Module 4
Concurrency

Inter-process communication, Synchronization - Implementing synchronization primitives (Peterson’s solution, Bakery algorithm, synchronization hardware) - Semaphores – Classical synchronization problems, Monitors: Solution to Dining Philosophers problem – IPC in Unix, Multiprocessors and Locking - Scalable Locks - Lock-free coordination

Interprocess Communication

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it does not share data with any other processes executing in the system.
- A process is cooperating if it can affect or be affected by the other processes executing in the system. Any process that shares data with other processes is a cooperating process.
- There are several reasons for providing an environment that allows process cooperation:

1. Information sharing: The same piece of information can be shared among several applications and an environment must be provided to allow concurrent access to such information.
2. Computation speedup: If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. But such a speedup can be achieved only if the computer has multiple processing cores.
3. Modularity: The system is constructed in a modular fashion, dividing the system functions into separate processes or threads.

Interprocess Communication

- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data— that is, send data to and receive data from each other.
- There are two fundamental models of interprocess communication: shared memory and message passing.
- In the shared-memory model, a region of memory that is shared by the cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
- The two communications models are contrasted in Figure 3.11.

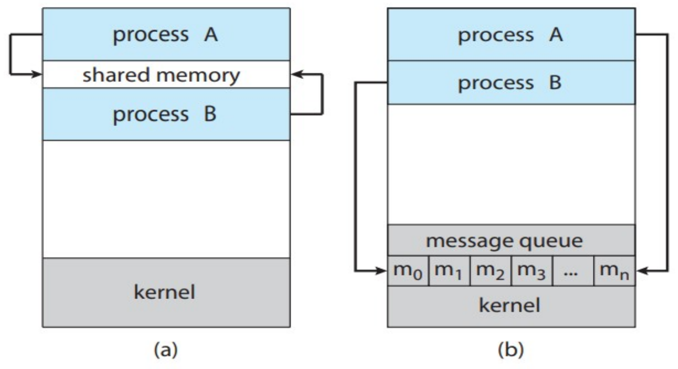


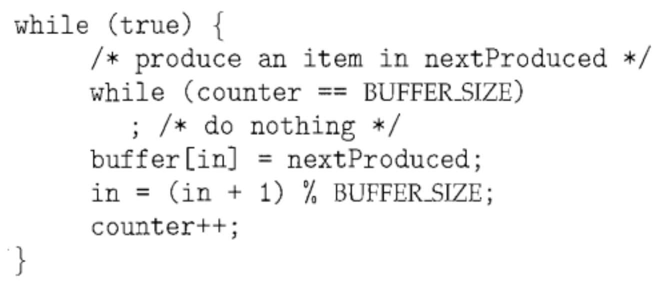
Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Interprocess Communication

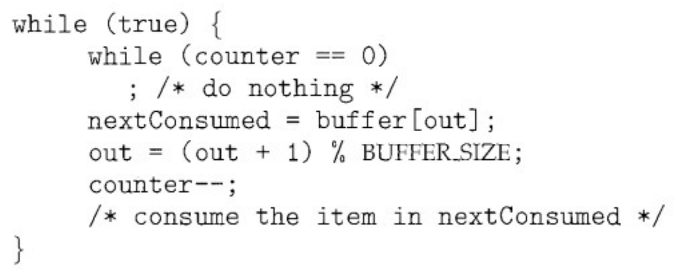
- Message passing is useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- Message passing is also easier to implement in a distributed system than shared memory.
- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention.
- In shared-memory systems, system calls are required only to establish shared memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Synchronization

- We developed a model of a system consisting of cooperating sequential processes or threads, all running asynchronously and possibly sharing data. This is illustrated with the producer-consumer problem.
- The original solution uses bounded buffer and allowed at most BUFFER_SIZE - 1 items in the buffer at the same time.
- Suppose we want to modify the algorithm by adding an integer variable counter, initialized to 0. The counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.
- The code umer s follows:



```
while (true) {
      /* produce an item in nextProduced */
      while (counter == BUFFER_SIZE)
         ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```



```
while (true) {
      while (counter == 0)
         ; /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
      /* consume the item in nextConsumed */
}
```

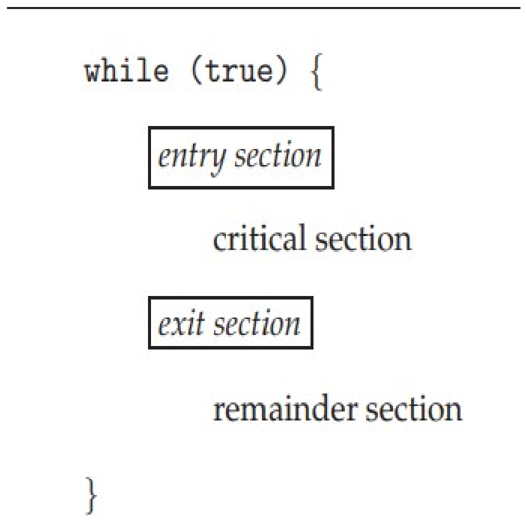
- Both the producer and consumer routines shown above are correct separately, but they may not function correctly when executed concurrently. We would arrive at an incorrect state because we allowed both processes to manipulate the variable counter concurrently.
- A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access

The critical Section Problem

- Consider a system consisting of n processes {P0, P1, ..., Pn−1}.
- Each process has a segment of code, called a critical section, in which the process may be accessing and updating data that is shared with at least one other process.
- The important feature of the system is that, when one process is executing in its critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time.
- The critical-section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data.
- Each process must request permission to enter its critical section.

The critical Section Problem

- The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.
- The general structure of a typical process is shown in Figure 6.1.
- The entry section and exit section are enclosed in boxes to highlight these important segments of code.



```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

The critical Section Problem

- A solution to the critical-section problem must satisfy the following three requirements:

1. Mutual exclusion: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
2. Progress: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.
3. Bounded waiting: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Race Condition - Example

- At a given point in time, many kernel-mode processes may be active in the operating system.
- As a result, the code implementing an operating system (kernel code) is subject to several possible race conditions.
- Consider as an example a kernel data structure that maintains a list of all open files in the system.
- This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list).
- If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Race Condition - Example

- Two processes, P0 and P1, are creating child processes using the fork() system call. fork() returns the process identifier of the newly created process to the parent process.
- In this example, there is a race condition on the variable kernel variable next_available_pid which represents the value of the next available process identifier. Unless mutual exclusion is provided, it is possible the same process identifier number could be assigned to two separate processes.

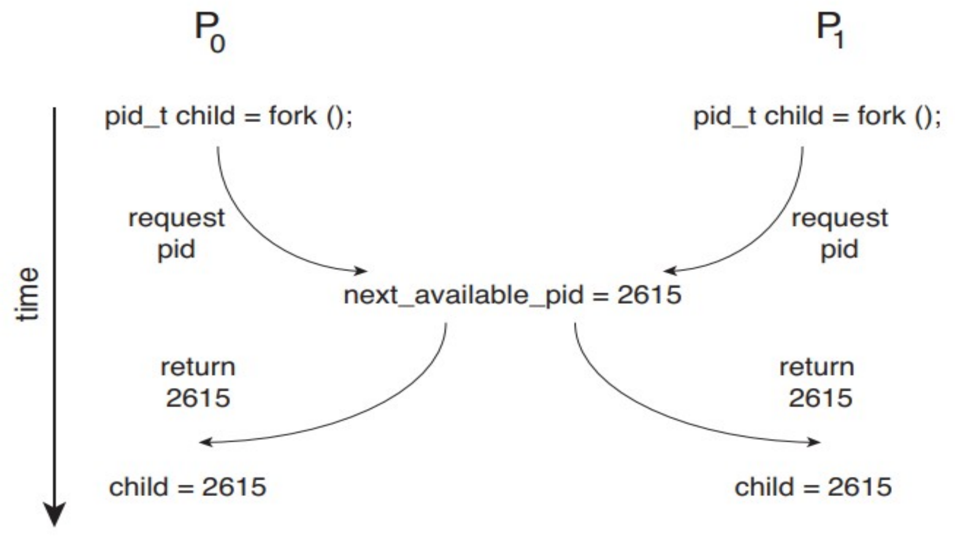


Figure 6.2 Race condition when assigning a pid.

