Operating Systems
CS-C3140, Lecture 7
Concurrency: Mutual Exclusion and Synchronization

Alexandru Paler

Multiple Processes

- Operating System design is 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
- Distributed Processing
  - The management of multiple processes executing on multiple, distributed computer systems
  - The recent proliferation of clusters is a prime example of this type of system

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

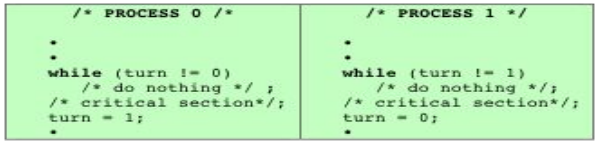
Key Terms Related to Concurrency

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

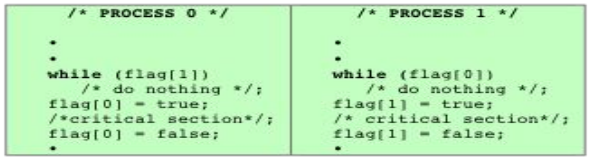
- Software approaches can be implemented for concurrent processes that execute on a single-processor or a multiprocessor machine with shared main memory
- These approaches usually assume elementary mutual exclusion at the memory access level
  - Simultaneous accesses (reading and/or writing) to the same location in main memory are serialized by some sort of memory arbiter, although the order of access granting is not specified ahead of time
  - Beyond this, no support in the hardware, operating system, or programming language is assumed
- Dijkstra reported an algorithm for mutual exclusion for two processes, designed by the Dutch mathematician Dekker

Mutual Exclusion Attempts (1)



```
/* 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



```
/* 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

Figure 5.1 Mutual Exclusion Attempts (page 1)

```
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
```

Because both processes are now in their critical sections, the program is incorrect. The problem is that the proposed solution is not independent of relative process execution speeds.

First Attempt

- Reserve a global memory location labeled turn.
- A process examines the contents of turn.
  - If the value of turn is equal to the number of the process, then the process may proceed to its critical section.
  - Otherwise, it is forced to wait.
  - The waiting process repeatedly reads the value of turn until it is allowed to enter its critical section.
- This procedure is known as busy waiting, or spin waiting, because the thwarted process can do nothing productive until it gets permission to enter its critical section.
- A serious problem is that if one process fails, the other process is permanently blocked. This is true whether a process fails in its critical section or outside of it.

Second Attempt

- We need state information about both processes.
- Each process should have its own key to the critical section so that if one fails, the other can still access its critical section.
- A Boolean vector flag is defined, with flag[0] corresponding to P0 and flag[1] corresponding to P1.
- Each process may examine the other's flag but may not alter it. When it leaves its critical section, it sets its flag to false.

