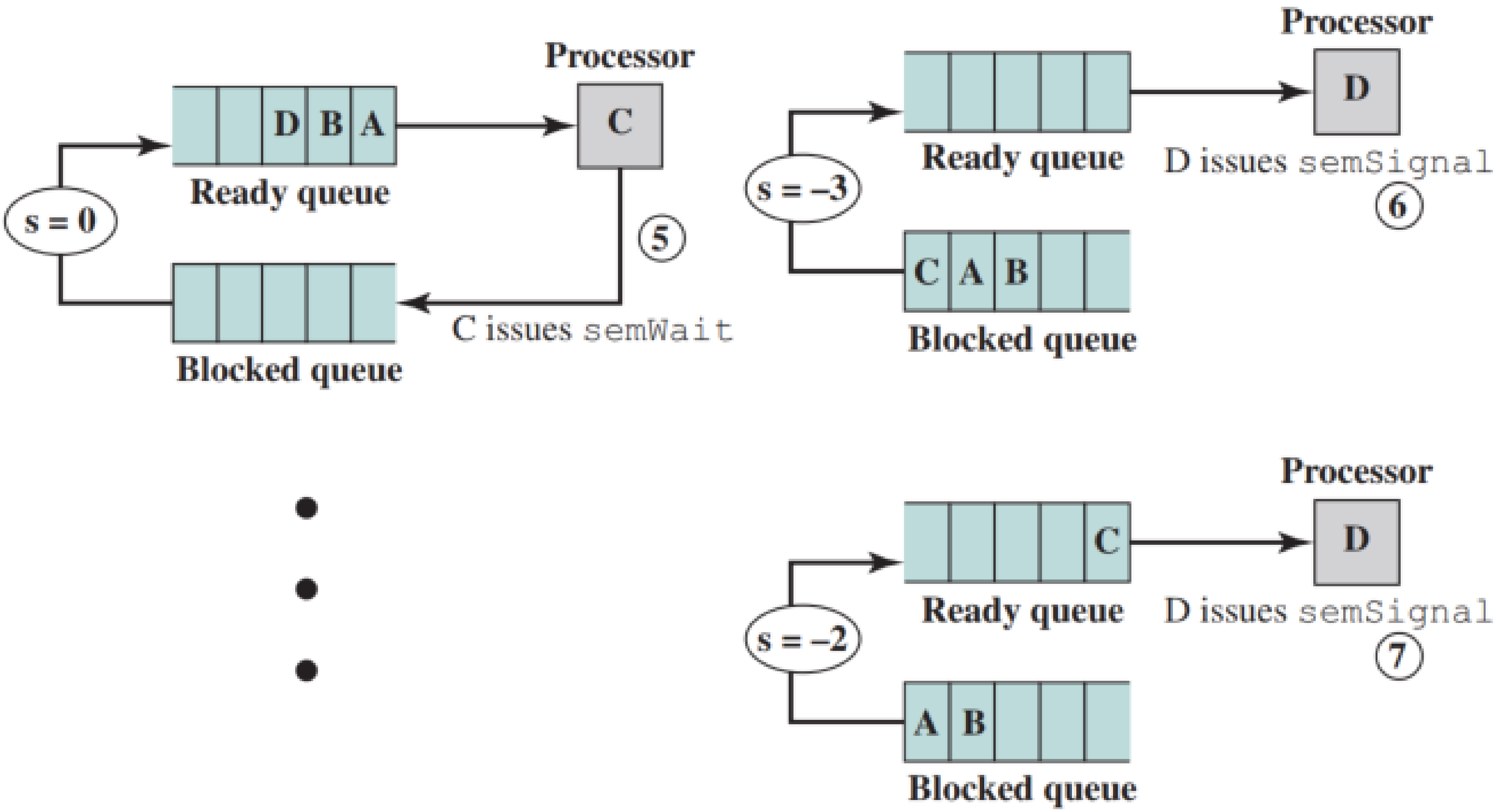
- the process that has been blocked the longest is released from the queue first (FIFO)