Race Condition - Example

- Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.
- It is up to kernel developers to ensure that the operating system is free from such race conditions.
- The critical-section problem could be solved simply in a single-core environment if we could prevent interrupts from occurring while a shared variable was being modified.
- In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.
- No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Race Condition - Example

- Two general approaches are used to handle critical sections in operating systems: preemptive kernels and non-preemptive kernels.
- A preemptive kernel allows a process to be preempted while it is running in kernel mode.
- A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.
- A non-preemptive kernel is free from race conditions on kernel data structures, as only one process is active in the kernel at a time.
- The preemptive kernels must be carefully designed to ensure that shared kernel data are free from race conditions.
- Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two

Race Condition - Example

Why would anyone favor a preemptive kernel over a non-preemptive one?

- A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. (Of course, this risk can also be minimized by designing kernel code that does not behave in this way.)
- Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

Peterson's Solution

A classic software-based solution to the critical-section problem is known as Peterson's solution.

Most modern computer architectures perform basic machine-language instructions, such as load and store, and there are no guarantees that Peterson's solution will work correctly on such architectures.

However, a solution is presented because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

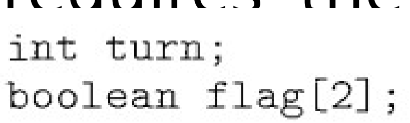
Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections.

The processes are numbered P0 and P1.

For convenience, when presenting Pi , we use Pj to denote the other process; that is, j equals 1 − i.

Peterson’s Solution

Peterson’s solution requires the two processes to share two data items.

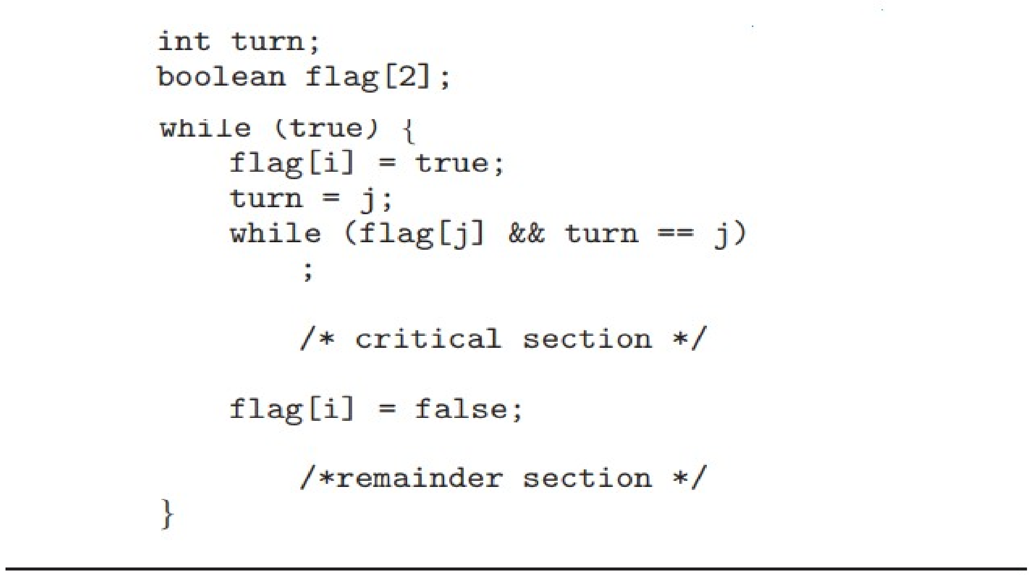


```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section.

That is, if turn == i, then process Pi is allowed to execute in its critical section.

The a process is ready to enter its
criti i] is true, Pi is ready to enter its
criti



```
int turn;
boolean flag[2];

while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Figure 6.3 The structure of process P_i in Peterson’s solution.

Peterson’s Solution

To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.

Only one of these assignments will last; the other will occur but will be overwritten immediately.

The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

Peterson’s Solution

To prove property 1, we note that each Pi enters its critical section only if either flag[j] == false or turn == i.

Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true.

These two observations imply that P0 and P1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both.

Hence, one of the processes say, Pj must have successfully executed the while statement, whereas Pi had to execute at least one additional statement (“turn == j”).

However, at that time, flag[j] == true and turn == j, and this condition will persist as long as Pj is in its critical section; as a result, mutual exclusion is preserved.

Peterson’s Solution

To prove properties 2 and 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible.

If Pj is not ready to enter the critical section, then flag[j] == false, and Pi can enter its critical section.

If Pj has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j.

If turn == i, then Pi will enter the critical section. If turn == j, then Pj will enter the critical section.

However, once Pj exits its critical section, it will reset flag[j] to false, allowing Pi to enter its critical section.

If Pj resets flag[j] to true, it must also set turn to i. Thus, since Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (progress) after at most one entry by Pj (bounded waiting).

Lamport's Bakery Algorithm

- A software solution to solve the critical section problem, for the n process that follows the first come, first serve principle.
- This algorithm is known as the bakery algorithm as this type of scheduling is adopted in bakeries where token numbers are issued to set the order of customers.
- When a customer enters a bakery store, he gets a unique token number on its entry.
- The global counter displays the number of customers currently being served, and all other customers must wait at that time.
- Once the baker finishes serving the current customer, the next number is displayed. The customer with the next token is now being served.
- Similarly, in Lamport's bakery algorithm, processes are treated as customers.
- In this, each process waiting to enter its critical section gets a token number, and the process with the lowest number enters the critical section.
- If two processes have the same token number, the process with a lower process ID enters its critical section.

Lamport's Bakery Algorithm

```
While(true)
{
         entering[i] :=  true;         // show interest in critical section
        // get a token number
        number[i] := 1 + max(number[0],  number[1], ..., number[n - 1]);
        entering [i] :=  false;
        for ( j :=  0 ;  j<n; j++)
        {  // busy wait until process Pj receives its token number
        while (entering [j]) ;
        while (number[j] !=  0 &&  (number[j], j) < (number[i], i)) ;
// token comparison
          }
        // critical section
        number[i] :=  0;  // Exit section
        // remainder section
          }
```

Lamport's Bakery Algorithm

This algorithm uses the following two Boolean variables:

```
boolean entering[n];
int number[n];
```

All entering variables are initialized to false, and n integer variables numbers are all initialized to 0. The value of integer variables is used to form token numbers.

When a process wishes to enter a critical section, it chooses a greater token number than any earlier number.

- Consider a process Pi wishes to enter a critical section, it sets entering[i] to true to make other processes aware that it is choosing a token number. It then chooses a token number greater than those held by other processes and writes its token number. Then it sets entering[i] to false after reading them. Then it enters a loop to evaluate the status of other processes. It waits until some other process Pj is choosing its token number.
- Pi then waits until all processes with smaller token numbers or the same token number but with higher priority are served fast.
- When the process has finished with its critical section execution, it resets its number variable to 0.

- **Mutual Exclusion:** we know that when no process is executing in its critical section, a process with the lowest number is allowed to enter its critical section. Suppose two processes have the same token number. In that case, the process with the lower process ID among these is selected as the process ID of each process is distinct, so at a particular time, there will be only one process executing in its critical section. Thus the requirement of mutual Exclusion is met.
- **Progress:** After selecting a token, a waiting process checks whether any other waiting process has higher priority to enter its critical section. If there is no such process, P will immediately enter its critical section. Thus meeting progress requirements.
- **Bounded Waiting:** As awaiting, the process will enter its critical section when no other process is in its critical section and
  - If its token number is the smallest among other waiting processes.
  - If token numbers are the same, it has the lowest process ID among other waiting processes.

Advantages of Lamport's Bakery Algorithm

- Lamport's bakery algorithms are free from starvation.
- Lamport's Bakery algorithm follows a FIFO.
- Lamport's Bakery algorithm works with atomic registers.
- Lamport's Bakery algorithm is one of the simplest known solutions to the mutual exclusion problem for the general case of the N process.
- This algorithm ensures the efficient use of shared resources in a multithreaded environment.

