Operating Systems

Antonio Vivace - 2016 revision 4 – Licensed under GPLv3

Process Synchronization

Background

A cooperating process can share directly a logical address space (code, data) or share data through files/messages. We study mechanisms to ensure consistency of shared data.

Allowing parallell process to manipulate the same data concurrently can lead to incorrect states (**race condition**: particular access-order determines the outcome).

We need to **synchronize** and **coordinate** the processes in some way so we can assure that only one process at a time can access sensitive variables.

The Critical-Section Problem

Each process has a **critical section** in which it may be changing shared data. The system simply allows only one process to run its critical section at a time. The problem is to design a protocal that the process can use to cooperate: each one must request permission to run its critical section (**entry section** implements this). The critical section may be followed by an **exit section** and the remaining code is the **remaider section**.

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

1. **Mutual exclusion**. Only one critical section running at a time.
2. **Progress Selection**. of the process must choose only from the pool of the process not executing remainder sections. The selection cannot be postponed indefinitely.
3. **Bounded waiting**. There's a bound or a limit on number of times that other processes are allowed to execute their critical section after a process has requested to enter its critical section and before that request is granted.

Race conditions can affect kernels too, especially for particular data structures (list of open files, smemory allocation, process lists, interrupt handler).

Two approaches are used to handle the race conditions in operating systems: **preemptive** and **nonpreemptive** kernels.

A preemptive kernel allow a process running in kernel mode to be preempted while nonpreemptive doesn't: the process will run until it exits kernel mode, blocks or yelds control of the CPU.

Nonpreemptive kernels are free from race conditions while preemptive aren't.

Preemptive kernels are more suitable for real-time programming and they avoids the problem of having kernel-mode process running for long periods but they need to manage with the race conditions.

Peterson’s Solution

This implementation is a good algorithmic description of solving critical-section problem but may be not working on modern architectures because of how different they perform basic machine-language instructions.

This solution is restricted to 2 processes and coordinate their execution of critical-section.

It makes use of 2 shared data items : a variable turn that indicates whose turn it is to enter its critical section (if turn==i then P_i is allowed to execute its critical section). And flag[i] that stores if P_i is ready to enter its critical section.

This solution is consistent and we can verify that satisfies the requirements.

Synchronization Hardware

There are more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to both kernel developers and application programmers. These solutions are based on the premise of **locking**: protecting critical regions using locks (locks can have sophisticated designs).

```
boolean test_and_set(boolean
*target) {
boolean rv = *target;
*target = true;
return rv;
}
```

The `test_and_set()` is executed atomically (if two of them are executed simultaneously, on different CPU, they will be executed sequentially). We can implement the mutual exclusion simply adding a `boolean` variable lock, initialized to `false`.

```
do {
waiting[i] = true;
key = true;
while (waiting[i] && key)
key = test and set(&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);
boolean waiting[n];
boolean lock;
```

Bounded-waiting and mutual exclusion implementation with `test_and_set()`.

`Waiting[n]` and `lock` are initialized to false. P_i can enter its critical section only if either `waiting[i] == false` or `key == false`. Key can become false only if the `test_and_set()` is executed. The first P to execute `test_and_set()` will find `key==false`, all others must wait. Waiting[i] can become false only if another process leaves its critical section. Only one `waiting[i]` is set to false at a time (**mutual-exclusion**).

A process exiting its 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 (**progress selection**).

When a process leaves its critical section, it scans the array waiting in the cyclic ordering (starting from `i+1`), it then designates the first process in this list that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section (**bounded-waiting**).

Mutex Locks

Since the solutions presented in hardware synchronization are inaccessible to application programmers, OS designers build software tools to solve the critical-section problem: the simplest of these tools is the mutex lock (short for mutual exclusion).

```
acquire() {
while (!available)
; /* busy wait */
available = false;;
}

release() {
available = true;
}

Do {
Acquire Lock
       Critical section
Release Lock
       Remainder section
} while (true);
```

A process must acquire the lock before executing a critical section, it then releases the lock when it exits it. `acquire()` and `release()` functions are defined as follows (they must be executed atomically):

A process that attempts to acquire an unavailable lock is blocked until the lock is released (**busy waiting**).

This type of mutex lock is also called a **spinlock**. While these implementations are not suitable in multiprogramming systems, the spinlock is someway used in them when no context switch is required (and may take considerable time) and a process must wait on a lock: one thread can spin on one processor while another thread performs its critical section on another processor.

Semaphores

Semaphores are a more robust tool that can behave like mutex lock but can also provide more sophisticated ways for process to synchronize their activities.