Weak Semaphores

- the order in which processes are removed from the queue is not specified



Example of Semaphore Mechanism

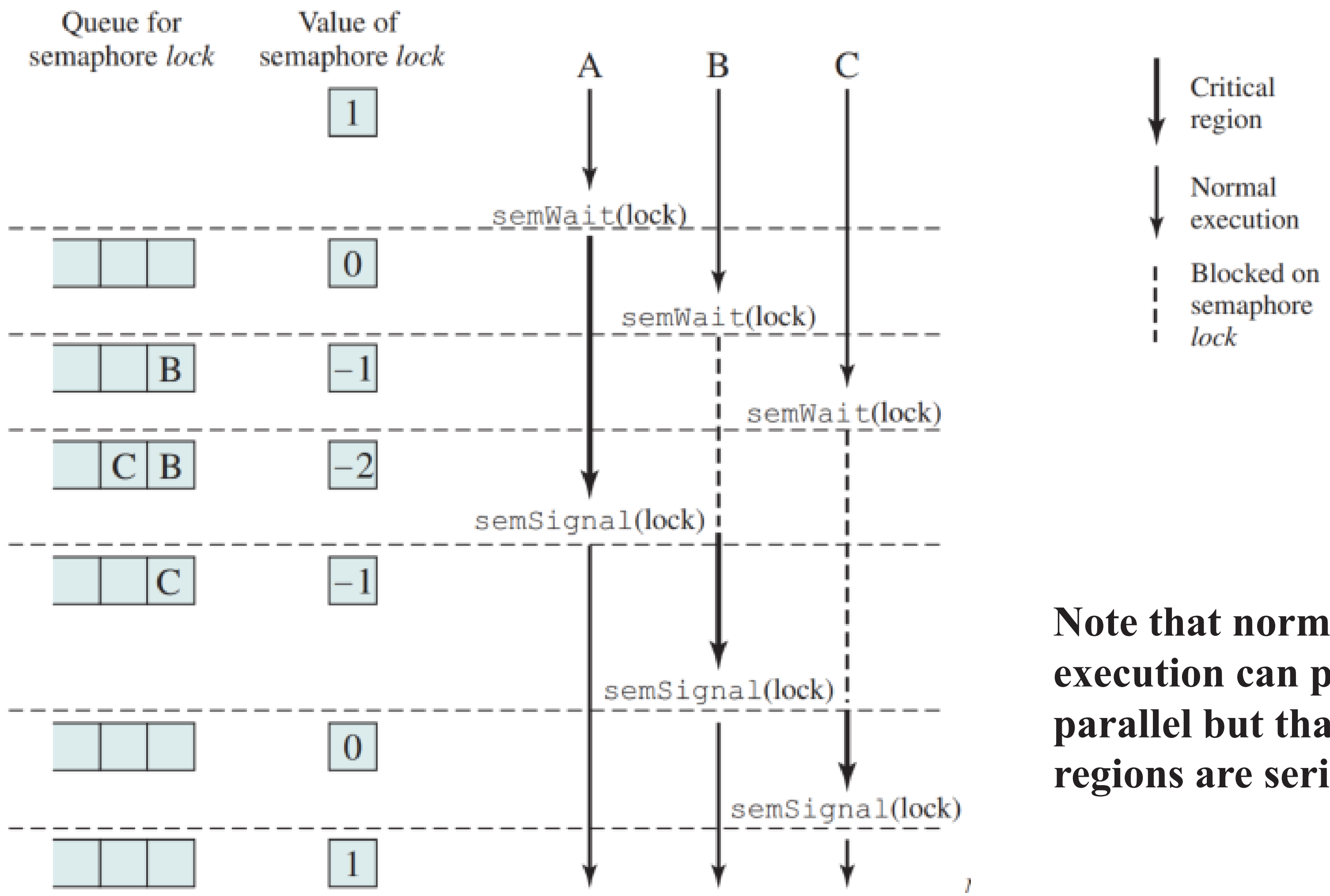


Example of Semaphore Mechanism

Mutual Exclusion Using Semaphores

```
/* program mutualexclusion */
const int n = /* number of processes */;
semaphore s = 1;
void P(int i)
{
        while (true) {
                semWait(s);
                /* critical section */;
                semSignal(s);
                /* remainder */;
        }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```

Shared Data Protected by a Semaphore

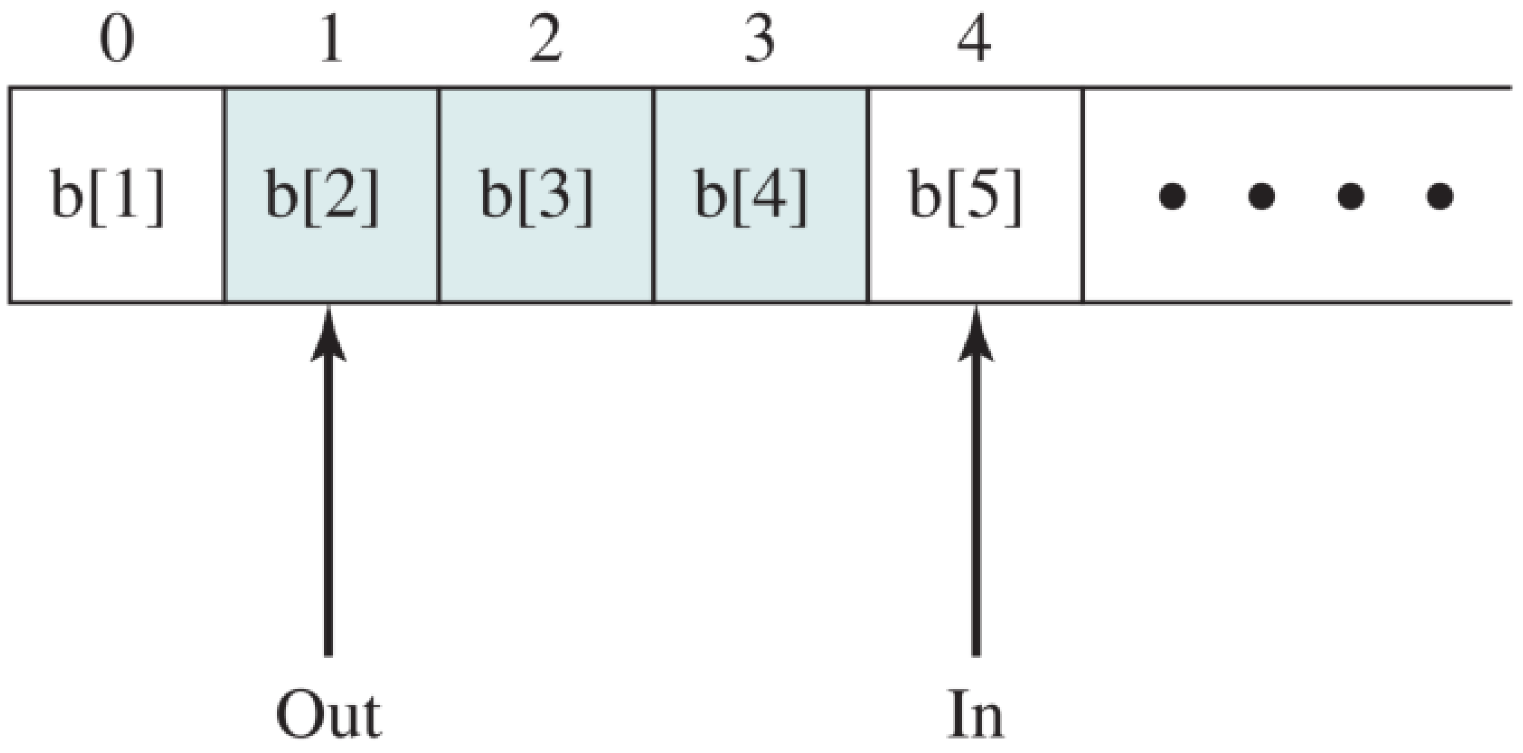


Note that normal execution can proceed in parallel but that critical regions are serialized.

Producer/Consumer Problem

- **General Situation:**
  - one or more producers are generating data and placing these in a buffer
  - a single consumer is taking items out of the buffer one at time
  - only one producer or consumer may access the buffer at any one time
- **The problem**
  - ensure that the producer can' t add data into full buffer
  - ensure the consumer can' t remove data from an empty buffer

Infinite Buffer



Note: Shaded area indicates portion of buffer that is occupied.

An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/*  program producerconsumer */
    int n;
    binary_semaphore s = 1, delay = 0;
    void producer()
    {
         while (true) {
              produce();
              semWaitB(s);
              append();
              n++;
              if (n==1) semSignalB(delay);
              semSignalB(s);
         }
    }
    void consumer()
    {
         semWaitB(delay);
         while (true) {
              semWaitB(s);
              take();
              n--;
              semSignalB(s);
              consume();
              if (n==0) semWaitB(delay);
         }
    }
    void main()
    {
         n = 0;
         parbegin (producer, consumer);
    }
```