Synchronization Hardware

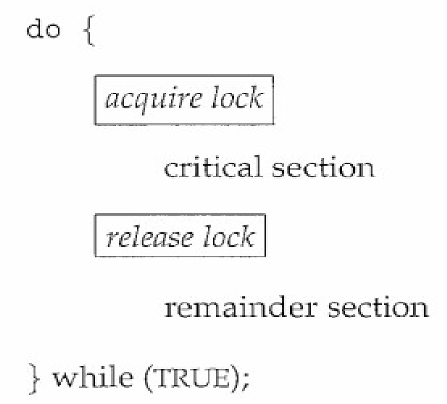
Software-based solutions such as Peterson's and Bakery algorithm are not guaranteed to work on modern computer architectures.

Instead, we can generally state that any solution to the critical-section problem requires a simple tool-a lock.

Race conditions are prevented by requiring that critical regions be protected by locks.

That is, a process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section.

This is illustrated in Figure 6.3.



```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (TRUE);
```

Figure 6.3 Solution to the critical-section problem using locks.

Synchronization Hardware

Hardware instructions can be used effectively in solving the critical-section problem.

Hardware features can make any programming task easier and improve system efficiency.

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occurring while a shared variable was being modified.

In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption.

No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by non-preemptive kernels.

This solution is not feasible in a multiprocessor environment.

Disabling interrupts on a multiprocessor can be time consuming, as the message is passed to all the processors.

This message passing delays entry into each critical section, and

Synchronization Hardware

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically, (ie) as one uninterruptible unit.

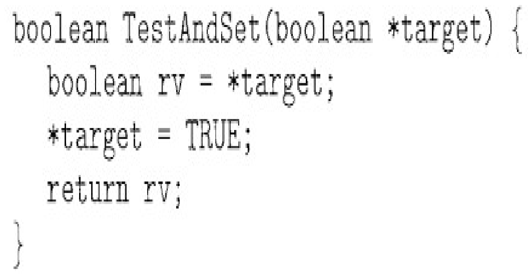
Rather than using one specific instruction for one specific machine, we use TestAndSet () and Swap() instructions.

The TestAndSet () instruction is a hardware solution to synchronization problem and can be defined as shown in Figure 6.4.

The important characteristic of this instruction is that it is executed atomically.

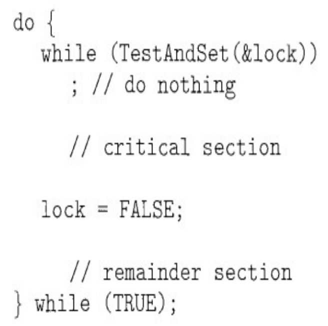
Thus, if two TestAndSet () instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structu[illegible] in Figure 6.5



```
boolean TestAndSet(boolean *target) {
   boolean rv = *target;
   *target = TRUE;
   return rv;
}
```

Figure 6.4 The definition of the TestAndSet() instruction.



```
do {
   while (TestAndSet(&lock))
      ; // do nothing

      // critical section

   lock = FALSE;

      // remainder section
} while (TRUE);
```

Figure 6.5 Mutual-exclusion implementation with TestAndSet().

Synchronization Hardware

The Swap() instruction, in contrast to the TestAndSet () instruction, operates on the contents of two words; it is defined as shown in Figure 6.6.

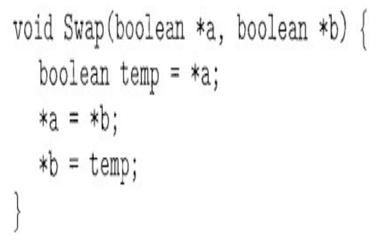
Like the TestAndSet () instruction, it is executed atomically.

If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows.

A global Boolean variable lock is declared and is initialized to false.

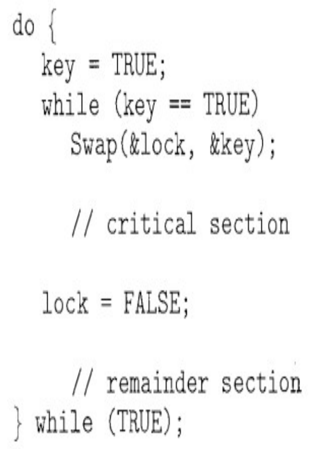
In addition, each process has a local Boolean variable key

The structure of process P1 is shown in Fig



```
void Swap(boolean *a, boolean *b) {
  boolean temp = *a;
  *a = *b;
  *b = temp;
}
```

Figure 6.6 The definition of the Swap() instruction.



```
do {
  key = TRUE;
  while (key == TRUE)
    Swap(&lock, &key);

    // critical section

  lock = FALSE;

    // remainder section
} while (TRUE);
```

Figure 6.7 Mutual-exclusion implementation with the Swap() instruction.

Synchronization Hardware

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

In Figure 6.8, we present another algorithm using the TestAndSet () instruction that satisfies all the critical-section requirements.

The common data structures are

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to false.

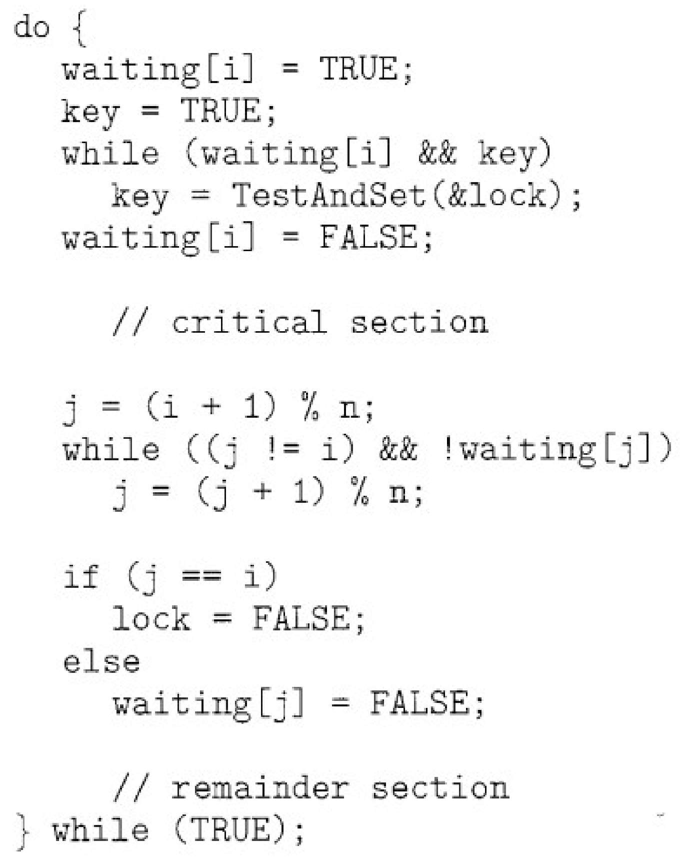
To prove that the mutual exclusion requirement is met, we note that process Pi can enter its critical section only if either waiting [i] == false or key == false.

The value of key can become false only if the TestAndSet () is executed.

The first process to execute the TestAndSet () will find key== false; all others must wait.

The variable waiting [i] can become false only if another process leaves its critical section; only one waiting [i] is set to false, maintaining the mutual-exclusion requirement.

Synchronization Hardware



```
do {
   waiting[i] = TRUE;
   key = TRUE;
   while (waiting[i] && key)
      key = TestAndSet(&lock);
   waiting[i] = FALSE;

      // critical section

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = FALSE;
   else
      waiting[j] = FALSE;

      // remainder section
} while (TRUE);
```

Figure 6.8 Bounded-waiting mutual exclusion with `TestAndSet()`.

Synchronization Hardware

- To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.
- To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ... , n 1, 0, ... , i 1). It designates the first process in this ordering that is in the entry section (waiting[j] ==true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within n - 1 turns.