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);
}
```

Figure 5.2 Dekker's Algorithm

A solution

- the array variable **flag** is for observing the state of both processes
- the variable **turn** impose an order on the activities of the two processes in order to avoid the problem of ”mutual courtesy” – which process has the right to insist on entering its critical region.

When P0 wants to enter its critical section

- it sets its flag to true.
- It then checks the flag of Pl.
  - If that is false , P0 may immediately enter its critical section.
  - Otherwise, P0 consults turn.
    - If P0 finds that turn = 0 , then it knows that it is its turn to insist and periodically checks Pl’s flag.
    - Pl will at some point note that it is its turn to defer and set its to flag false , allowing P0 to proceed.
- After P0 has used its critical section, it sets its flag to false to free the critical section, and sets turn to l to transfer the right to insist to Pl

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

- The sharing of global resources is fraught with peril
  - two processes both make use of the same global variable and perform reads and writes on that variable
  - the order in which the various reads and writes are executed is critical
- It is difficult for the OS to manage the allocation of resources optimally
  - a process may request use of, and be granted control of, a particular I/O channel and then be suspended before using that channel
  - it may be undesirable for the OS simply to lock the channel and prevent its use by other processes
  - may lead to a deadlock condition
- Very difficult to locate a programming error because results are typically not deterministic and reproducible

All of the foregoing difficulties present themselves in a multiprocessor system, as well

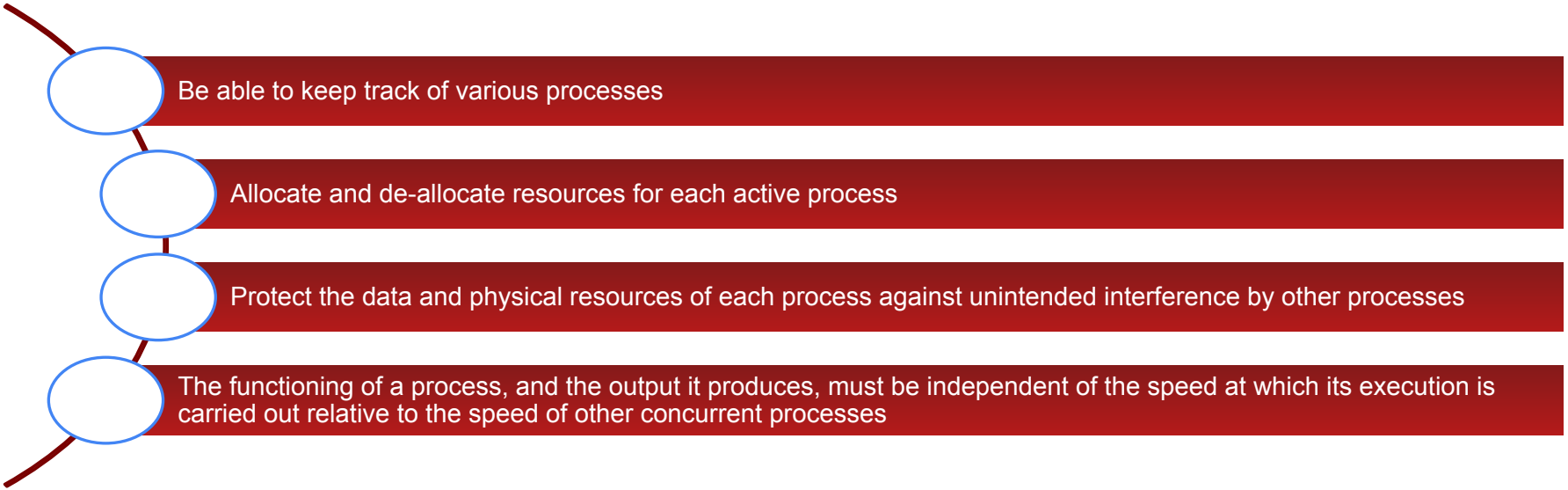
- the relative speed of execution of processes is unpredictable
- deal with problems arising from the simultaneous execution of multiple processes

Race Condition

- A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes. Let us consider two simple examples.
- First example
  - two processes, P1 and P2, share the global variable **a**
  - at some point in its execution, P1 updates a to the value 1
  - at some point in its execution, P2 updates a to the value 2
  - the two tasks are in a **race to write variable a**
  - the “loser” of the race (the process that updates last) determines the final value of a
- Second example
  - two process, P3 and P4, that share global variables b and c , with initial values b = 1 and c = 2
  - at some point in its execution, P3 executes the assignment b = b + c
  - at some point in its execution, P4 executes the assignment c = b + c
  - the final values of b and c depend on the order in which the two processes execute
  - if P3 executes its assignment statement first, then the final values are b = 3 and c = 5
  - if P4 executes its assignment statement first, then the final values are b = 4 and c = 3

Operating System Concerns

- Design and management issues raised by the existence of concurrency. The OS must:



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

1. 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**:
    - For example, consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: the OS assigns R1 to P2, and R2 to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.
    - **Starvation:**
    - Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the OS grants access to P3 and that P1 again requires access before P3 completes its critical section. If the OS grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

1. Resource Competition

```
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 5.4 Illustration of Mutual Exclusion