What's meaning?

Possible Scenario for the Program above

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	if (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	if (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		if (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		if (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	- 1	0
21		semSignalB(s)	1	- 1	0

A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

```
/* program producerconsumer */
      int n;
      binary_semaphore s = 1, delay = 0;
      void producer()
      {
            while (true) {
                  produce();
                  semWaitB(s);
                  append();
                  n++;
                  if (n==1) semSignalB(delay);
                  semSignalB(s);
            }
      }
      void  consumer()
      {
            int  m; /* a local variable */
            semWaitB(delay);
            while  (true)  {
                  semWaitB(s);
                  take();
                  n--;
                  m = n;
                  semSignalB(s);
                  consume();
                  if  (m==0)  semWaitB(delay);
            }
      }
      void  main()
      {
            n = 0;
            parbegin  (producer, consumer);
      }
```

A Solution to the Infinite-Buffer Producer/Consumer Problem Using Semaphores

```
/* program producerconsumer */
semaphore n = 0, s = 1;
void producer()
{
    while (1) {
        produce();
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (1) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

Which one produce a serious flaw?

A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1, n = 0, e = sizeofbuffer;
void producer()
{
    while (true) {
        produce();
        semWait(e);
        semWait(s);
        append();
        semSignal(s);
        semSignal(n);
    }
}
void consumer()
{
    while (true) {
        semWait(n);
        semWait(s);
        take();
        semSignal(s);
        semSignal(e);
        consume();
    }
}
void main()
{
    parbegin (producer, consumer);
}
```

What's meaning?

An Example of Semaphore Usage

例： 假如系统中有**2**台打印机可用；

现有**4**个进程**P1，P2，P3，P4**都在不同时间里以不同数量申请该设备。

S初值＝**2**，表示共有**2**台打印机可用。

P1、**P2**、**P3**、**P4**为并发进程，本例假设第**1**个被调度的为**P2**。

列出使用**P**、**V**操作使这**4**个进程互斥工作过程。

并发进程Pi工作序列 ( )数字表示运行顺序				并发执行顺序	当前处于运行态进程	所执行的操作(P/V)	信号量S的值(初值=2)	被唤醒的进程	信号量S的等待队列
P1	P2	P3	P4						
⋮	⋮	⋮	⋮	(1)					
(2) P(S)	(1) P(S)	(3) P(S)	(5) P(S)	(2)					
· 打印 ·	· 打印 ·	· 打印 ·	· 打印 ·	(3)					
(6) V(S)	(4) P(S)	(7) V(S)	(10) V(S)	(4)					
⋮	· 打印 ·	⋮	⋮	(5)					
	(9) V(S)	(8) P(S)		(6)					
	· 打印 ·	· 打印 ·		(7)					
	(11) V(S)	(12) V(S)		(8)					
	⋮	⋮		(9)					
				(10)					
				(11)					
				(12)					

Implementation of Semaphores

- It is imperative that the semWait and semSignal operations be implemented as atomic primitives
- Can be implemented in hardware or firmware
- Software schemes such as Dekker’s or Peterson’s algorithms can be used
- Use one of the hardware-supported schemes for mutual exclusion

Disadvantages of Semaphores

- **Semaphores provide a primitive yet powerful and flexible tool**
  - for enforcing mutual exclusion
  - for coordinating processes
- **Difficult to produce a correct program using semaphores**
  - semWait and semSignal operations may be scattered throughout a program
  - not easy to see the overall effect of these operations on the semaphores they affect

```
/*   program producerconsumer */
     int n;
     binary_semaphore s = 1, delay = 0;
     void producer()
     {
          while (true) {
               produce();
               semWaitB(s);
               append();
               n++;
               if (n==1) semSignalB(delay);
               semSignalB(s);
          }
     }
     void consumer()
     {
          semWaitB(delay);
          while (true) {
               semWaitB(s);
               take();
               n--;
               semSignalB(s);
               consume();
               if (n==0) semWaitB(delay);
          }
     }
     void main()
     {
          n = 0;
          parbegin (producer, consumer);
     }
```

Monitors

- **The monitor is a programming-language construct**
  - that provides equivalent functionality to that of semaphores
- **First formally defined by Hoare in 1974**

For his fundamental contributions to the definition and design of programming languages

C. Antony ("Tony") R. Hoare
United Kingdom – 1980

- **It is easier to control**
  - Implemented in many programming languages
    - including Concurrent Pascal, Modula-2, Modula-3, and Java
  - Implemented as a program library
- **It is a software module**
  - one or more procedures
  - an initialization sequence
  - local data

Monitor Characteristics

Local data variables are accessible **only** by the monitor's procedures and **not** by any **external** procedure

