Operating Systems: Internals and Design Principles

Chapter 5 Concurrency: Mutual Exclusion and Synchronization

Seventh Edition

By William Stallings

Lecture 5: Concurrency SW & HW Mutual Exclusion

- Concurrency
- Race condition
- Critical section
- Mutual exclusion
- Mutual exclusion solution with only shared memory and SW
- Mutual exclusion with HW assistance

Operating Systems: Internals and Design Principles

" 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."

—THE COMPUTER SCIENCE AND ENGINEERING RESEARCH STUDY, MIT Press, 1980

Multiple Processes

- Operating System design is concerned with the management of processes and threads:
  - Multiprogramming
  - Multiprocessing
  - Distributed Processing

Concurrency

Arises in Three Different Contexts:

Multiple Applications

- invented to allow processing time to be shared among active applications

Structured Applications

- extension of modular design and structured programming

Operating System Structure

- OS themselves implemented as a set of processes or threads

Concurrency Key Terms

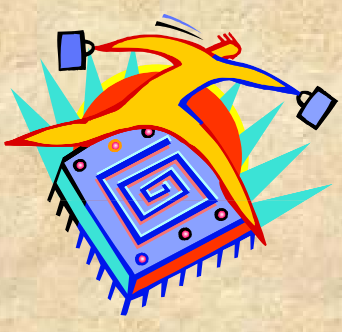
atominen operaatio	**atomic operation**	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
kriittinen vaihe kriittinen alue	**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.
lukkiutuminen	**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.
elolukkko	**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.
poissulkemis-ongelma	**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.
kilpailutilanne	**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.
nälkiintyminen	**starvation**	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Discuss

Table 5.1 Some Key Terms Related to Concurrency

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
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

Race Condition

- Occurs when multiple processes or threads read and write data items
- 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

”Loser” is really the ”winner”?

Operating System Concerns

- Design and management issues 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
    - 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

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
  - for example: I/O devices, memory, processor time, clock

In the case of competing processes three control problems must be faced:

- **the need for mutual exclusion**
- **deadlock**
- **starvation**

Mutual Exclusion

preprotocol	esiprotokolla
critical section	kriittinen vaihe
postprotocol	jälkiprotokolla

```
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 */;
   }
}
```

Figure 5.1 Illustration of Mutual Exclusion

Requirements for Mutual Exclusion

- Must be enforced
- A process that halts must do so without interfering with other processes
  - Process can halt outside critical section (CS)
  - Process can not halt in CS or CS pre/post-protocol
- 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

Mutual Exclusion: Software Solution

- Basic idea: mutex solution with just software
  - No assistance from HW (special instructions)
  - No assistance from OS (special services)
- Requires shared memory
- Indivisible (atomic) operations: memory read/write
- First solution: Dekker, 1965

SW mutex, 1st attempt

shared int turn = 0;

```
/* PROCESS 0 /*

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

```
/* PROCESS 1 */

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

Fig. A.1

- Which scenario fails?

SW mutex, 2nd attempt

shared boolean flag[0:1] = [false, false];

```
/* PROCESS 0 */

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

```
/* PROCESS 1 */

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

Fig. A.1 (b)

- Which scenario fails?

SW mutex, 3rd attempt

shared boolean flag[0:1] = [false, false];

/* 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;
•	•

Fig. A.1 (c)

- Which scenario fails?

SW mutex, 4th attempt

shared boolean flag[0:1] = [false, false];

/* 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;
•	•

Fig. A.1 (d)

- Which scenario fails?

SW mutex: Dekker (1965)

```
void main ()
{
      flag [0] = false;
      flag [1] = false;
      turn = 1;
      parbegin (P0, P1);
}
```

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          while (flag [1]) {
               if (turn == 1) {
                    flag [0] = false;
                    while (turn == 1);

                    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);

                    flag [1] = true;
               }
          }
          /* critical section   */;
          turn = 0;
          flag [1] = false;
          /* remainder   */;
     }
}
```

SW mutex Peterson (1981)

```
void main()
{
      flag [0] = false;
      flag [1] = false;
      parbegin (P0, P1);
}
```

```
boolean flag [2];
int turn;
void P0()
{
     while (true) {
          flag [0] = true;
          turn = 1;
          while (flag [1] && turn == 1);
          /* critical section  */;
          flag [0] = false;
          /* remainder  */;
     }
}
```

```
void P1()
{
     while (true) {
          flag [1] = true;
          turn = 0;
          while (flag [0] && turn == 0);
          /* critical section  */;
          flag [1] = false;
          /* remainder  */
     }
}
```

Discuss

Mutual Exclusion: Hardware Support

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

- **Disadvantages:**
  - the efficiency of execution could be noticeably degraded
  - this approach will not work in a multiprocessor architecture

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  - Compare&Swap Instruction
    - also called a "compare and exchange instruction"
    - a **compare** is made between a memory value and a test value
    - if the values are the same a **swap** occurs
    - carried out atomically

Compare and Swap Instruction

lock variable

if open

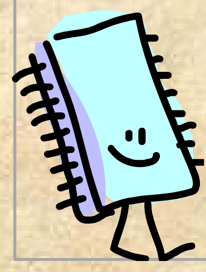
set locked

busy wait

locked originally

```
/* 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



Discuss

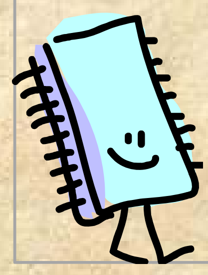
Figure 5.2 Hardware Support for Mutual Exclusion