Control of competition

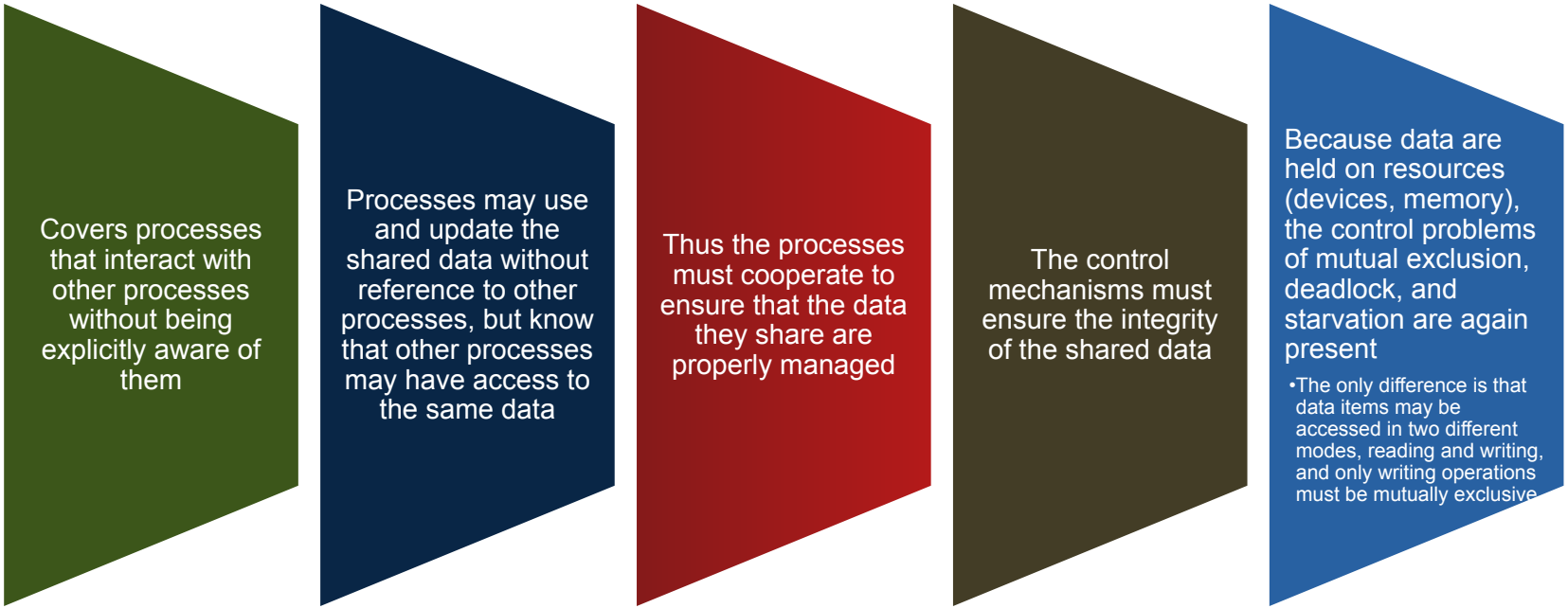
- processes need to be able to express the requirement for mutual exclusion in some fashion, such as **locking** a resource prior to its use
- involves the OS, such as the provision of the locking facility

There are *n processes to be executed concurrently.* Each process includes:

- a critical section that operates on some resource
- additional code preceding and following the critical section that does not involve access to the resource
- any process that attempts to enter its critical section while another process is in its critical section, for the same resource, is made to wait

To enforce mutual exclusion, two functions are provided: entercritical and exitcritical

2. Cooperation by Sharing



3. Cooperation by Communication

- The various processes participate in a common effort that links all of the processes
- The communication provides a way to synchronize, or coordinate, the various activities
- Communication can be characterized as consisting of messages of some sort
- Primitives for sending and receiving messages may be provided as part of the programming language or provided by the OS kernel
- Mutual exclusion is not a control requirement for this sort of cooperation
- The problems of deadlock and starvation are still present

Requirements for Mutual Exclusion

- Any facility or capability that is to provide support for mutual exclusion should meet the following requirements
  - Mutual exclusion must be enforced: only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object
  - A process that halts must do so without interfering with other processes
  - It must not be possible for a process requiring access to a critical section to be delayed indefinitely: **no deadlock or starvation**
  - When no process is in a critical section, any process that request entry to its critical section must be permitted to enter without delay
  - 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: Hardware Support (1)

Interrupt Disabling

- In a uniprocessor system
  - concurrent processes cannot have overlapped execution
  - can only be interleaved
- A process will continue to run until it invokes an OS service or until it is interrupted
- To guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted
- This capability can be provided in the form of primitives defined by the OS kernel for disabling and enabling interrupts

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

Mutual Exclusion: Hardware Support (2)

- 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 (not subject to interruption)

```
/* 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

Figure 5.5 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

- **Disadvantages**
- Busy-waiting is employed
  - Thus while a process is waiting for access to a critical section it continues to consume processor time
- Deadlock is possible
- Starvation is possible
  - When a process leaves a critical section and more than one process is waiting, the selection of a waiting process is arbitrary; some process could indefinitely be denied access

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) A semaphore may be initialized to a nonnegative integer value
2) The semWait operation decrements the semaphore value
3) The semSignal operation increments the semaphore 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

General Semaphores

```
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 */;
      }
}
```

Figure 5.6 A Definition of Semaphore Primitives

- 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

Binary Semaphore