Process enters monitor by **invoking one of its procedures**

Only one process may be executing in the monitor **at a time**

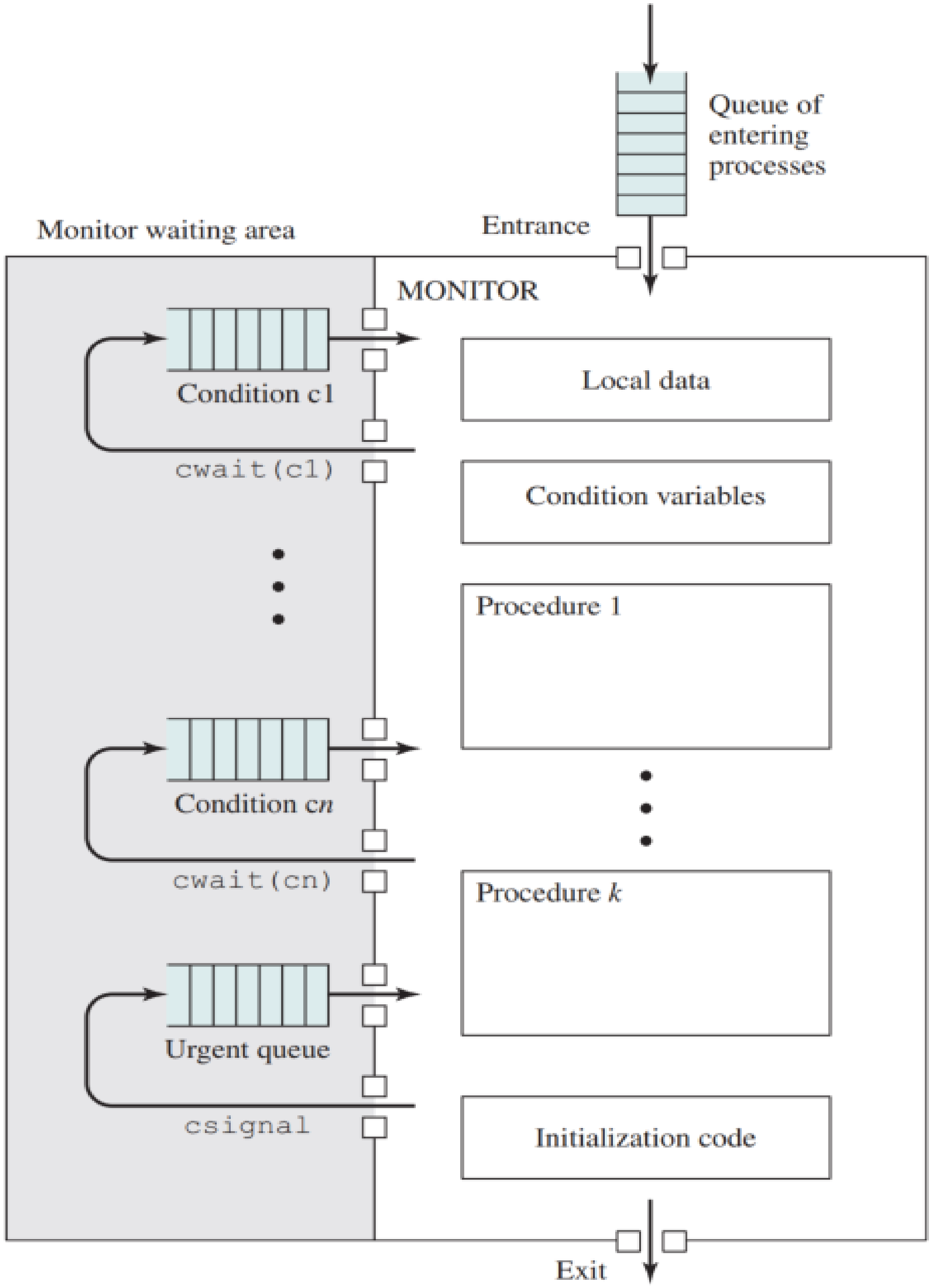
Synchronization

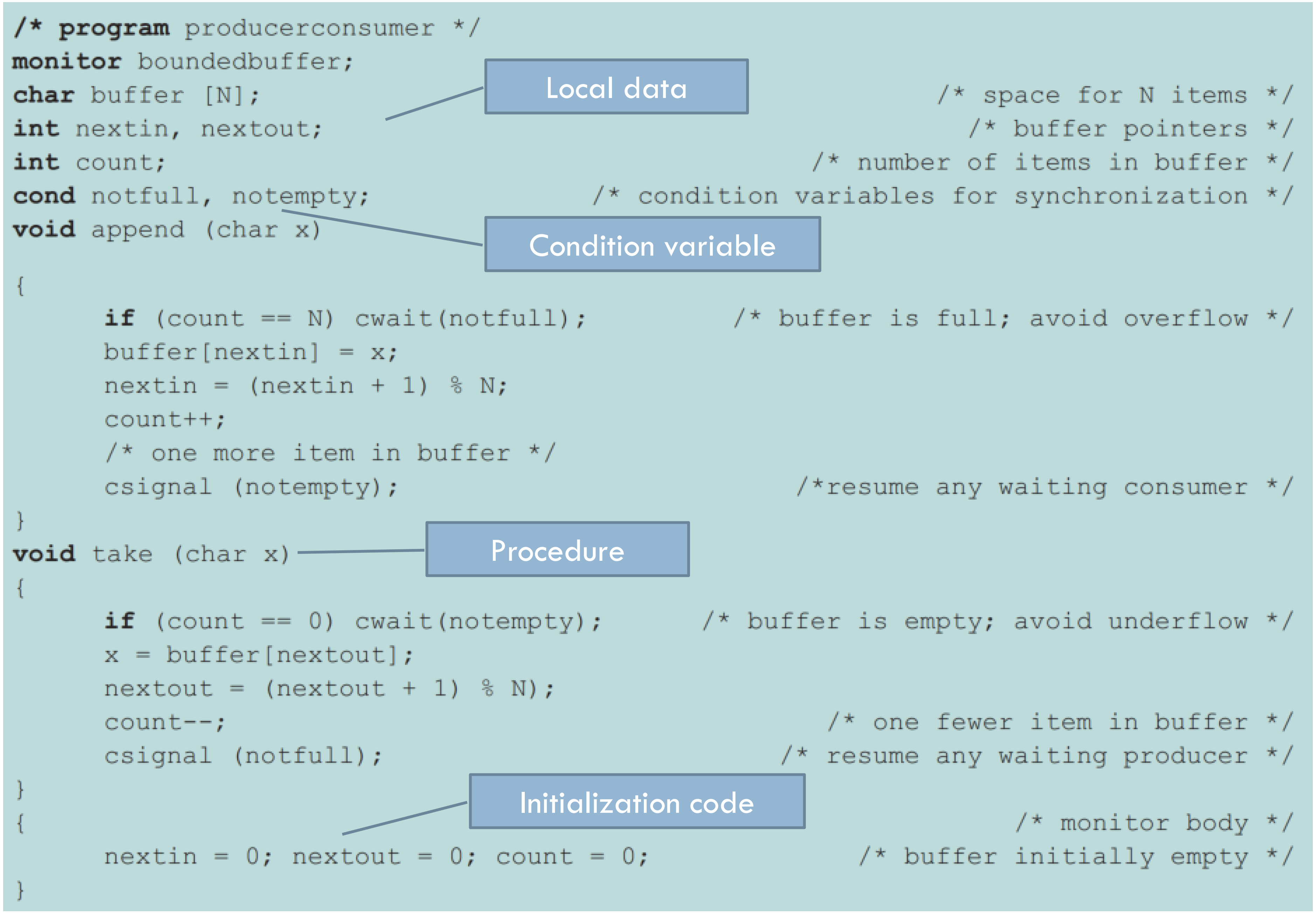
- Achieved by the use of condition variables that are contained within the monitor and accessible only within the monitor