Exchange Instruction

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

if it <u>was</u> open

(b) Exchange instruction

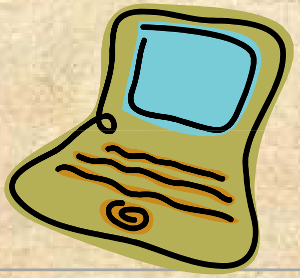


Also: test-and-set instruction

Figure 5.2 Hardware Support for Mutual Exclusion

Special Machine Instruction: 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



Special Machine Instruction: 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

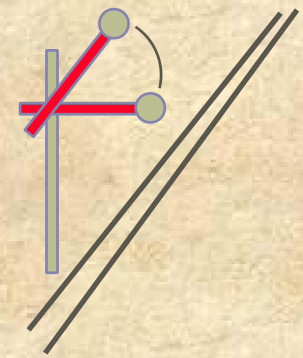
Discuss

SW & HW Mutex Summary

- Possible to solve mutex problem just with SW
  - Theorethically proven solutions
- Easier to solve with HW support
  - Interrupt disabling works with uniprocessor systems
  - Busy wait is more suitable to multicore systems
- These methods all involve
  - Busy waiting (waiting process in running state)
  - Usually suitable only for very short critical sections
    - E.g., not suitable for data base updates
- Better methods needed: wait in suspended state

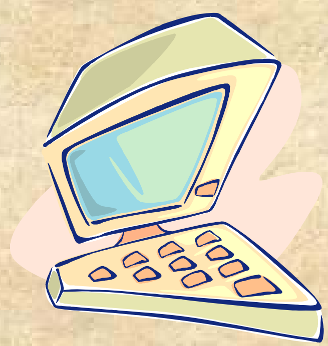
Lecture 6
Semaphores and Monitors

- Semaphores
- Mutex with semaphores
- Synchronization with semaphores
- Producer/consumer problem
- Monitors
- Mutex and synchronization with monitors



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 zero) 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

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

1) **May be initialized to a nonnegative integer value**
2) **The semWait operation decrements the value**
3) **The semSignal operation increments the value**

Consequences

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 will continue immediately on a uniprocessor system when two processes are running concurrently

You don't know whether another process is waiting so the number of unblocked processes may be zero or one

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 */;
     }
}
```

queue for strong semaphores
list for weak semaphores

wait, down, P, passeren, take

signal, up, V, vrijgeven, release

Figure 5.3 A Definition of Semaphore Primitives

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 */;
     }
}
```

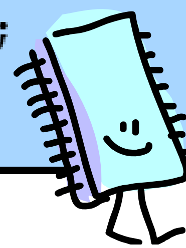


Figure 5.4 A Definition of Binary Semaphore Primitives

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

D gives something to A, B, and C.

