Synchronization

* A **cooperating process** is one that can affect or be affected by other processes executing in the system.
* A situation like this, 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 takes place, is called a **race condition**.
* To guard against the race condition above, we need to ensure that **only one process** at a time can be manipulating the variable counter.
* We require that the processes be synchronized in some way.

6.2 The Critical-Section Problem

Consider a system consisting of n processes {Po, P1, ... , pn-1· Each process has a segment of code, called a **critical section**,

* ***In critical section:*** the process may be changing **common variables, updating a table, writing a file, and so on.**
* When one process is executing in its critical section, no other process is allowed to execute in its critical section.
* **The critical-section problem** is to design a **protocol** that the processes can use to cooperate.

Critical section

Entry section

* Each process must request permission to enter its **critical section**.
* 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**.

Exit section

A **solution** to the critical-section problem requirement:

Remainder

1. Mutual exclusion: no 2 processes are executing in their critical section together.
2. **Progress**: If **no** process is executing in its critical section **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.**

**Possible race conditions in OS:**

1-a kernel data structure that **maintains** a list of all open files in the system.

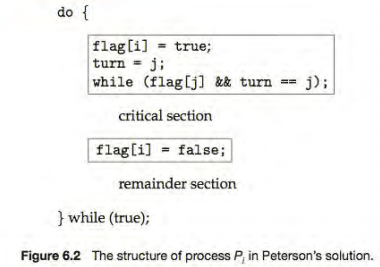
2-structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.

Two general **approaches** are used to handle critical sections in OS:

|  |  |
| --- | --- |
| preemptive kernels | nonpreemptive kernels |
| Allows a process to be preempted while it is running in kernel mode. | nonpreemptive kernel does not allow a process running in kernel mode to be preempted |
| Cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. | Essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. |
| difficult to design for SMP architectures,  Because it is possible for two kernel-mode processes to run simultaneously on different processors. |  |
| More favorable:   1. more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing (تخلي)the processor to waiting processes. 2. More suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel. | Less favorable |

6.3 Peterson’s Solution

**Classic software-based** solution to the critical-section.



* Peterson's solution is **restricted** to two processes that **alternate** **execution** between their critical sections and remainder sections.
* 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 **flag array** is used to indicate if a process is ready to enter its critical section.

* For example, if flag [i] is true, this value indicates that Pi is ready to enter its critical section.

6.4 Synchronization Hardware

* **Software-based solutions** such as Peterson's are not guaranteed to work on modem computer architectures.
* Another solutions (ranging from hardware to software) are based on the premise of **locking**- that is, protecting critical regions through the use of locks (sophisticated).
* The critical-section problem could be solved **simply** in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.
* This approach taken by **nonpreemptive** kernels.
* Not **feasible in multiprocessor** environment because it:
* 1- can be time consuming,2- has an effect on a system's clock.

Many modern computer systems therefore provide special hardware instructions to solve ***locking*** problem.

* Allow us either to test and modify the content of a word or to swap the contents of two words **atomically**- that is, as one uninterruptible unit.
* Ex: The test-and-set() instruction is executed atomically.
* For explanation: look textbookp260.

**6.5 Mutex Locks**

Instead of hardware-based solutions, operating-systems designers build software tools to solve the critical-section problem **mutex lock(short for mutual exclusion).**

* Used to protect critical regions and thus prevent race conditions.
* A process must acquire (acquire ()) the lock before entering a critical section; it releases (release ()) the lock when it exits the critical section.
* A mutex lock has a **Boolean** variable **available**
* A process that attempts to acquire an unavailable lock **is blocked** until the lock is released.
* The main disadvantage of the implementation given here is that it requires **busy waiting**.
* **Spinlock**: While a process is in its critical section, any other process that tries to enter its critical section must **loop continuously.**
* In fact, this type of mutex lock called a **spinlock** because the process "**spins**" while waiting for the lock to become available.
* Advantage: no context switch is required when a process must wait on a lock, and a context switch may take considerable time- employed on multiprocessor systems

**6.6 Semaphores**

**Mutex** locks is the simplest of synchronization tools.

A more robust tool is **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait () and signal().

* The wait () operation called **P** signal () was originally called **V**
* All modifications to semaphore executed **indivisibly**:

1-No permission is given to two processes to **modify** the same semaphore value in one time.