What is the difference from those for the semaphore?

- Condition variables are operated on by two functions
  - cwait(c)
    - suspend execution of the calling process on condition c
  - csignal(c)
    - resume execution of some process blocked after a cwait on the same condition
    - if there is no such process, do nothing

Structure of a Monitor

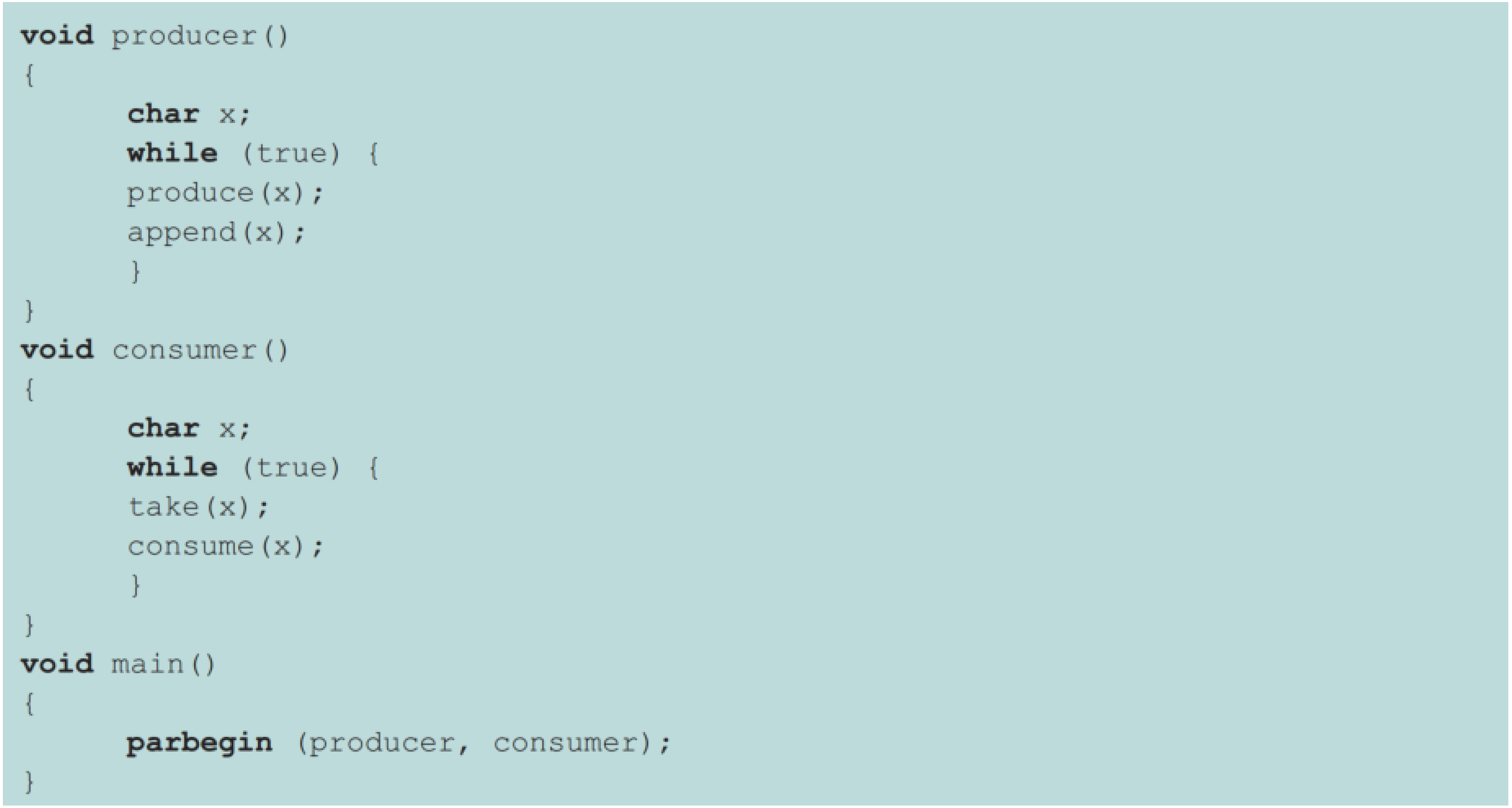




```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                           /* space for N items */
int nextin, nextout;                                         /* buffer pointers */
int count;                                         /* number of items in buffer */
cond notfull, notempty;                /* condition variables for synchronization */
void append (char x)

{
      if (count == N) cwait(notfull);            /* buffer is full; avoid overflow */
      buffer[nextin] = x;
      nextin = (nextin + 1) % N;
      count++;
      /* one more item in buffer */
      csignal (notempty);                           /*resume any waiting consumer */
}
void take (char x)
{
      if (count == 0) cwait(notempty);        /* buffer is empty; avoid underflow */
      x = buffer[nextout];
      nextout = (nextout + 1) % N);
      count--;                                        /* one fewer item in buffer */
      csignal (notfull);                             /* resume any waiting producer */
}
{                                                                /* monitor body */
      nextin = 0; nextout = 0; count = 0;                  /* buffer initially empty */
}
```