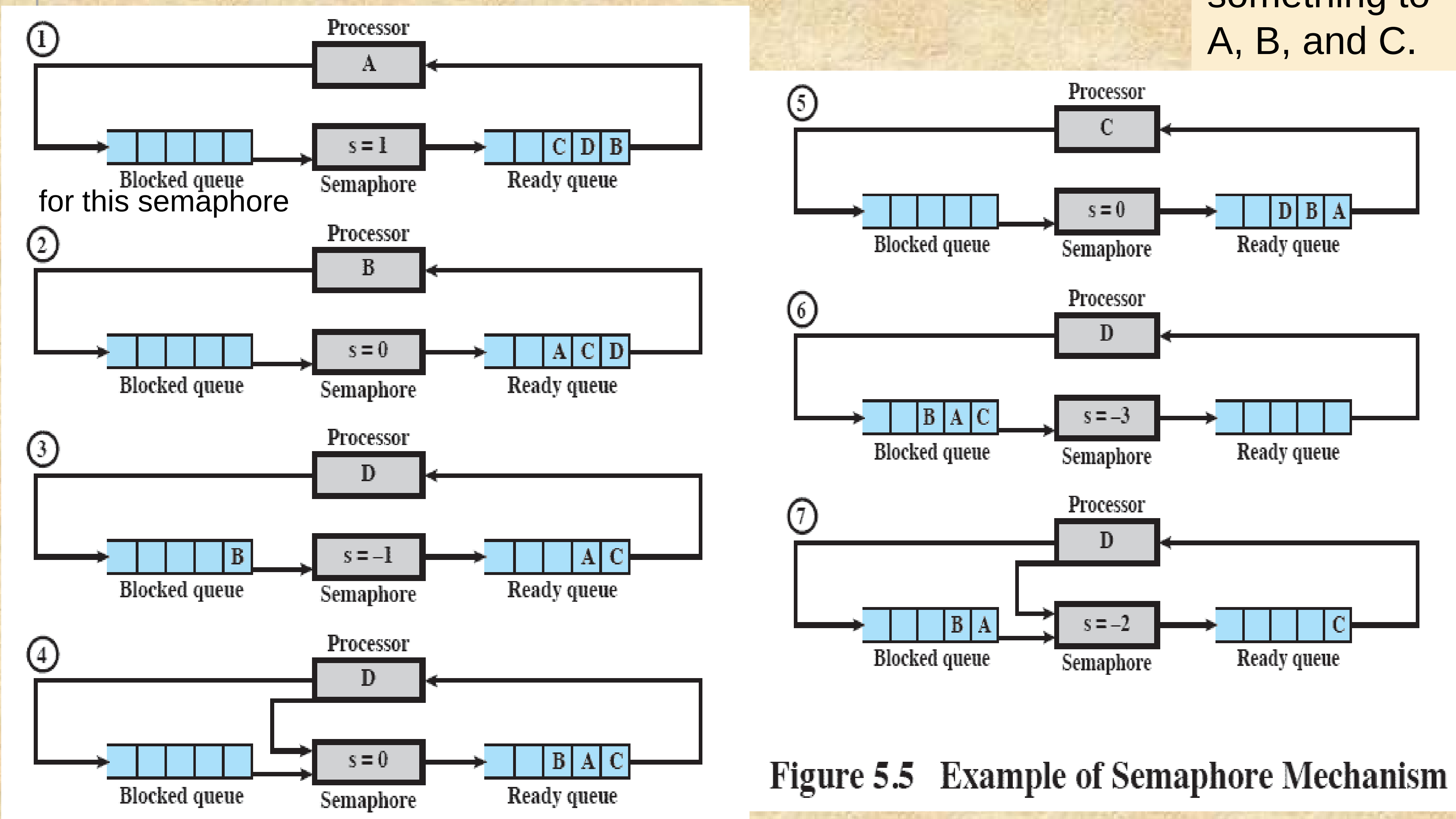


Figure 5.5 Example of Semaphore Mechanism

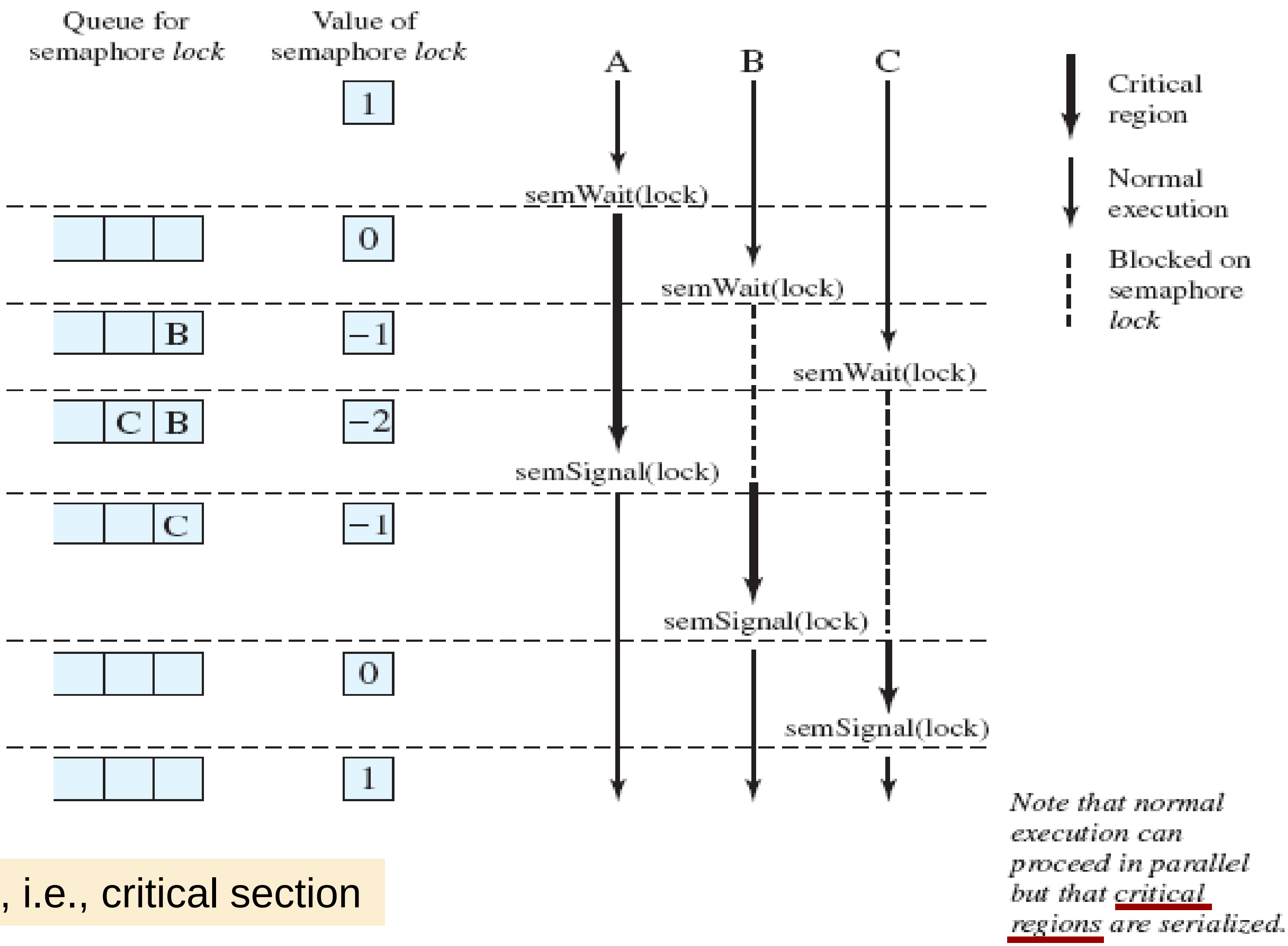
Mutual Exclusion

```
/* 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));
}
```

”permit count” = 1 for mutex protection

Figure 5.6 Mutual Exclusion Using Semaphores

Shared Data Protected by a Semaphore



critical region, i.e., critical section

Figure 5.7 **Processes Accessing Shared Data Protected by a Semaphore**

Producer/Consumer Problem

General Situation:

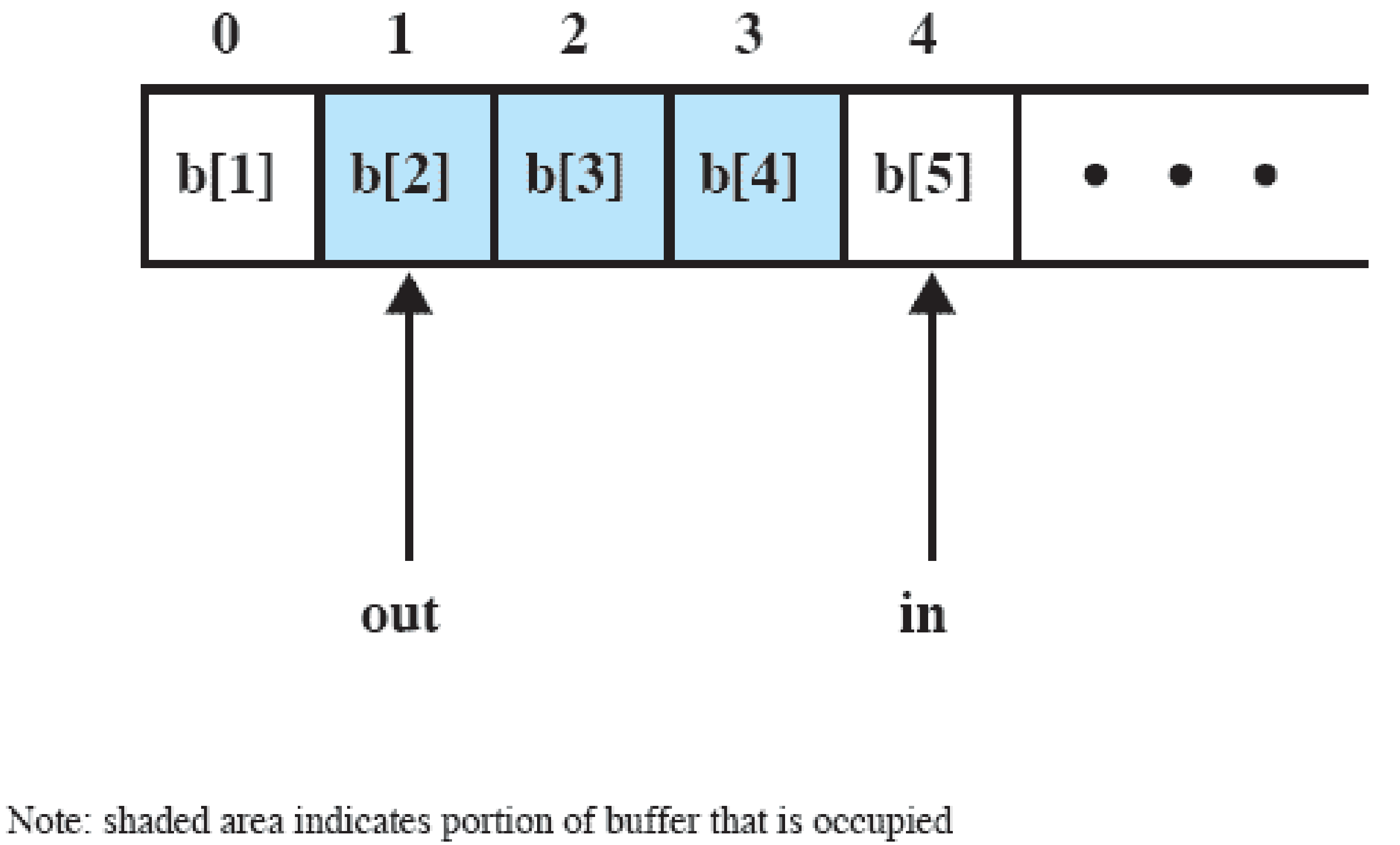
- one or more producers are generating data and placing these in a buffer
- a single consumer is taking items out of the buffer one at time
- only one producer or consumer may access the buffer at any one time

The Problem:

- ensure that the producer can't add data into full buffer and consumer can't remove data from an empty buffer

Can have also many consumers

Buffer Structure



”infinite” buffer is a buffer that can not overflow, no need to worry about buffer overflow!

Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

Incorrect producer / consumer solution (binary sema-phores)

```
/* 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);
}
```

Many problems, failing scenario?

Figure 5.9 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Possible Solution

Table 5.4 Possible Scenario for the Program of Figure 5.9

	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		semiSignlaB(s)	1	–1	0

NOTE: White areas represent the critical section controlled by semaphore s.

Correct producer / consumer solution (binary sema-phores)

```
/* 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);
}
```

Figure 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

Producer / Consumer, infinite buffer, (general) sema-phores

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

mutex solution

synchronization solution

mutex solution

Figure 5.11 A Solution to the Infinite-Buffer Producer/Consu lem Using Semaphores

Discuss

Finite Circular Buffer

Block on:	Unblock on:
Producer: insert in full buffer	Consumer: item inserted
Consumer: remove from empty buffer	Producer: item removed

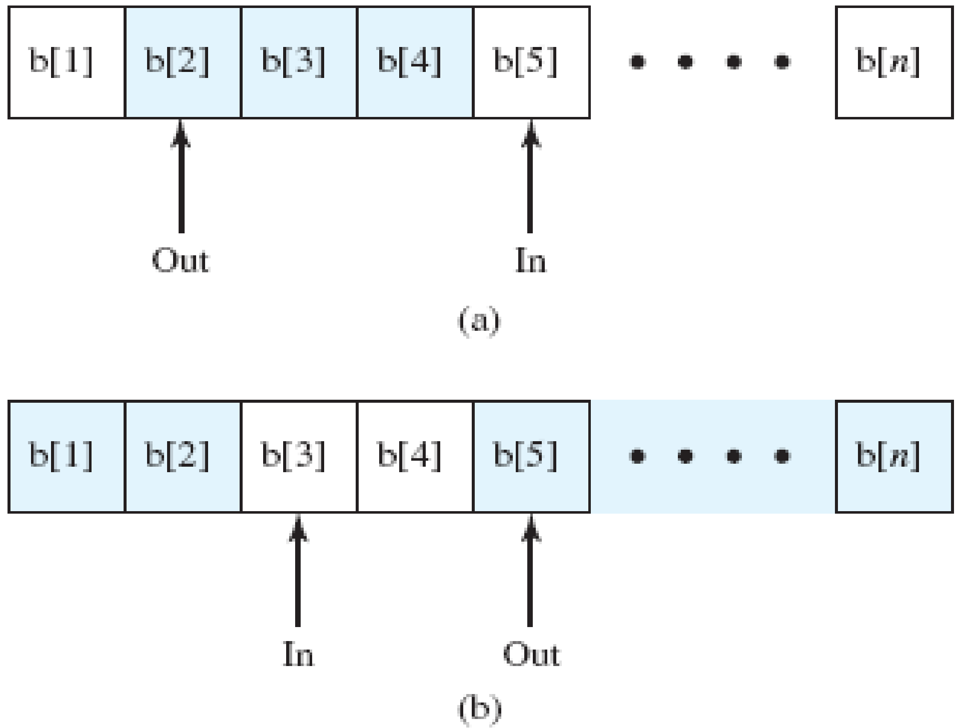
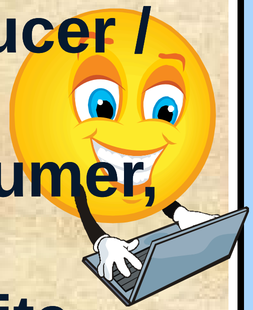