Semaphores

- A semaphore is a synchronization tool.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal ().
- The wait () operation was originally termed P (Proberan) ("to test"); signal() was originally called V (Verhogen) ("to increment").
- The definition of wait () is as follows:

```
wait(S) {
while S <= 0;
s--; }
```

The definition of signal() is as follows:

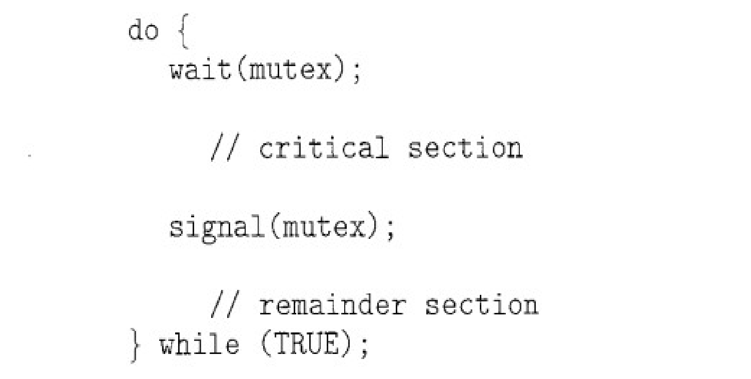
```
signal(S)
{
S++;
}
```

Semaphores

- All modifications to the integer value of the semaphore in the wait () and signal() operations must be executed indivisibly.
- That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of wait (S), the testing of the integer value of S (S<=0), as well as its possible modification (S--), must be executed without interruption.

Usage

- Operating systems often distinguish between counting and binary semaphores.
- The value of a binary semaphore can range only between 0 and 1.
- The value of a counting semaphore can range over an unrestricted domain.
- On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.
- We can use binary semaphores to deal with the critical-section problem for multiple processes. Then n processes share a semaphore, mutex, in
  s Pi is organized as shown in Figure 6.9.



```
do {
  wait(mutex);

    // critical section

  signal(mutex);

    // remainder section
} while (TRUE);
```

Figure 6.9 Mutual-exclusion implementation with semaphores.

Usage

- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances.
- The semaphore is initialized to the number of resources available.
- Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count).
- When a process releases a resource, it performs a signal() operation (incrementing the count).
- When the count for the semaphore goes to 0, all resources are being used.
- After that, processes that wish to use a resource will block until the count becomes greater than 0.

Usage

- We can also use semaphores to solve various synchronization problems.
- For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2 .
- Suppose we require that S2 be executed only after S1 has completed.
- We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements

```
S1;
signal(synch) ;
```

in process P1 and the statements

```
wait(synch);
S2;
```

in process P2.

- Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch), which is after statement S1 has been executed.

Implementation

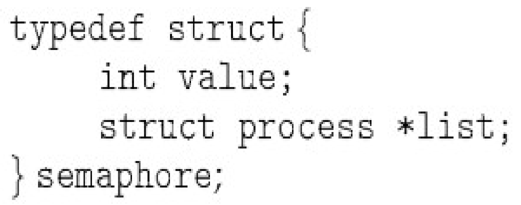
- The main disadvantage of the semaphore is that it requires busy waiting.
- While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is a problem in a multiprogramming system, where a single CPU is shared among many processes.
- Busy waiting wastes CPU cycles that some other process might be able to use productively.
- This type of semaphore is also called a spinlock, because the process "spins" while waiting for the lock.
- (Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful; they are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.)

Implementation

- To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.
- When a process executes the wait () operation and finds that the semaphore value is not positive, it must wait.
- However, rather than engaging in busy waiting, the process can block itself.
- The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.
- Then control is transferred to the CPU scheduler, which selects another process to execute.
- A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation.
- The process is restarted by a wakeup () operation, which changes the process from the waiting state to the ready state.
- The process is then placed in the ready queue.

Implementation

- To implement semaphores under this definition, we define a semaphore as a "C' struct:

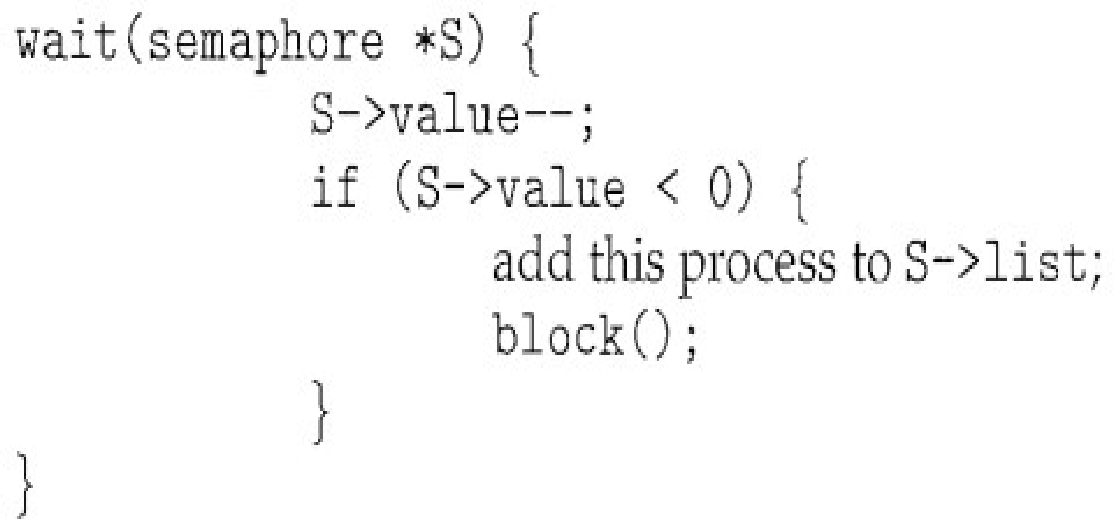


```
typedef struct {
      int value;
      struct process *list;
} semaphore;
```

- Each semaphore has an integer value and a list of processes list.
- When a process must wait on a semaphore, it is added to the list of processes.
- A signal() operation removes one process from the list of waiting processes and awakens that process.

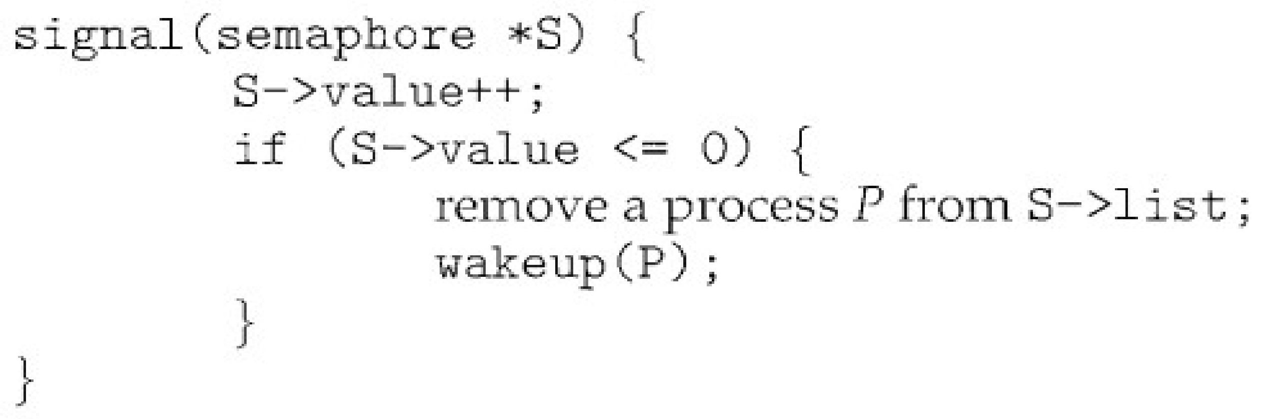
Implementation

- The wait() semaphore operation can now be defined as



```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}
```