A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor



```
void producer()
{
        char x;
        while (true) {
        produce(x);
        append(x);
        }
}
void consumer()
{
        char x;
        while (true) {
        take(x);
        consume(x);
        }
}
void main()
{
        parbegin (producer, consumer);
}
```

A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

Three advantages over Semaphore

- all of the synchronization functions are confined to the monitor
- it is easier to verify that the synchronization has been done correctly and to detect bugs
- once a monitor is correctly programmed, access to the protected resource is correct for access from all processes

Monitors with Notify and Broadcast

- **Two drawbacks to Hoare's approach**
  - If the process issuing the csignal has not finished with the monitor, then two additional process switches are required
    - one to block this process
    - another to resume it when the monitor becomes available
  - Process scheduling associated with a signal must be perfectly reliable

Butler W Lampson
United States – 1992

Message Passing

- **When processes interact with one another two fundamental requirements must be satisfied**

synchronization	communication
• to enforce mutual exclusion	• to exchange information

- **Message Passing is one approach to providing both of these functions**
  - works with distributed systems and shared memory multiprocessor and uniprocessor systems

Message Passing

- The actual function is normally provided in the form of a pair of primitives
  - send (destination, message)
  - receive (source, message)

- A process sends information in the form of a message to another process designated by a destination
- A process receives information by executing the receive primitive, indicating the source and the message

Design Characteristics of Message Systems

- **Synchronization**
  - Send
    - blocking
    - nonblocking
  - Receive
    - blocking
    - nonblocking
    - test for arrival
- **Format**
  - Content
    - Length
    - fixed
    - variable
- **Addressing**
  - Direct
    - send
    - receive
      - explicit
      - implicit
  - Indirect
    - static
    - dynamic
    - ownership
- **Queueing Discipline**
  - FIFO
  - Priority

Synchronization

The communication of a message between two processes implies some level of synchronization between the two

When a receive primitive is executed in a process, there are two possibilities

If a message has previously been sent, the message is received and execution continues.

The receiver cannot receive a message until it has been sent by another process

If there is no waiting message, then either (a) the process is blocked until a message arrives, or (b) the process continues to execute, abandoning the attemptto receive.

Blocking Send, Blocking Receive

- **Both sender and receiver are blocked until the message is delivered**
  - Sometimes referred to as a **rendezvous**
  - Allows for **tight synchronization** between processes

Nonblocking Send

Nonblocking send, blocking receive

- sender continues on but receiver is blocked until the requested message arrives
- **most useful combination**
- sends one or more messages to a variety of destinations as quickly as possible
- example -- a service process that exists to provide a service or resource to other processes

Nonblocking send, nonblocking receive

- neither party is required to wait

Direct Addressing

- Send primitive includes a specific identifier of the destination process
- Receive primitive can be handled in one of two ways:
  - require that the process explicitly designate a sending process
    - effective for cooperating concurrent processes
  - implicit addressing
    - source parameter of the receive primitive possesses a value returned when the receive operation has been performed

Indirect Addressing

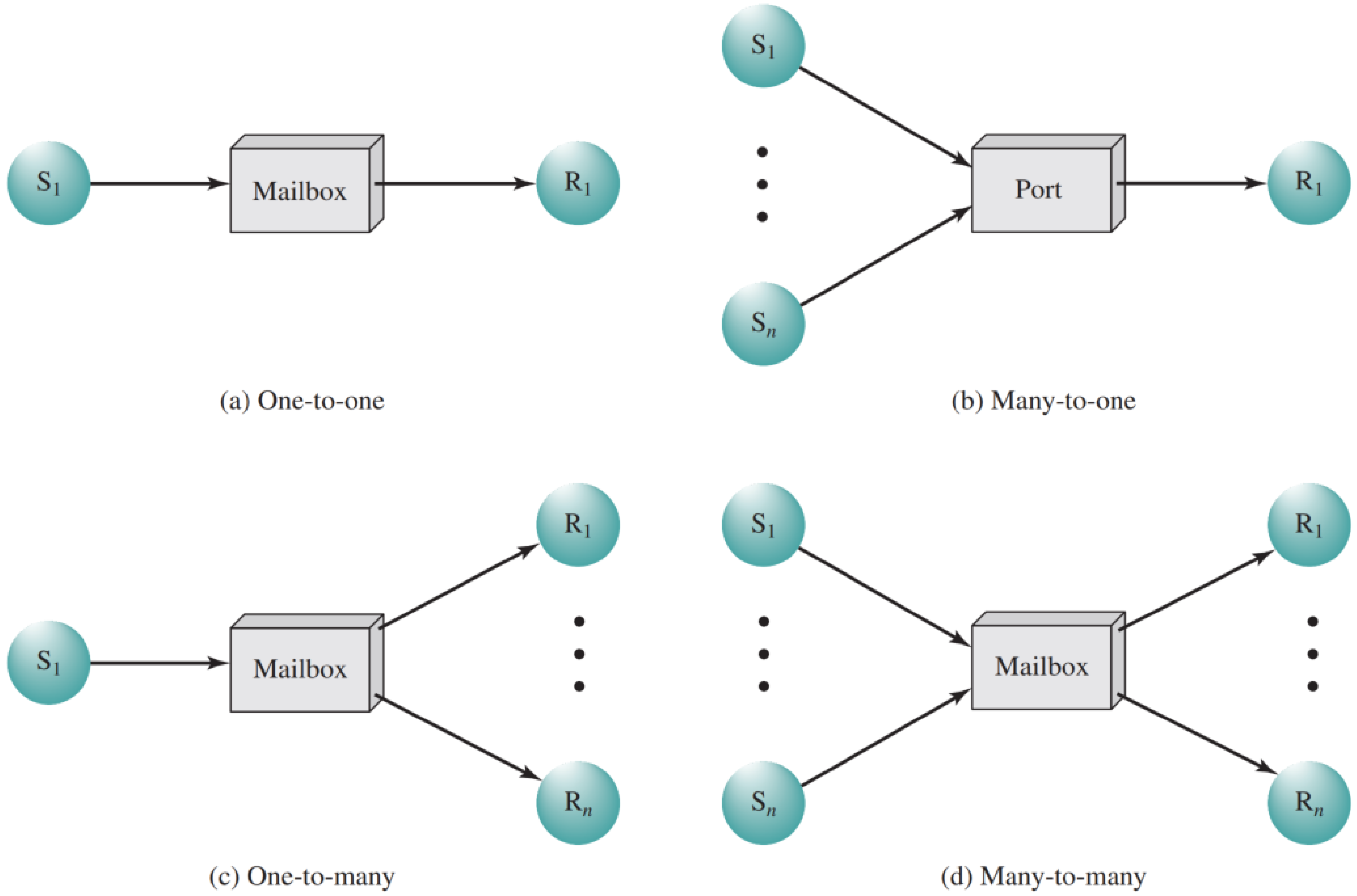
Messages are sent to a shared data structure consisting of queues that can temporarily hold messages