```
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 5.7 A Definition of Binary Semaphore Primitives

- A binary semaphore may be initialized to 0 or 1.
- The semWaitB operation
  - checks the semaphore value
  - If the value is zero, then the process executing the semWaitB is blocked
  - If the value is one, then the value is changed to zero and the process continues execution.
- The semSignalB operation
  - checks to see if any processes are blocked on this semaphore (semaphore value equals 0). If so, then a process blocked by a semWaitB operation is unblocked. If no processes are blocked, then the value of the semaphore is set to one.
- Mutual exclusion lock (mutex)
  - concept related to the binary semaphore
  - a programming flag used to grab and release an object
  - set to lock (typically zero), which blocks other attempts to use it.
  - set to unlock when the data are no longer needed or the routine is finished
- A key difference between the a mutex and a binary semaphore
  - mutex: the process that locks (sets the value to zero) must be the one to unlock it (sets the value to 1)
  - semaphore: it is possible for one process to lock a binary semaphore and for another to unlock it

Example: Strong Semaphore

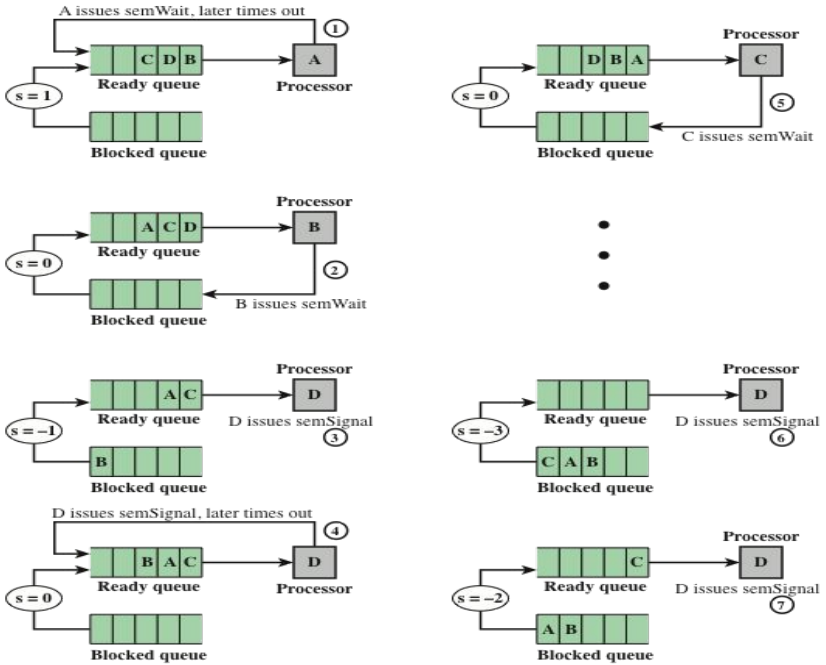


Figure 5.8 Example of Semaphore Mechanism

Processes A, B, and C depend on a result from process D.
(1), A is running; B, C, and D are ready; and the semaphore count is 1, indicating that one of D's results is available. When A issues a semWait instruction on semaphore *s* , the semaphore decrements to 0, and A can continue to execute; subsequently it rejoins the ready queue.
(2) Then B runs , eventually issues a semWait instruction, and is blocked, allowing D to run
(3). When D completes a new result, it issues a semSignal instruction, which allows B to move to the ready queue
(4) D rejoins the ready queue and C begins to run
(5) but is blocked when it issues a semWait instruction. Similarly, A and B run and are blocked on the semaphore, allowing D to resume execution
(6). When D has a result, it issues a semSignal , which transfers C to the ready queue. Later cycles of D will release A and B from the Blocked state.