Figure 5.12 **Finite Circular Buffer for the Producer/Consumer Problem**

Producer / Consumer, finite buffer, (general) sema-phores



```
/* 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);
}
```

synchronize for empty slot in buffer

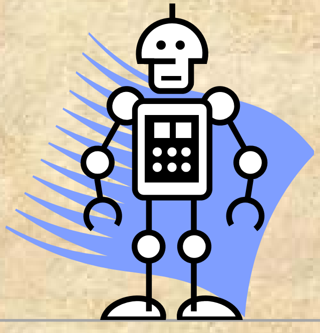
synchronize for full slot in buffer

Discuss

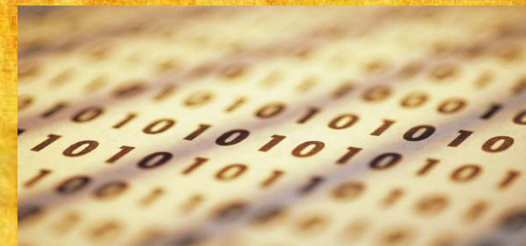
Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

Implementation of Semaphores

- 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



Monitors



- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java
- Has also been implemented as a program library
- Software module consisting of one or more procedures, an initialization sequence, and local data

Monitor Characteristics

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

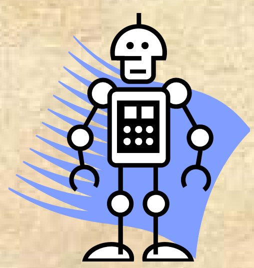
Process enters monitor by invoking one of its procedures

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
  - 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