- The signal efined as

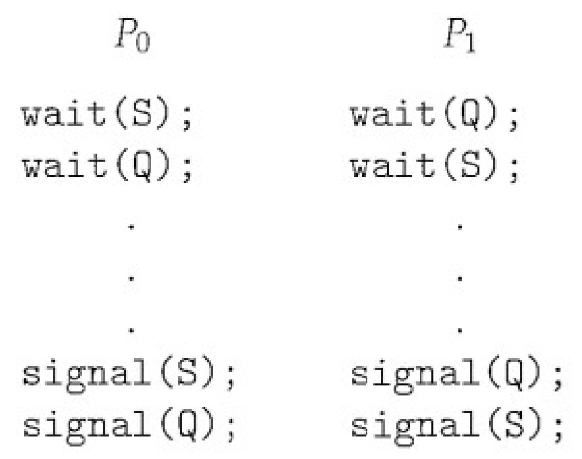


```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

- The block() opera
- The wakeup(P) o l process P.
- These two opera as basic system calls.

Deadlocks and Starvation

- The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.
- To illustrate this consider a system consisting of two processes, Po an sing two semaphores, S and Q, set to the



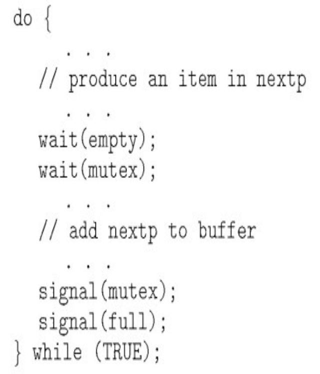
```
       P0                P1

wait(S);           wait(Q);
wait(Q);           wait(S);
   .                  .
   .                  .
   .                  .
signal(S);         signal(Q);
signal(Q);         signal(S);
```

Deadlocks and Starvation

- Suppose that Po executes wait (S) and then P1 executes wait (Q).
- When Po executes wait (Q), it must wait until P1 executes signal (Q).
- Similarly, when P1 executes wait (S), it must wait until Po executes signal(S).
- Since these signal() operations cannot be executed, Po and P1 are deadlocked.
- We say that a set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
- The events with which we are mainly concerned here are resource acquisition and release.
- Another problem related to deadlocks is or a situation in which processes wait indefinitely within the semaphore.
- Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

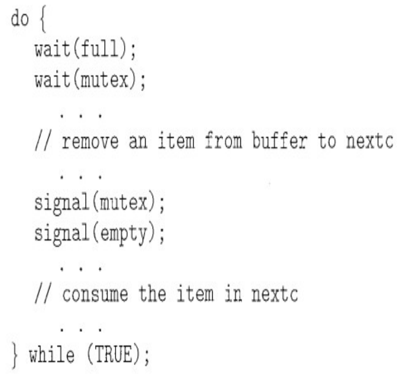
Classical Problems of Synchronization - The Bounded-Buffer Problem

- In bounded-buffer problem, assume that the pool consists of n buffers, each capable of holding one item.
- The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers.
- The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
- The code for the producer and consumer process is shown in Figure 6.10, Fig.6.11;
- We can interpret this code as the producer producing full buffers for th or as the consumer npty buffers for the pr



```
do {
    . . .
  // produce an item in nextp
    . . .
  wait(empty);
  wait(mutex);
    . . .
  // add nextp to buffer
    . . .
  signal(mutex);
  signal(full);
} while (TRUE);
```

Figure 6.10 The structure of the producer process.



```
do {
  wait(full);
  wait(mutex);
    . . .
  // remove an item from buffer to nextc
    . . .
  signal(mutex);
  signal(empty);
    . . .
  // consume the item in nextc
    . . .
} while (TRUE);
```

Figure 6.11 The structure of the consumer process.

Classical Problems of Synchronization – Readers Writers Problem

- Suppose that a database is to be shared among several concurrent processes.
- Some of these processes may want only to read the database, whereas others may want to update (that is, to read and write) the database.
- We distinguish between these two types of processes by referring to the former as readers and to the latter as writers.
- Obviously, if two readers access the shared data simultaneously, no adverse effects will result.
- However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.
- To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database.

Classical Problems of Synchronization – Readers Writers Problem

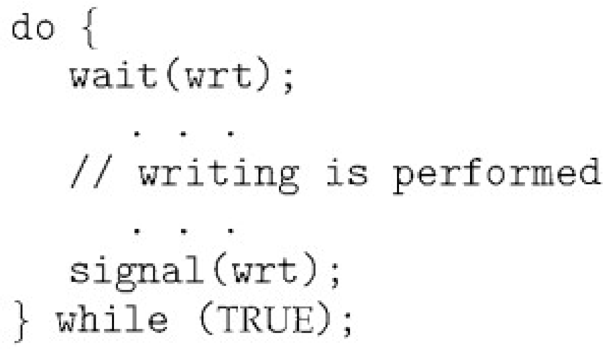
- The first readers-writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object.
- In other words, no reader should wait for other readers to finish simply because a writer is waiting.
- The second readers writers problem requires that, once a writer is ready, that writer performs its write as soon as possible.
- In other words, if a writer is waiting to access the object, no new readers may start reading.
- A solution to either problem may result in starvation.
- In the first case, writers may starve; in the second case, readers may starve.

Classical Problems of Synchronization – Readers Writers Problem

- In the solution to the first readers-writers problem, the reader processes share the following data structures:
  semaphore mutex, wrt; int readcount;
- The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0.
- The semaphore wrt is common to both reader and writer processes.
- The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated.
- The readcount variable keeps track of how many processes are currently reading the object.
- The semaphore wrt functions as a mutual-exclusion semaphore for the writers.
- It is also used by the first or last reader that enters or exits the critical section.
- It is not used by readers who enter or exit while other readers are in their critical sections.

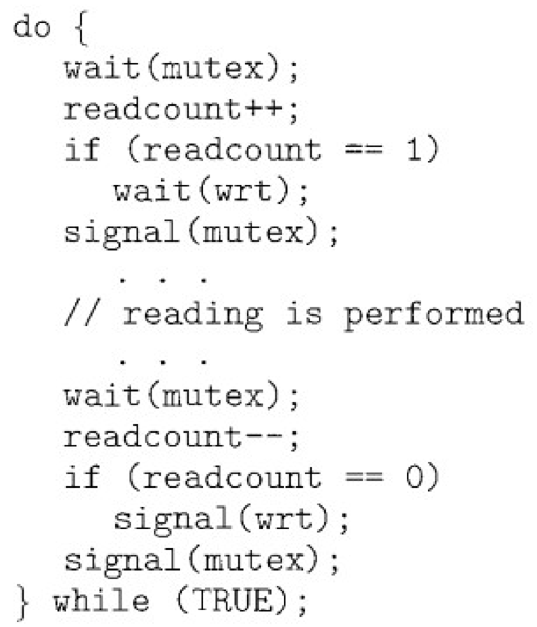
Classical Problems of Synchronization – Readers Writers Problem

- The code for a writer process is shown in Figure 6.12; the code for a reader process is shown in Figure 6.13.
- Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n- 1 readers are queued on mutex.



```
do {
   wait(wrt);
      . . .
   // writing is performed
      . . .
   signal(wrt);
} while (TRUE);
```

Figure 6.12 The structure of a writer process.



```
do {
   wait(mutex);
   readcount++;
   if (readcount == 1)
      wait(wrt);
   signal(mutex);
      . . .
   // reading is performed
      . . .
   wait(mutex);
   readcount--;
   if (readcount == 0)
      signal(wrt);
   signal(mutex);
} while (TRUE);
```

Figure 6.13 The structure of a reader process.

Classical Problems of Synchronization – Dining Philosopher Problem

- Consider five philosophers who spend their lives thinking and eating.
- The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher.
- In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.
- When a philosopher thinks, she does not interact with her colleagues.
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).
- A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor.
- When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
- When she is finished eating, she puts down both of her chopsticks

Classical Problems of Synchronization – Dining Philosopher Problem

- ❖ The dining-philosophers problem is considered a classic synchronization problem and it is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.
- ❖ One simple solution is to represent each chopstick with a semaphore.
- ❖ A philosopher tries to grab a chopstick by executing a wait () operation on that semaphore; she releases her chopsticks by executing the signal() operation on the appropriate semaphores.
- ❖ Thus, the shared data are