Semaphore for 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));
}
```

Figure 5.9 Mutual Exclusion Using Semaphores

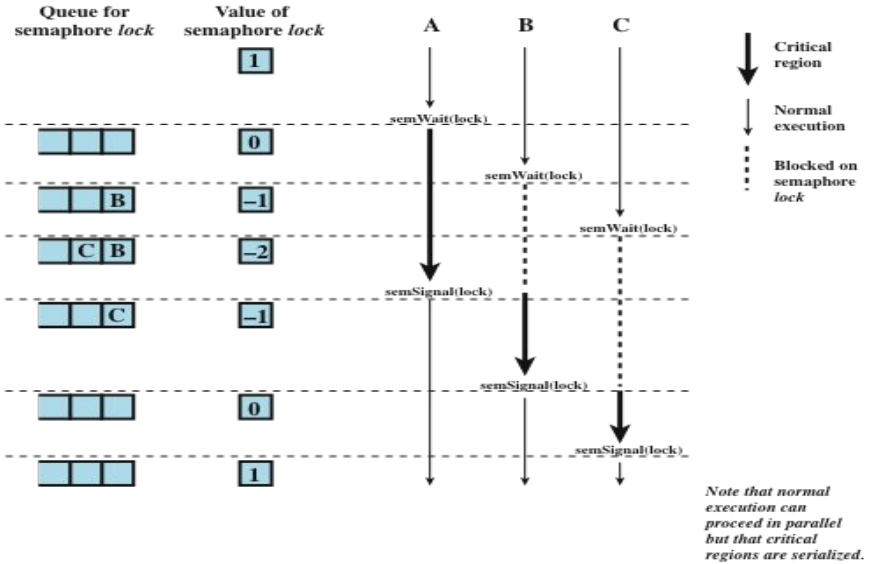


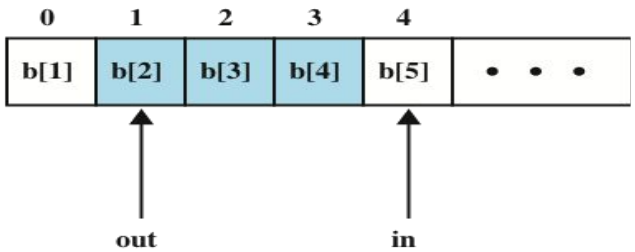
Figure 5.10 Processes Accessing Shared Data Protected by a Semaphore

Producer/Consumer Problem

General Statement:

- 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 a time
- Only one producer or consumer may access the buffer at any one time

The Problem: Ensure that the producer won't try to add data into the buffer if its full, and that the consumer won't try to remove data from an empty buffer



Note: shaded area indicates portion of buffer that is occupied

Figure 5.11 Infinite Buffer for the Producer/Consumer Problem

Incorrect Solution:

```
/* 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);
}
```

Figure 5.12 An Incorrect Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

	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

Implementation of Semaphores

- 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
- Alternative: use one of the hardware-supported schemes for mutual exclusion

```
semWait(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue*/;
        /* block this process (must also set s.flag to 0)
*/;
    }
    s.flag = 0;
}

semSignal(s)
{
    while (compare_and_swap(s.flag, 0 , 1) == 1)
        /* do nothing */;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    s.flag = 0;
}
```

(a) Compare and Swap Instruction

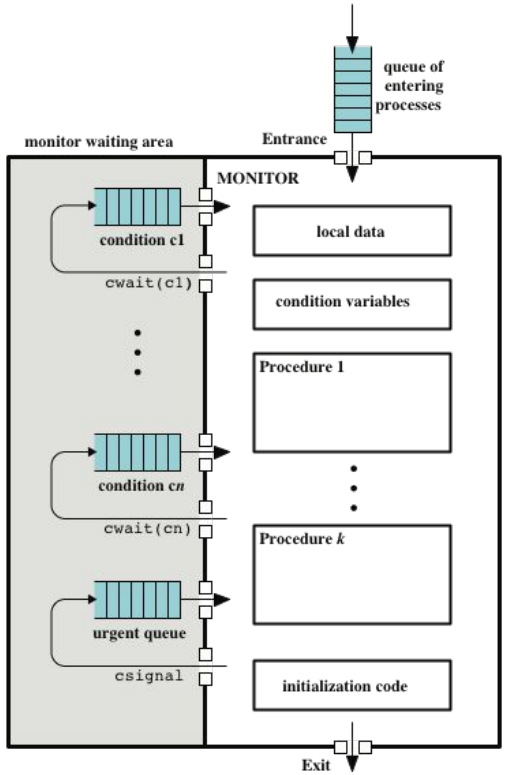
```
semWait(s)
{
    inhibit interrupts;
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process and allow interrupts */;
    }
    else
       allow interrupts;
}