Monitor Structure



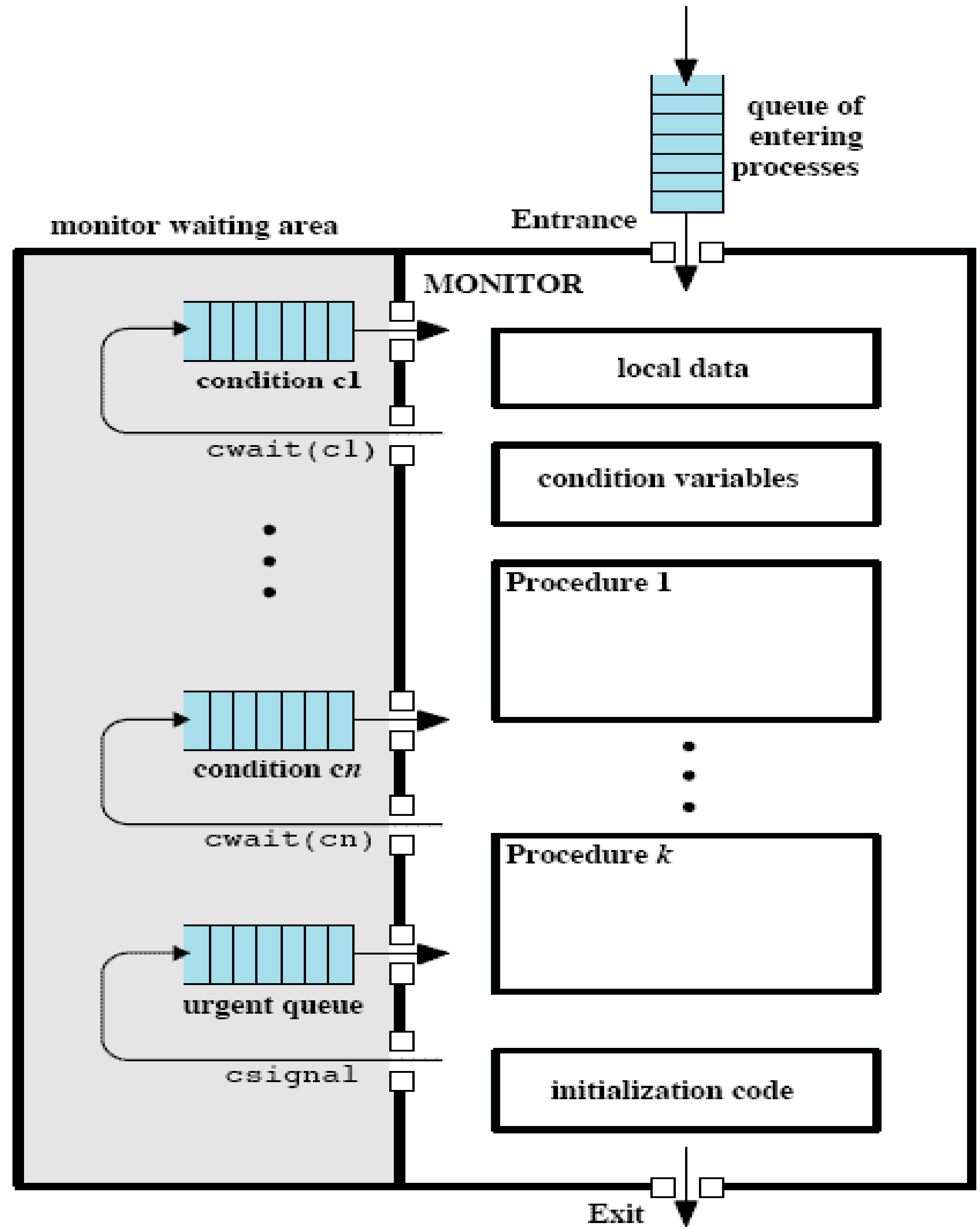


Figure 5.15 Structure of a Monitor

Declaration and CWait

- Condition CV
  - Declare new condition variable
  - No value, just fifo queue of waiting processes
- CWait(CV)
  - Always suspends, process placed in queue
  - Unlocks monitor mutex
    - Allows someone else into monitor?
    - Allows another process awakened from (another?) CWait to proceed?
    - Allows process that lost mutex in CSignal to proceed?
  - When awakened, waits for mutex lock to proceed
    - Not really ready-to-run yet

CSignal

- Wakes up first waiting process, if any
  - Which one continues execution in monitor (in mutex)?
    - The process doing the signalling? (Lampson & Redell)
    - The process just woken up? (Hoare, original)
    - Some other processes trying to get into monitor? No.
  - Two signalling disciplines (two semantics)
    - Signal and continue - signalling process keeps mutex
    - Signal and wait - signalled process gets mutex
- If no one was waiting, signal is lost (no memory)
  - Advanced signalling (with memory) must be handled in some other manner

Signaling Semantics

- <u>Signal and Wait</u>   *CSignal (CV)*   Hoare
  - Awakened (signalled) process executes immediately
    - Mutex *<u>baton passing</u>*
      - No one else can get the mutex lock at this time
    - Condition waited for is certainly true when process resumes execution
  - Signaller waits in mutex lock
    - With other processes trying to enter the semaphore
    - No priority, or priority over arrivals for mutex?
    - Process may lose mutex at any signal operation
      - But does not lose, if no one was waiting!
      - Problem, if critical section would continue over CSignal

Signaling Semantics

- Signal and Continue *CSignal(CV)* Lampson & Redell
  - ***Signaller process continues***
    - Mutex can not terminate at signal operation
  - Awakened (signalled) process will wait in mutex lock
    - With other processes trying to enter the semaphore
    - May not be the next one active
      - Many control variables signalled by one process?
    - Condition waited for may not be true any more once awaked process resumes (becomes active again)
    - No priority or priority over arrivals for sem. mutex?

Producer and Consumer with Monitor

Hoare

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

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];
int nextin, nextout;
int count;
cond notfull, notempty;          /* condit

void append (char x)
{
     if (count == N) cwait(notfull);
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);
}
void take (char x)
{
     if (count == 0) cwait(notempty);   /*
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;
     csignal(notfull);
}
{
     nextin = 0; nextout = 0; count = 0;
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

Another Producer/Consumer with Monitor

Producer **(size N buffer)**

```
Loop forever
        D <- Produce
        PC.append_ok(N)
          append_tail(buffer, D)
        PC.append_done()
```

Consumer

```
Loop forever
        PC.take_ok()
          D <- head(buffer)
        PC.take_done()
        consume(D)
```

Monitor PC Hoare

Int buf_cnt = 0;
condition notEmpty, notFull;

Operation ***append_ok***(N)
  if (buf_cnt==N)
    Cwait(notFull)
  buf_cnt++;

Operation ***append_done***()
  Csignal(notEmpty)

Operation ***take_ok***()
  if (buf_cnt==0)
    Cwait(notEmpty)
  buf_cnt--;

Operation ***take_done***()
  Csignal(notFull)

Discuss

Which one is better? Why?

Summary le06

- Semaphores
  - Synchronization with semaphores
- Monitors
  - Mutex and synchronization with monitors
  - Differences between semaphores and condition variables
- Producer/consumer problem
  - Solutions with semaphores and monitor

Lecture 7
Messages, Readers/Writers

- Message passing introduction
- Mutex with messages
- Producer/consumer with messages
- Readers/writers problem
- Readers/writers problem with semaphores
- Readers/writers problem with with messages

Message Passing

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

synchronization	communication
• e.g., 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*

Message Passing

Synchronization
- Send
  - blocking
  - nonblocking
- Receive
  - blocking
  - nonblocking
  - test for arrival

Addressing
- Direct
  - send
  - receive
    - explicit
    - implicit
- Indirect
  - static
  - dynamic
  - ownership

Format
- Content
- Length
  - fixed
  - variable

Queuing Discipline
- FIFO
- Priority

Table 5.5 Design Characteristics of Message Systems for Interprocess Communication and Synchronization

Synchronization

Communication of a message between two processes implies synchronization between the two

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

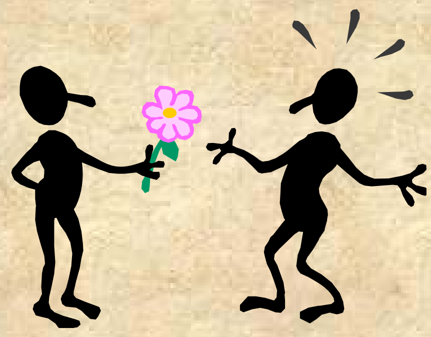
When a receive primitive is executed in a process there are two possibilities:

if there is no waiting message the process is blocked until a message arrives or the process continues to execute, abandoning the attempt to receive

if a message has previously been sent the message is received and execution continues

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



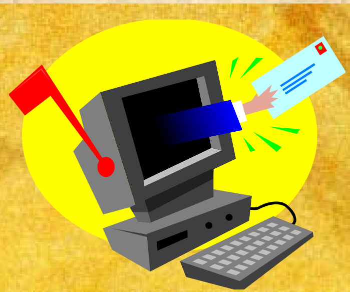
Nonblocking Send

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
- can message be lost?