```
semaphore chopstick[5];
```

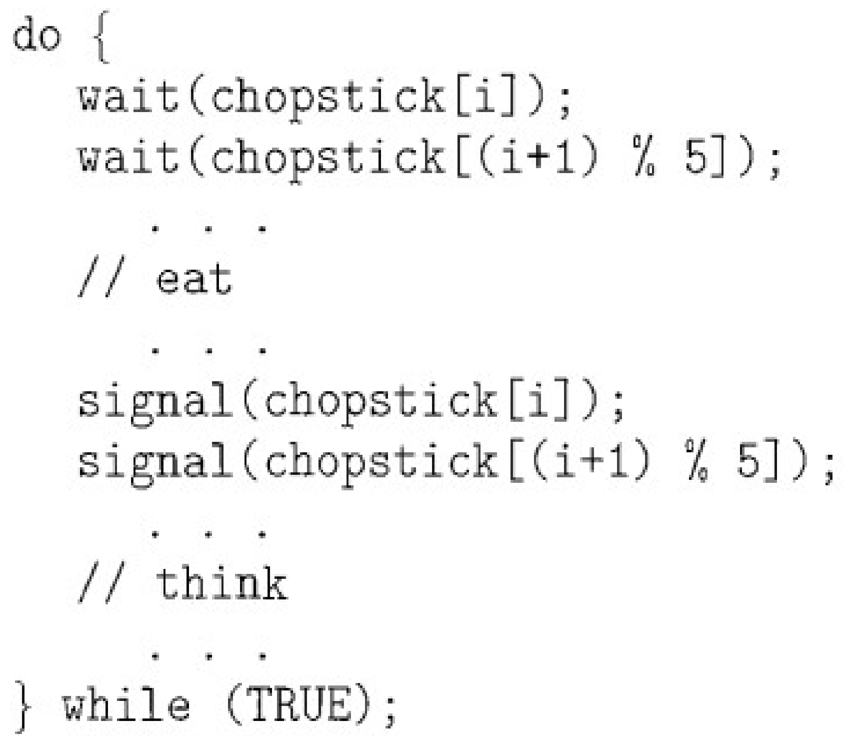
where all the elements of chopstick are initialized to 1.

Classical Problems of Synchronization – Dining Philosopher Problem

- The structure of philosopher i is shown in Figure 6.15.
- Although this solution guarantees that no two neighbors are eating simultaneously, it must be rejected because it could create a deadlock.
- Suppose that all five philosophers become hungry simultaneously and each grabs her left chopstick.
- All the elements of chopstick will now be equal to 0.
- When each philosopher tries to grab her right chopstick, she will be delayed forever.
- Several possible remedies to the deadlock problem are listed next.
  1. Allow at most four philosophers to be sitting simultaneously at the table.
  2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
  3. Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick

Classical Problems of Synchronization – Dining Philosopher Problem

- The structure of philosopher i is shown in Figure 6.15.



```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1) % 5]);
        . . .
    // eat
        . . .
    signal(chopstick[i]);
    signal(chopstick[(i+1) % 5]);
        . . .
    // think
        . . .
} while (TRUE);
```

Figure 6.15 The structure of philosopher *i*.

Monitors

Although semaphores provide an effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if some wrong execution sequences take place.

All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait (mutex) before entering the critical section and signal (mutex) afterward.

If this sequence is not observed, two processes may be in their critical sections simultaneously.

Monitors

- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
    ...
      critical section
    ...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be discovered only if several processes are simultaneously active in their critical sections.

- Suppose that a process replaces signal (mutex) with wait (mutex). That is, it executes:

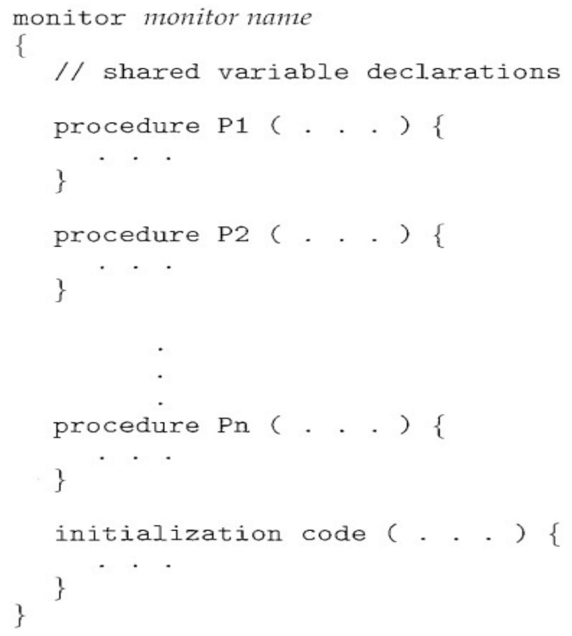
```
wait(mutex);
    ...
      critical section
    ...
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait (mutex), or the signal (mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

Monitors

- A monitor type is an ADT which presents a set of programmer-defined operations that are provided mutual exclusion within the monitor. A abstract data type- or ADT- encapsulates private data with public methods to operate on that data.
- The monitor type also contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.
- The syntax of a monitor type is shown in Figure 6.16.
- The representation of a monitor type cannot be used directly by the various processes.
- Thus, a procedure defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.



```
monitor monitor name
{
   // shared variable declarations

   procedure P1 ( . . . ) {
      . . .
   }

   procedure P2 ( . . . ) {
      . . .
   }
          .
          .
          .
   procedure Pn ( . . . ) {
      . . .
   }

   initialization code ( . . . ) {
      . . .
   }
}
```

Figure 6.16 Syntax of a monitor.

Monitors

- ❖The monitor construct ensures that only one process at a time is active within the monitor.
- ❖Consequently, the programmer does not need to code this synchronization constraint explicitly (F

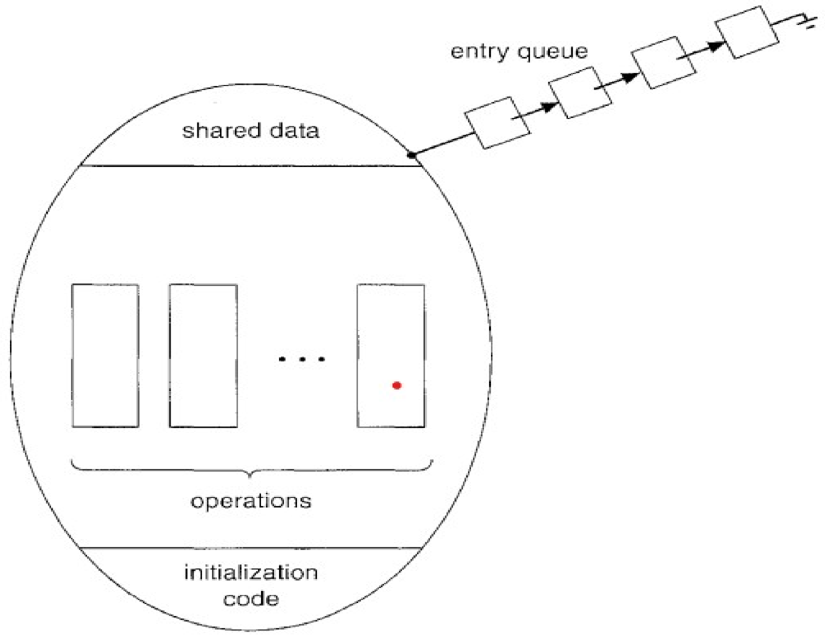


Figure 6.17 Schematic view of a monitor.

- ❖However the monitor construct, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms.
- ❖These mechanisms are provided by the condition construct

Monitors

❖A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type condition:

condition x, y;

The only operations that can be invoked on a condition variable are wait () and signal(). The operation

x. wait();

means that the process invoking this operation is suspended until another process invokes

x. signal();

The x. signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never b [illegible] 5.18). Contrast this operation with the signal() operation a [illegible] ores, which always affects the state of the semaphore.