A semaphore `S` is an integer variable that, apart from init, is accessed only through two standard atomic and indivisible operations: `wait()` and `signal()`. There are counting semaphores and binary semaphores. Semaphores can be used to control access to a given limited resource (consisting of a finite number of instances). It's initialized to the number of resources available and every process that wishes to use a resource performs a `wait()` decrementing the count. When it finishes and releases the resource it performs a `signal()` incrementing the count. Another simple use of the semaphore is "blocking" a process till another one finished execution.

```
wait(S) {
  while (S <= 0 )
    ; // busy wait
  S--;
}
signal(S) {
  S++;
}
```

Our definitions of `wait()` and `signal()` suffer from busy waiting in the same way mutex locks did, so we need to modify our implementation:

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

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

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

When a process executes `wait()` and finds that the semaphore values isn't positive, the process blocks itself, this operation places the process into a waiting queue associated with the semaphore and the state of the process is now Waiting. Control is transferred to the CPU scheduler which selects another process to execute. A blocked process can be restarted by a `wakeup()` operation (changes the process state from waiting to ready). The process is now in the Ready queue.

`Wait()` and `signal()` must be executed atomically (we can do this inhibiting interrupts, so instructions from different processes cannot be interleaved). One way to add and remove processes from the list ensuring bounded waiting is to use a FIFO queue (tough the correct usage of semaphores does not depend on particular queueing strategy for the list).

In a multiprocessor environment, interrupts must be disabled on every processor and this can be a difficult task and expensive. Therefore, in these cases, systems must provide alternative locking techniques, such as `compare_and_swap()` or spinlocks, to ensure that wait and `signal()` are performed atomically.

We did not remove the busy waiting but we have moved it from the entry section to the critical sections of programs. We have limited busy waiting to the critical sections of the `wait()` and `signal()` operations and these section are (if properly coded) short.

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 other waiting processes. The event in question is the execution of a `signal()` operation. When this point is reached, these processes are said to be **deadlocked**. Resource acquisition and release may lead to deadlocks.

Another problem related to deadlocks **is indefinite blocking** or **starvation**, when processes wait indefinitely within the semaphore (i.e. if we remove processes from the list associated with a semaphore in LIFO order).

Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data currently used by a lower-priority process (or a chain of them). The situation becomes more complicated if the lower-priority process is pre-empted in favour of another process with a higher priority.

This problem is known as **priority inversion** and occurs only in systems with more than two priorities. It is solved implementing a **priority-inheritance protocol**:

All process accessing resources needed by a higher-priority process inherit the higher priority until they are finished. When they are finished their priority revert to their original values (preventing other process from pre-empting its execution and lower-priority process from affecting wait of a higher one).

Classic Problems of Synchronization

The Bounded-Buffer Problem

The **problem** describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The **solution** for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be reached by means of inter-process communication, typically using **semaphores**.

The Readers-Writers Problem

The **readers-writers problems** are examples of a common computing problem in concurrency. There are at least three variations of the problems, which deal with situations in which many threads try to access the same shared resource at one time.

The simplest one, referred to as 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 perform 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.

The readers–writers problem and its solutions have been generalized to provide **reader–writer locks** on some systems. Acquiring a reader–writer lock requires specifying the mode of the lock: either read or write access. Reader–writer locks are most useful in the following situations:

1. In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
2. In applications that have more readers than writers. This is because reader–Writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

The Dining-Philosophers Problem

Five silent philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers.

Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.

Eating is not limited by the remaining amounts of spaghetti or stomach space; an infinite supply and an infinite demand are assumed.

```
semaphore chopstick[5];
do {
  wait(chopstick[i]);
  wait(chopstick[(i+1) % 5]);
  . . .
  /* eat for awhile */
  . . .
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
  . . .
  /* think for awhile */
  . . .
} while (true);
```

The problem is how to design a discipline of behaviour (a concurrent algorithm) such that no philosopher will starve; i.e., each can forever continue to alternate between eating and thinking, assuming that no philosopher can know when others may want to eat or think.

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.

Although this solution guarantees that no two neighbours are eating simultaneously, it nevertheless must be rejected because **it could create a deadlock**. Suppose that all five philosophers become hungry at the same time 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 replaced by:

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-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick

Monitors

A **monitor type** is an abstract data type (ADT) that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. This construct ensures that only one process at a time is active within the monitor and the programmer does not need to code the synchronization constraint explicitly. We define additional synch mechanisms to use with monitors: variables of type condition.

The only operations that can be invoked on a condition variable are wait and signal. Wait suspends the invoking processes and signal resumes one suspended process. Suppose that x.signal() is invoked by a process P, there exits a suspended process Q associated with the condition x. If Q is allowed to resume, the signalling P must wait, otherwise both P and Q would be active simultaneously within the monitor. However, 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.

Both options have reasonable arguments in their favour.

Synchronization Examples

Synchronization in Windows

When the Windows kernel accesses a global resource on a single-processor system, it temporarily masks interrupts for all interrupt handlers that may be also access the global resource.