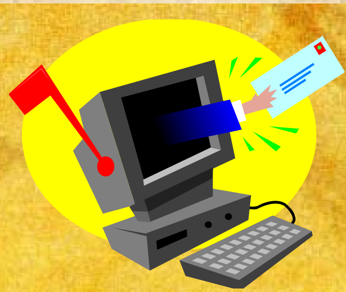


Addressing

- Schemes for specifying processes in `send` and `receive` primitives fall into two categories:

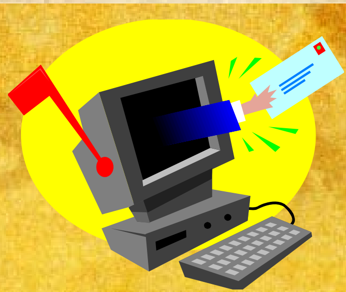
Direct addressing	Indirect addressing

pid



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

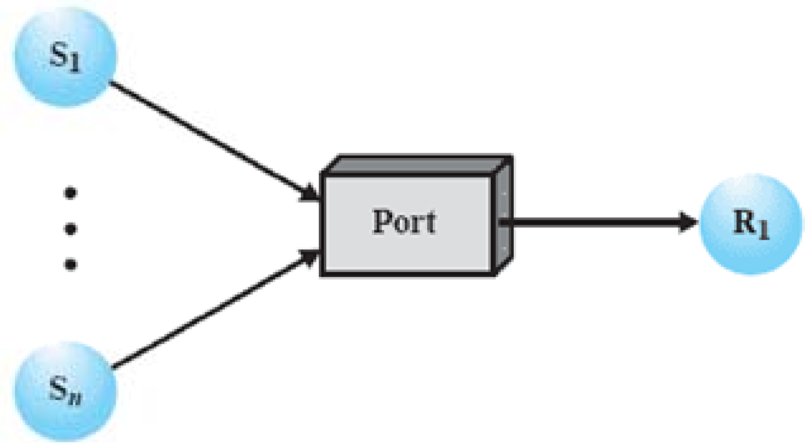
Queues are referred to as *mailboxes*

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

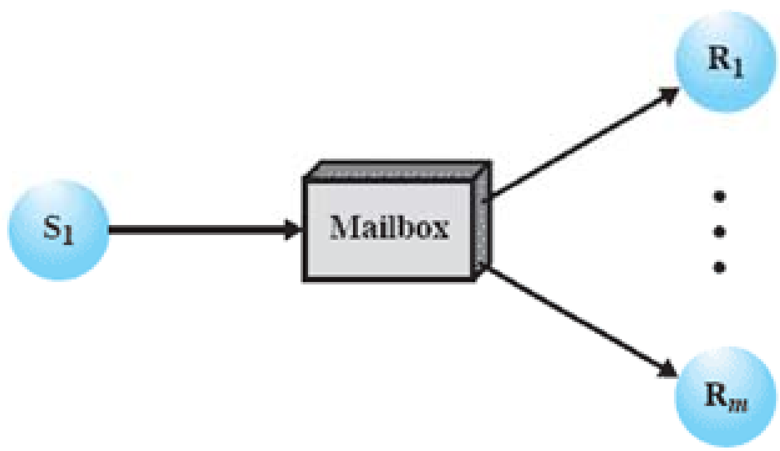
Allows for greater flexibility in the use of messages

Indirect Process Communication

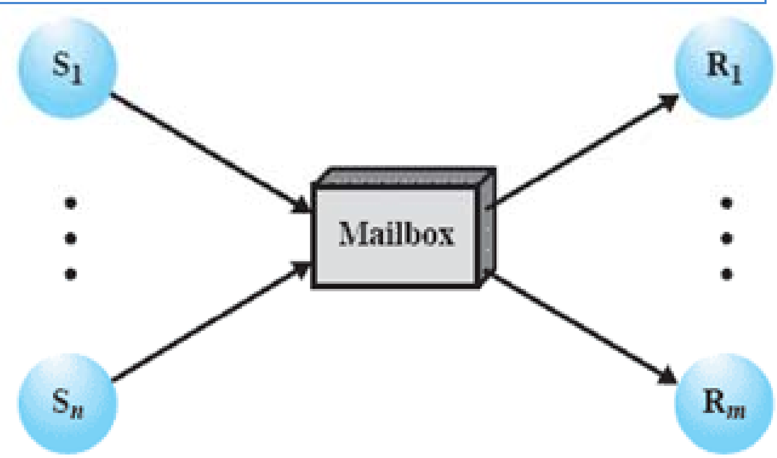
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