Queues are referred to as **mailboxes**

One process sends a message to the mailbox and the other process picks up the message from the mailbox

Allows for greater flexibility in the use of messages

Indirect Process Communication



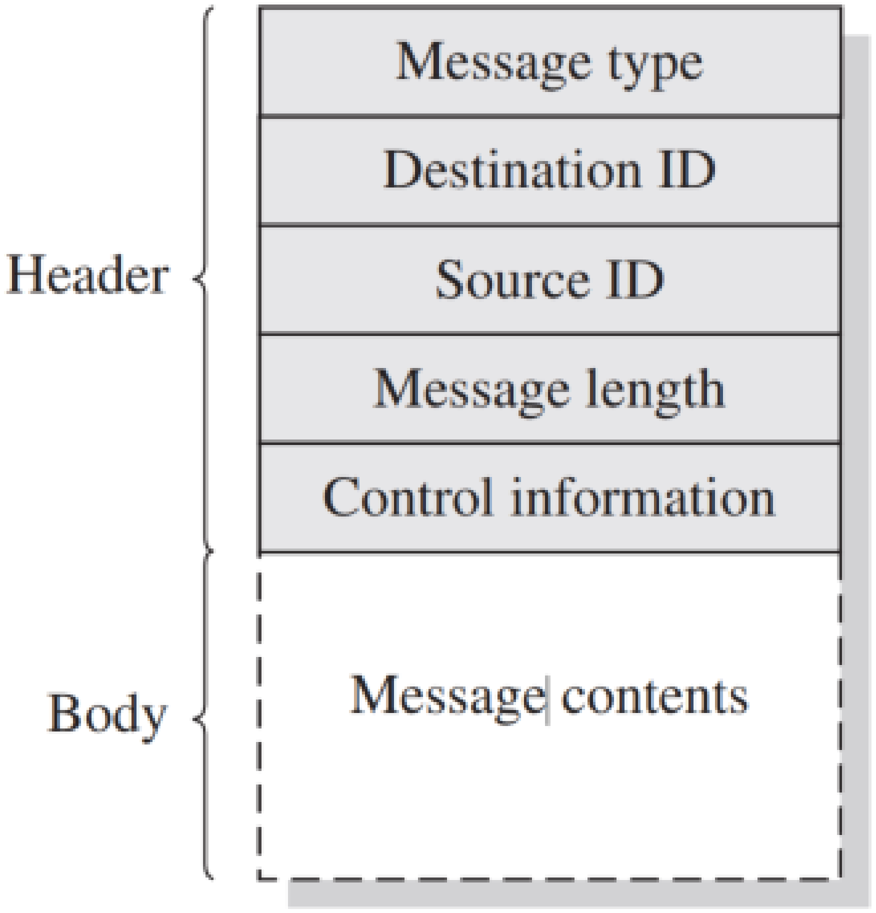
(a) One-to-one

(b) Many-to-one

(c) One-to-many

(d) Many-to-many

General Message Format



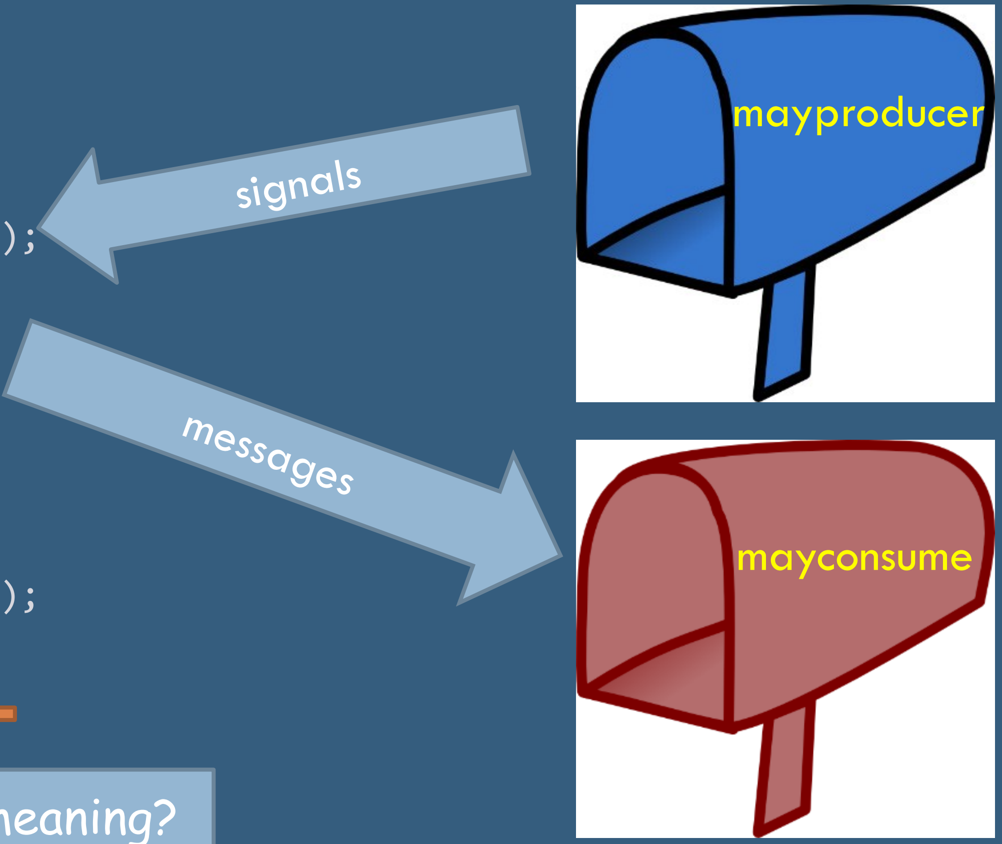
Mutual Exclusion Using Messages

```
/* program mutualexclusion */
const int n = /* number of process */
void P(int i)
{
    message msg;
    while (true) {
        receive (box, msg);
        /* critical section */;
        send (box, msg);
        /* remainder */;
    }
}
void main()
{
    create mailbox (box);
    send (box, null);
    parbegin (P(1), P(2), . . . , P(n));
}
```