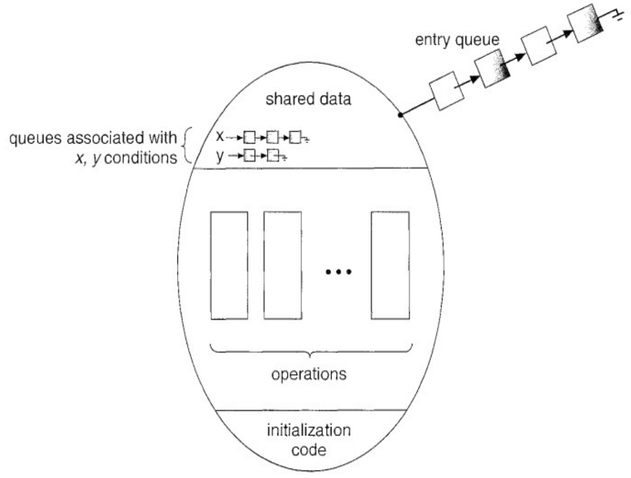


Figure 6.18 Monitor with condition variables.

Monitors

- ❖Now suppose that, when the x. signal () operation is invoked by a process P, there exists a suspended process Q associated with condition x.
- ❖Clearly, if the suspended process Q is allowed to resume its execution, the signaling process P must wait.
- ❖Otherwise, both P and Q would be active simultaneously within the monitor. Note, however, that both processes can conceptually continue with their execution.
- ❖Two possibilities exist:

1. Signal and wait: P either waits until Q leaves the monitor or waits for another condition.
2. Signal and continue: Q either waits until P leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option.

On the one hand, since P was already executing in the monitor, the signal-and-continue method seems more reasonable.

Dining Philosopher Solution using Monitors

The solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available.

To code this solution, we need to distinguish among three states in which we may find a philosopher.

For this purpose, we introduce the following data structure:

enum{THINKING, HUNGRY, EATING}state[5];

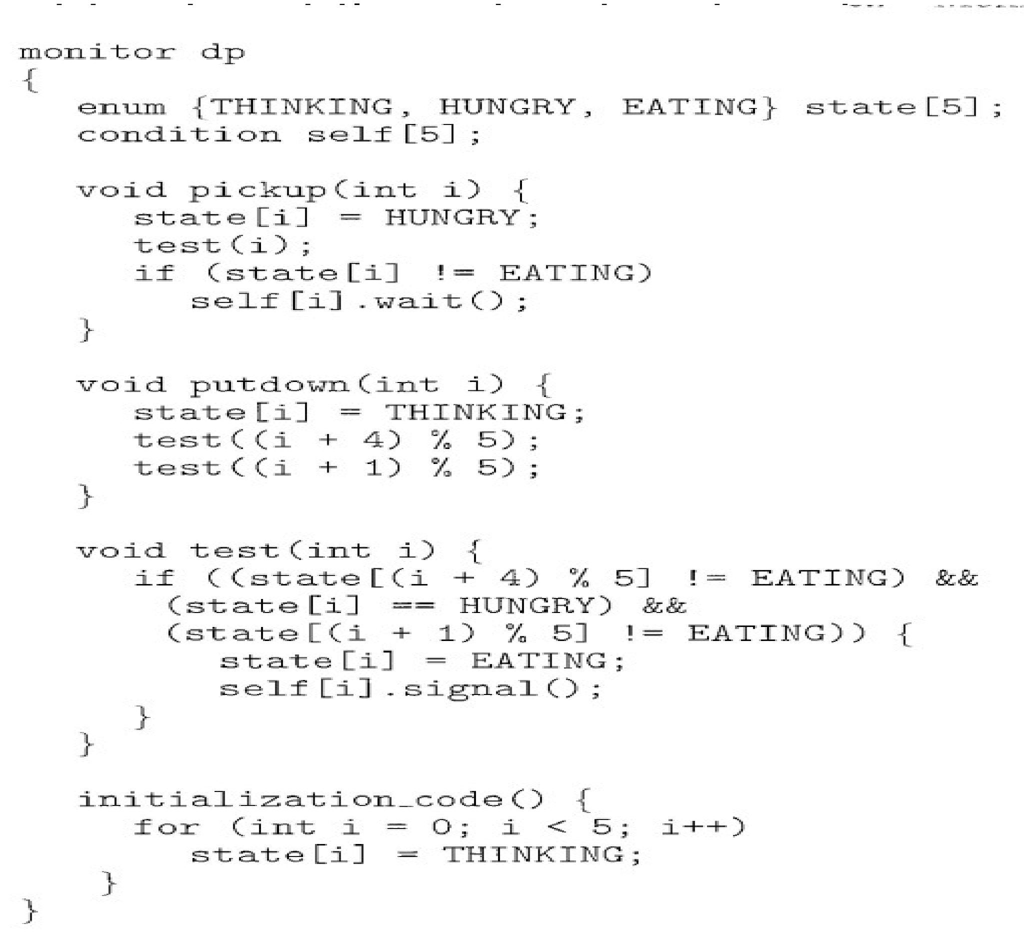
Philosopher i can set the variable state [i] = EATING only if her two neighbors are not eating: (state [(i +4) % 5] ! = EATING) and (state [(i +1) % 5] '= EATING).

We also need to declare condition self[5];

in which philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs.

Dining Philosopher Solution using Monitors

The distribution of the chopsticks is controlled by the monitor and the Dining Philosophers definition is shown in Figure 6.19. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Follow wn() operation.



```
monitor dp
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING)) {
           state[i] = EATING;
           self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Figure 6.19 A monitor solution to the dining-philosopher problem.

Dining Philosopher Solution using Monitors

Thus, philosopher i must invoke the operations pickup() and put down() in the following sequence:

```
DiningPhilosophers.pickup(i);
eat
DiningPhilosophers.putdown(i);
```

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur.

Implementing a Monitor Using Semaphores

For each Monitor, a semaphore mutex (initialized to 1) is provided.

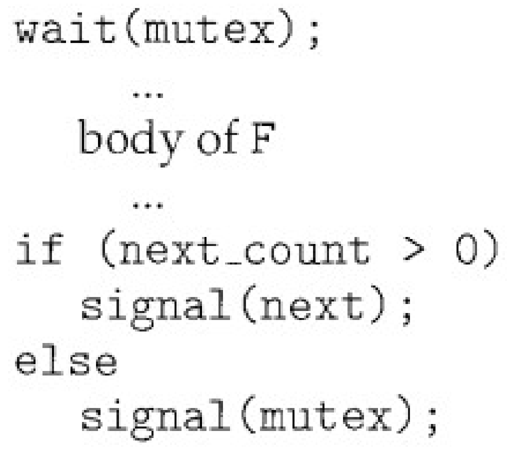
A process must execute wait (mutex) before entering the monitor and must execute signal (mutex) after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0.

The signaling processes can use next to suspend themselves.

An integer variable next_count is also provided to count the number of processes suspended on next.

Thus, each external procedure F is replaced by



```
wait(mutex);
     ...
   body of F
     ...
if (next_count > 0)
   signal(next);
else
   signal(mutex);
```

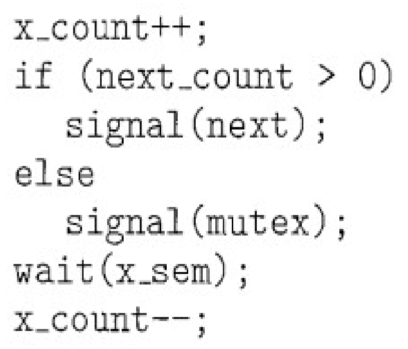
Mutual exclusion within a monitor is ensured.

Implementing a Monitor Using Semaphores

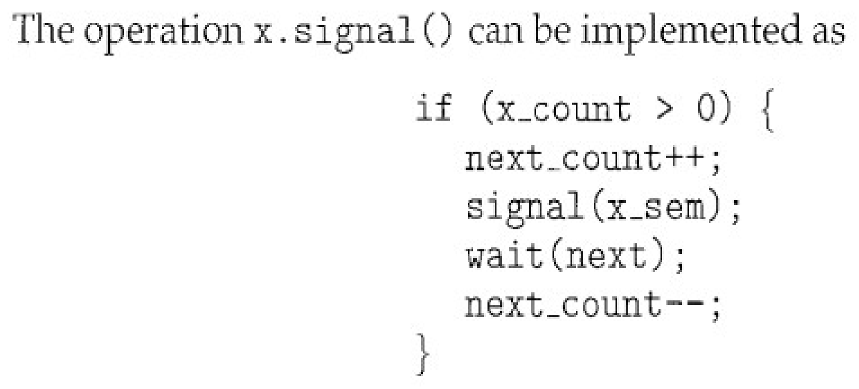
We can now describe how condition variables are implemented as well.

For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0.

The operation x. wait() can now be implemented as