General Message Format

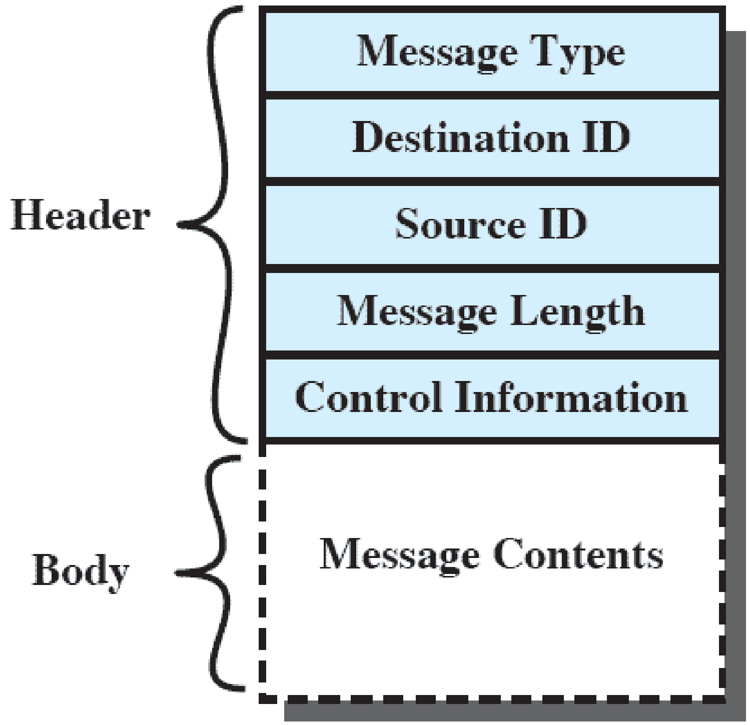


Figure 5.19 General Message Format

Mutual Exclusion with Message “token”

```
/* program mutualexclusion */
const int n = /* number of processes  */;
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));
}
```

create ”token”

Figure 5.20 Mutual Exclusion Using Messages

Discuss

Prod/Cons with 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);
}
```

centralized solution
based on message server

does not scale up

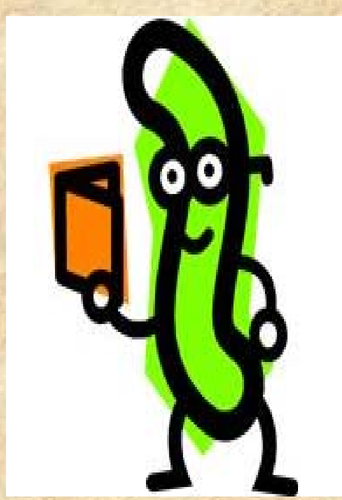
permits to produce
permits to consume

Figure 5.21

Readers/Writers Problem with Semaphores

- A data area is shared among many processes
  - some processes only read the data area, (readers) and some only write to the data area (writers)
- Conditions that must be satisfied:
  1. any number of readers may simultaneously read the file
  2. only one writer at a time may write to the file
  3. if a writer is writing to the file, no reader may read it

Readers Have Priority Solution



mutex

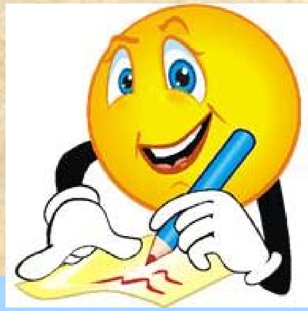
```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
     semWait (x);
     readcount++;
     if (readcount == 1) semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0) semSignal (wsem);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

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

1st reader waits for reader turn

All writers wait for own turn

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphore: Readers Have Priority

Writers Have Priority



```
/* program readersandwriters */
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);
   }
}
```

other new readers wait here

1st new reader waits here

serializes all readers and each writer

```
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 (readers, writers); /* many */
}
```

switch to writers

Discuss

Figure 5.23 A Solution to the Readers/Writers Problem Using Semaphore: Writers Have Priority

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 one writer queues on rsem other writers queue on y new readers (after 1st writer) queue on rsem
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*

Table 5.6 State of the Process Queues for Program of Figure 5.23

2013

```
process Reader[i = 1 to M] {
  while (true) {
    # ⟨await (nw == 0) nr = nr+1;⟩
      P(e);
      if (nw > 0) { dr = dr+1; V(e); P(r);}
      nr = nr+1;
      if (dr > 0) { dr = dr-1; V(r);}
      else V(e);
    read the database;
    # ⟨nr = nr-1;⟩
      P(e);
      nr = nr-1;
      if (nr == 0 and dw > 0)
        { dw = dw-1; V(w); }
      else V(e);
  }
}
```

next reader

1st reader

1st writer

Fig. 4.13 [Andr00]: readers / writers readers first solution using baton passing

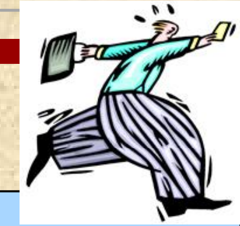
viestikapula

```
process Writer[j = 1 to N] {
  while (true) {
    # ⟨await (nr==0 and nw==0) nw = nw+1;⟩
      P(e);
      if (nr > 0 or nw > 0)
        { dw = dw+1; V(e); P(w); }
      nw = nw+1;
      V(e);
    write the database;
    # ⟨nw = nw-1;⟩
      P(e);
      nw = nw-1;
      if (dr > 0) { dr = dr-1; V(r); }
      elseif (dw > 0) { dw = dw-1; V(w); }
      else V(e);
  }
}
```

nr = nr of readers
dr = nr of delayed readers
nw = nr of writers
dw = nr of delayed writers
Sem e=1, r=0, w=0;
(split binary semaphores, e+r+w=1)

Discuss

Readers/Writers with 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);
   }
}
```

int count = 100;

```
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++;
         }
      }
}
```

no new readers

the writer can proceed

Discuss

Figure 5.24

Lecture 7 Summary

- Messages can be used for synchronization
  - Suitable also for distributed systems
  - Suitable for any situation for processes without shared memory
  - Solutions for critical section, consumer/producer, readers/writers
- Readers/writers problem
  - Complex problem, difficult synchronization
  - Needs critical sections within solution
  - With baton passing can give critical section to certain class of processes instead of just releasing it to anybody to grab next
  - Split binary semaphores can be used for baton passing