A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Messages

```
const int capacity = /* buffering capacity */ ;
null = /* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true) {
        receive (mayproduce,pmsg);
        pmsg = produce();
        send (mayconsume,pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true) {
        receive (mayconsume,cmsg);
        consume (cmsg);
        send (mayproduce,null);
    }
}
void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1;i<= capacity;i++) send (mayproduce,null);
    parbegin (producer,consumer);
}
```

What's meaning?



Readers/Writers Problem

- **A data area is shared among many processes**
  - some processes only read the data area (readers)
  - some only write to the data area (writers)

- **Conditions that must be satisfied:**
  - any number of readers may simultaneously read the file
  - only one writer at a time may write to the file
  - if a writer is writing to the file, no reader may read it

Readers/Writers Problem

A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

```
/* program readersandwriters */
    int readcount;
    semaphore x = 1, wsem = 1;
    void reader()
    {
        while (true){
            semWait(x);
            readcount++;
            if (readcount == 1) semWait(wsem);
            semSignal(x);
            READUNIT();
            semWait(x);
            readcount--;
            if (readcount == 0) semSignal(wsem);
            semSignal(x);
        }
    }
    void writer()
    {
        while (true){
            semWait(wsem);
            WRITEUNIT();
            semSignal(wsem);
        }
    }
    void main()
    {
        readcount = 0;
        parbegin(reader, writer);
    }
```

A Solution to the Readers/Writers Problem Using Semaphores: Writer Have Priority

```
 int readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true){
        semWait(z);
        semWait(rsem);
        semWait(x);
        readcount++;
        if (readcount == 1) semWait(wsem);
        semSignal(x);
        semSignal(rsem);
        semSignal(z);
        READUNIT();
        semWait(x);
        readcount--;
        if (readcount == 0) semSignal(wsem);
        semSignal(x);
    }
}
void writer()
{
    while (true){
        semWait(y);
        writecount++;
        if (writecount == 1) semWait(rsem);
        semSignal(y);
        semWait(wsem);
        WRITEUNIT();
        semSignal(wsem);
        semWait(y);
        writecount--;
        if (writecount == 0) semSignal(rsem);
        semSignal(y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin(reader, writer);
}
```

State of the Process Queues

Readers only in the system	•wsem set •no queues
Writers only in the system	•wsem and rsem set •writers queue on wsem
Both readers and writers with read first	•wsem set by reader •rsem set by writer •all writers queue on wsem •one reader queues on rsem •other readers queue on z
Both readers and writers with write first	•wsem set by writer •rsem set by writer •writers queue on wsem •one reader queues on rsem •other readers queue on z

A Solution to the Readers/Writers Problem Using Message Passing

```
void reader(int i)
{
    message rmsg;
        while (true) {
                rmsg = i;
            send (readrequest, rmsg);
            receive (mbox[i], rmsg);
            READUNIT ();
            rmsg = i;
            send (finished, rmsg);
        }
}
void writer(int j)
{
    message rmsg;
    while (true){
        rmsg = j;
        send (writerequest, rmsg);
        receive (mbox[j], rmsg);
        WRITEUNIT ();
        rmsg = j;
        send (finished, rmsg);
    }
}
```

```
void controller()
{
    while (true)
    {
        if (count > 0) {
            if (!empty (finished)) {
                receive (finished, msg);
                count++;
            }
            else if (!empty (writerequest)) {
                receive (writerequest, msg);
                writer_id = msg.id;
                count = count – 100;
            }
            else if (!empty (readrequest)) {
                receive (readrequest, msg);
                count--;
                send (msg.id, "OK");
            }
        }
        if (count == 0) {
            send (writer_id, "OK");
            receive (finished, msg);
            count = 100;
        }
        while (count < 0) {
            receive (finished, msg);
            count++;
        }
    }
}
```

Summary

- **Operating system themes**
  - Multiprogramming, multiprocessing, distributed processing
  - **Fundamental** to these themes is **concurrency**
    - issues of **conflict** resolution and **cooperation** arise
- **Mutual Exclusion**
  - Condition in which there is a set of **concurrent processes**, **only one** of which is able to access a given resource or perform a given function at any time
  - Three approaches to supporting
- **Mutual Exclusion: software approaches**
- **Hardware support**
- **OS or a programming language**
  - **Semaphores**
    - Used for **signaling** among processes and can be readily used to **enforce a mutual exclusion**
  - **discipline**
  - **Monitors**
  - **Messages**
    - Useful for the **enforcement of mutual exclusion discipline**
    - provide an effective means of **interprocess communication**