semSignal(s)
{
    inhibit interrupts;
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
    allow interrupts;
}
```

(b) Interrupts

Figure 5.17 Two Possible Implementations of Semaphores

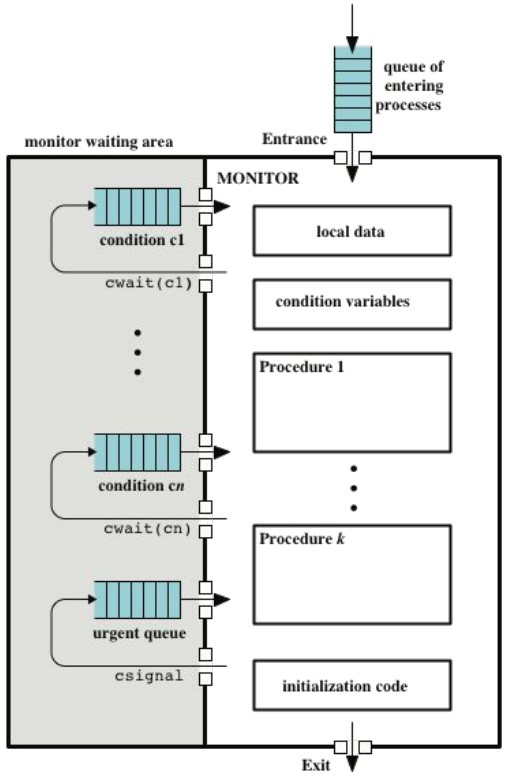
Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, Java
  - Has also been implemented as a program library
  - In particular, for something like a linked list, you may want to lock all linked lists with one lock, or have one lock for each list, or have one lock for each element of each list.
- A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data.



Monitor Synchronization

- 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
- A monitor supports synchronization by the use of condition variables that are contained within the monitor and accessible only within the monitor
    - **Condition variables** are a special data type in monitors which 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



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*

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, 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

Addressing in send and receive primitives

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

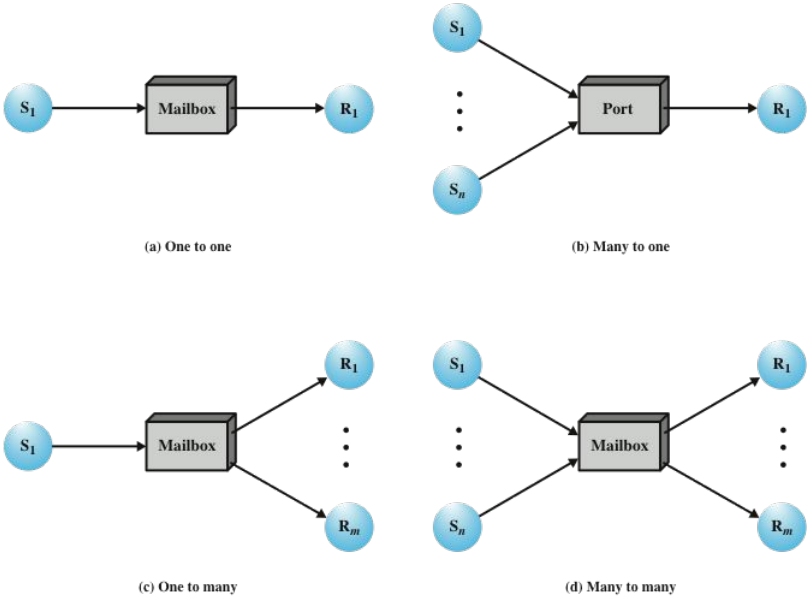


Figure 5.21 Indirect Process Communication

- A one-to-one relationship
  - allows a private communications link to be set up between two processes.
  - This insulates their interaction from erroneous interference from other processes.
- A many-to-one relationship is useful for client/server interaction
  - one process provides service to a number of other processes.
  - The mailbox is often referred to as a port
- A one-to-many relationship allows for one sender and multiple receivers
  - useful for applications where a message or some information is to be broadcast to a set of processes
- A many-to-many relationship
  - allows multiple server processes to provide concurrent service to multiple clients
- **Queueing Discipline**
  - Simplest queueing discipline is first-in-first-out
  - Message priority
  - Allow the receiver to inspect the message queue and select which message to receive next

Summary

- Mutual exclusion
  - Software approaches
  - Dekker's algorithm
- Principles of concurrency
  - Race condition
  - OS concerns
  - Process interaction
  - Requirements for mutual exclusion
- Mutual exclusion: hardware support
  - Interrupt disabling
  - Special machine instructions
- Semaphores
  - Mutual exclusion
  - Producer/consumer problem
  - Implementation of semaphores

- Message passing
  - Synchronization
  - Addressing
  - Message format
  - Queueing discipline
  - Mutual exclusion
- Readers/writers problem
  - Readers have priority
  - Writers have priority