```
x_count++;
if (next_count > 0)
   signal(next);
else
   signal(mutex);
wait(x_sem);
x_count--;
```



The operation x.signal() can be implemented as

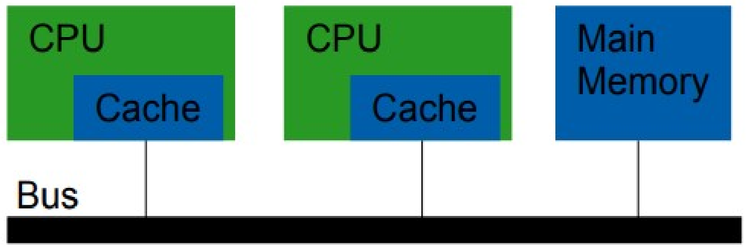
```
if (x_count > 0) {
   next_count++;
   signal(x_sem);
   wait(next);
   next_count--;
}
```

Mutual exclusion within a monitor is ensured.

Multiprocessors and Locking

Types of Multiprocessors (MPs):

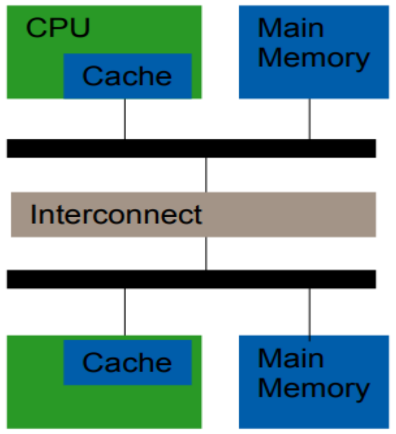
Uniform memory-access (UMA) MP



- Access to all memory occurs at the same speed for all processors.

Non-uniform memory-access (NUMA) MP

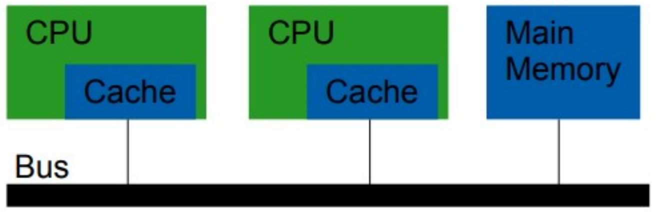
- Access to some parts of memory is faster for some processors than other parts of memory



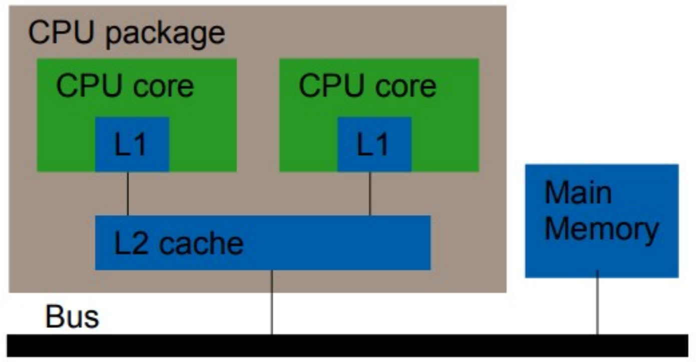
Multiprocessors and Locking

Types of UMA Multiprocessors:

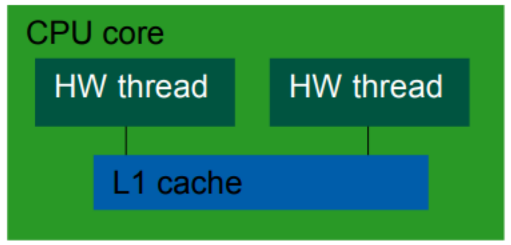
1. Classical multiprocessor
   - CPUs with local caches
   - connected by bus
   - fully separated cache hierarchy ⇒ cache coherency issues



2. Chip Multiprocessor (CMP)
   - per-core L1 caches
   - shared lower on-chip caches
   - usually called “multicore”
   - mild cache coherency issues – easily addressed ir



3. Symmetric multithreading (SMT)
   - replicated functional units, register state
   - interleaved execution of several threads
   - fully shared cache hierarchy
   - no cache coherency issues



Multiprocessors and Locking

Cache Coherency:

What happens if one CPU writes to (cached) address and another CPU reads from the same address

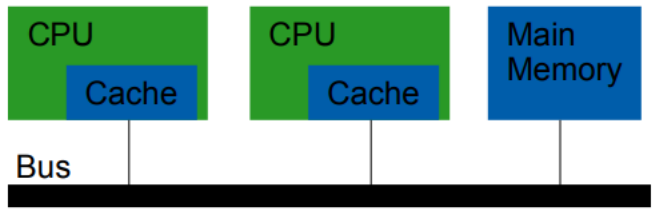
- Can be thought of as replication and migration of data between CPUs

Ideally, a read produces the result of the last write to the particular memory location

- Approaches that avoid the issue in software also prevent exploiting replication for parallelism
- Typically, a hardware solution is used:

Snooping – typically for bus-based architectures

Directory based – typically for non-bus interconnects



Multiprocessors and Locking

Snooping:

Each cache "broadcasts" transactions on the bus

Each cache monitors the bus for transactions that affect its state

Typically use "MESI" protocol in bus-based architectures

Each cache line is in one of four states:

Multiprocessors and Locking

Modified (M):

- The line is valid in the cache and in only this cache.
- The line is modified with respect to system memory—that is, the modified data in the line has not been written back to memory.

Exclusive (E):

- The addressed line is in this cache only.
- The data in this line is consistent with system memory.

Shared (S):

- The addressed line is valid in the cache and in at least one other cache.
- A shared line is always consistent with system memory. That is, the shared state is shared-unmodified; there is no shared-modified state.

Invalid (I):

- This state indicates that the addressed line is not resident in the cache and/ or any data contained is considered not useful.

Multiprocessors and Locking

Directory Based Coherence:

Each memory block has a home node

Home node keeps directory of cache that have a copy

• E.g., a bitmap of processors per memory block ✔ Invalidation/update messages can be directed explicitly ✗ Requires more storage to keep directory

• E.g. each 256 bits or memory requires 32 bits of directory

Scalable Locks

- test and set (t_s_acquire)
- The "test and set" (t_s_acquire) operation is a key component in implementing scalable locks. It is a synchronization primitive that allows a thread to atomically test and set a memory location to control access to a shared resource.
- This operation is commonly used in lock algorithms to ensure mutual exclusion and prevent multiple threads from accessing the shared resource simultaneously.

Scalable Locks

When a thread wants to acquire a lock using the test and set operation, it performs the following steps:

- It repeatedly executes the test and set instruction, which atomically checks the value of a memory location and sets it to a specified value. The exact implementation of the test and set instruction may vary depending on the architecture and programming language being used.

- The thread continues to loop until the test and set operation returns a specific value that indicates successful acquisition of the lock. This value could be a specific flag or a boolean indicating that the lock is now held by the thread.

Scalable Locks

- Test-and-Test-and-Set (T_T_S_Acquire) is a synchronization primitive used in concurrent programming to implement locks and ensure mutual exclusion. It is an extension of the Test-and-Set (T_S_Acquire) operation.
- The Test-and-Set operation involves atomically testing the value of a shared memory location and setting it to a specified value. This operation is typically used to acquire a lock by setting a flag or boolean variable to indicate that the lock is held.
- The thread first performs a non-atomic test operation on the lock variable. It reads the value of the lock variable without modifying it.
- If the test operation indicates that the lock is not currently held (i.e., the lock variable is unset or 0), the thread proceeds to perform the Test-and-Set operation atomically, setting the lock variable to indicate that it has acquired the lock.
- If the test operation indicates that the lock is held (i.e., the lock variable is set or 1), the thread enters a spin-wait loop, repeatedly performing the test operation until it observes that the lock is released. This spin-waiting allows other threads to make progress while the current thread waits for the lock to become available.