2-The integer value and its possible modification, must be executed **without interruption.**

**6.6.1 Semaphore Usage**

Operating systems often distinguish between **counting** and **binary** semaphores.

* **Counting semaphore:** has a value can range over an unrestricted domain.
* **Binary semaphore** value can range only between 0 and 1.

On systems that do not provide mutex locks, binary semaphores can be used **instead** for providing mutual exclusion (similar behavior).

Counting semaphores can be used **to control access to a given resource** consisting of a finite number of instances.

Process TO use a resource performs a wait (), TO releases a resource it performs a signal ().

We can also use semaphores to solve various synchronization problems.

**6.6.2 Semaphore Implementation**

**Mutex locks** discussed in Section 6.5 **suffers** from busy waiting as well as wait () and signal ().

We can modify the definition of the wait () and signal ().

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 with a “waiting state”.
* Then, CPU scheduler selects another process to execute.
* Then the process should be restarted (wakeup ()) when some other process executes a signal() operation” ready state”.
* The process is then placed in the ready queue.

The **block ()** operation **suspends** the process that invokes it. The **wakeup (P**) operation **resumes** the execution of a blocked process P. These two operations are provided by the OS as basic system calls.

The list of waiting processes can be easily implemented by a **link field** in each process control block (PCB).

* Each semaphore contains an **integer value** and a **pointer** to a list of PCBs.
* One way to add and remove processes from the list is to use a FIFO queue, where the semaphore contains both **head** and **tail** pointers to the queue.

**6.6.3 Deadlocks and Starvation**

* A set of processes is in a **deadlocked** **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 **indefinite** blocking or **starvation**, 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.

**6.6.4 Priority Inversion**

A **scheduling challenge** happened when a **higher-priority** process needs to read or modify kernel data that are currently accessed by (one/chain) of **lower-priority** processes.

* Since kernel data are protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.
* More complicated Situation if the lower-priority process is preempted in favor of another process with a higher priority.
* This problem is known as **priority inversion**. It occurs only in systems with more than two priorities, so one solution is to have only two priorities(insufficient solution).
* They solve the problem by implementing:

**Priority-inheritance protocol:** all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

* When they are finished, their priorities revert to their original values.

**6.7 Classic Problems of Synchronization page269**

**6.8 Monitors**

* Although semaphores provide a **convenient** and **effective** mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect.

**Types of errors** by using semaphores incorrectly to solve the critical-section problem:

* a process interchanges the order in which the wait() and signal () operations on the semaphore mutex are executed:

Signal (mutex);

critical section

Wait (mutex) ;

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement.

* a process **replaces** signal (mutex) with wait (mutex). That is, it executes

wait (mutex);

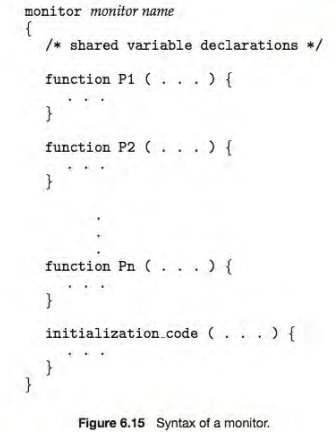
critical section

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.

To deal with such errors, we describe one fundamental high-level synchronization construct-the monitor type.

**6.8.1 Monitor Usage**

* **An abstract data type-or ADT**-encapsulates data with a set of functions to

Operate on that data independently of any specific implementation of the ADT.

* A **monitor type** is an ADT that includes a set of programmer-defined

operations that are provided with mutual exclusion within the monitor.

The representation of a monitor type cannot be used directly.

Thus, a function defined within a monitor can **access only those variables declared**

**Locally within the monitor.**

* Similarly, the local variables of a monitor can be accessed by only the local functions.
* The monitor construct ensures that only **one** process at a time is active within the monitor.
* 6.8.2 Dining-Philosophers Solution Using Monitors.p276-277

**6.8.3 Implementing a Monitor Using Semaphores**

We now consider a possible implementation of the monitor mechanism using semaphores.

* For each monitor,

1. A semaphore mutex (**initialized** to 1).
2. A process must **execute** wait (mutex) before entering the monitor.
3. **Execute** signal (mutex) after leaving the monitor.
4. Since a signaling process must wait until the resumed process either leaves or waits, next additional semaphore is introduced, initialized to 0.
5. Processes can use next to suspend themselves/**next\_count** is to count the number of processes suspended.

**6.8.4 Resuming Processes within a Monitor**

How do we determine which of the suspended processes should be resumed next?

One simple solution is **to use a first-come, first-served (FCFS) ordering**, so that the process that has been waiting the longest is resumed first.(not used)

The conditional-wait construct can be used. This construct has the form

**x.wait(c);**

C is an integer expression that is evaluated when the wait () operation is executed.

* The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended.
* When x. signal () is executed, the process with the **smallest priority** number is resumed next.
* **To illustrate this new mechanism,** consider the ResourceAllocator monitor:
* Each process, when requesting an allocation of this resource,

1-specifies the maximum time it plans to use the resource.

2-The monitor allocates the resource to the process that has the shortest time-allocation request.

3-A process that needs to access the resource in question must observe

the following sequence:

R.acquire(t);

access the resource;

R. release () ;

**Where R is an instance of type ResourceAllocator**.

**Unfortunately**, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

• A process might **access** a resource without first gaining access permission to the resource.

• A process might **never release** a resource once it has been granted access to the resource.

• A process might **attempt to release** a resource that it never requested.

• A process might **request the same resource** twice (without first releasing the resource).

* One possible solution to the current problem is to include the resourceaccess operations within the ResourceAllocator monitor.

**6.9 Synchronization Examplesp283-287**

**6.10 Alternative Approaches**

In this section, we explore various features provided in both programming languages and hardware that support designing thread-safe concurrent applications.

**6.10.1 Transactional Memory**

Ideas from one area of study can be used to solve problems in other areas.

The concept of **transactional memory** originated in database theory, for example, yet it provides a strategy for process synchronization.

* **A memory transaction** is a sequence of memory read-write operations that are atomic.
* If all operations in a transaction are **completed**, the memory transaction is **committed**.
* **Otherwise**, the operations must be **aborted** and **rolled back**.
* The benefits of transactional memory can be obtained through features added to a programming language:
* As an alternative to traditional locking methods, new features that take advantage of transactional memory can be added to a programming language.
* **The advantages of using such a mechanism rather than locks :**

1. The transactional memory system- not the developer- is responsible for guaranteeing atomicity.
2. Because no locks are involved, deadlock is not possible.
3. A transactional memory system can easily identify which statements in atomic blocks can be executed concurrently, such as concurrent read access to a shared variable.

* **Software transactional memory (STM),** exclusively in **software**-no special hardware is needed.
* STM works by inserting instrumentation code by compiler inside transaction blocks and manages each transaction by:
* By examining where statements may run concurrently and where specific low-level locking is required.
* **Hardware transactional memory (HTM)** uses hardware cache hierarchies and cache coherency protocols to manage and resolve conflicts involving
* Shared data residing in separate processors' caches.
* HTM requires no special code instrumentation and thus has less overhead than STM.
* HTM does require that existing cache hierarchies and cache coherency protocols be modified to support transactional memory.

**6.10.2 OpenMP**

**OpenMP includes** a set of compiler directives and an API.

* Any code following the compiler directive **#pragma omp parallel** is 1- identified as a parallel region 2- performed by a number of threads equal to the number of processing cores in the system.
* The critical-section compiler directive behaves much like a binary semaphore or mutex lock, ensuring that only one thread at a time is active in the critical section.
* An **advantage** of using the critical-section compiler directive in OpenMP is that it is generally considered easier to use than standard mutex locks.
* **Disadvantage** is that still identify possible race conditions and deadlocks.

**6.10.3 Functional Programming Languages**

Most well-known programming languages-such as C, C++, Java, and C#-are known as **imperative** (or **procedural**) languages.

* **Imperative languages are used for implementing algorithms that are state-based.**

With the current emphasis on concurrent and parallel programming for multicore systems, there has been greater focus on :

* **Functional programming languages**, which follow a programming paradigm much different from that offered by **imperative** languages.

The fundamental difference between imperative and functional languages is that **functional languages do not maintain state.** That is, once a variable has been defined and assigned a value, its value is **immutable**-it cannot change.