On a multiprocessor system, Windows protects access to global resources using spinlocks. The kernel uses spinlocks only to protect short code segments and ensures that a thread will never be preempted while holding a spinlock.

For thread synchronization outside the kernel, Windows provides dispatcher objects: thread synchronize according to several mechanism, including mutex locks, semaphores, events and timers. The system protects shared data requiring a thread to gain ownership of a mutex to access data and to release ownership when it finished. Dispatcher objects may be in either signaled (the object is available and the thread will not block) or nonsignaled (the object is not available and the thread will block attempting to acquire it) state.

When a thread blocks on a nonsignaled dispatcher object, its state changes from ready to waiting, and the thread is placed in a waiting queue for that object. When the state for the dispatcher object moves to signalled, then kernel checks whether any threads are waiting on the object, if so, it moves one thread from waiting to ready, where they can resume executing.

We can use mutex lock as a representation of dispatcher objects and thread states: if a thread tries to acquire a mutex dispatcher object that is a nonsignaled state (not available) that thread will be suspended and placed in a waiting queue for the mutex object. When it moves to the signaled state (another thread has released the lock), the thread waiting at the front of the queue will be moved from the waiting state to ready state and it will finally acquire the mutex lock.

A critical-section object is a user-mode mutex that can often be acquired and released without kernel intervention. On multiprocessor system, a critical-section object first uses a spinlock while waiting for the other thread to release, if it spins too long, the acquiring thread will then allocate a kernel mutex and yield its CPU (this is efficient because the kernel mutex is allocated only when there is contention for the object, in practise there is very little).

Deadlocks

System Model

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances.

A process must request a resource before using it and must release the resource after using it and it can request as many resources to carry out its designated task. A process may utilize a resource in only the following sequence:

1. Request
2. Use
3. Release

Request and release of resources may be system calls. Operating system allocate resources and keeps tracks of queues and the state of every resource.

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.

Deadlocks may also involve different resource types.

Deadlock Characterization

A deadlock can arise if the following four **necessary conditions** hold simultaneously in a system:

1. **Mutual exclusion**. At least one resource must be held in a non-sharable mode.
2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional ones that are currently being held by other processes.
3. **No preemption.** Resources cannot be preempted, a resource can be released only voluntarily by the process holding it.
4. **Circular wait.** P_0 is waiting for a R held by P_1, P_1 is waiting for a R held by P_2,…

Methods for Handling Deadlocks

We can deal with deadlock problem in these ways:

1. Prevent or avoid deadlocks through a protocol, ensuring that the system will never enter a deadlocked state.
2. Allow the system to enter a deadlocked state, detect it, and recover.
3. Ignore the problem and pretend that deadlocks never occur in the system.

The third solution – adopted by Linux and Windows – moves the responsibility of handling deadlocks to the application developers.

Deadlock Prevention

By ensuring that at least one of the necessary conditions cannot hold, we can prevent the occurrence of a deadlock.

Mutual Exclusion

Sharable resources are not a problem, but non-sharable are: we cannot prevent deadlocks by denying mutual-exclusion, because some resources are intrinsically non-sharable.

Hold and Wait

In this case we must guarantee that, whenever a process requests a resource, it does not hold any other resources. We can use different protocols to do this, like requiring that system calls requesting resources for a process **precede all other system calls** or to allow a process to request resource only when it has **none** (first protocol has the disadvantage of low resource utilization, and the second may lead to starvation).

No preemption

We can solve this with the following protocols.
- If a process is holding some resources and requests another one that cannot be immediately allocated to it, all resources the process is holing are pre-empted (implicitly released). The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Alternatively, if a process requests some resources, we first check whether they available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for addition resources. If so, we pre-empt the desired resources from the waiting process and allocate them to the requesting process. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were pre-empted while it was waiting.

Circular Wait

One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests in an increasing order of enumeration. Developing an ordering/hierarchy of resources, does not in itself prevent deadlock, it is up to application developers write programs that actually **follow** the ordering. Sometimes, certain software can be used to verify that locks are acquired in the proper order and to give appropriate warnings when locks are acquired out of order.

Deadlock Avoidance

Deadlock-prevention algorithms, prevent deadlocks by regulating how requests can be made. However, this method can lead to low device utilization and reduced system throughput.

An alternative method for avoiding deadlocks is to require addition information about **how** resources are to be requested. Based on preliminary knowledge, the system can decide for each request if the process should wait in order to avoid a possible future deadlock.

The various algorithms that us this approach differ in the amount and type of information required. The simplest and most useful model requires that each process declare the **maximum number** of resources of each type that it may need. Given this information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.

The **resource-allocation state** is defined by the number of available and allocated resources and the maximum demands of the processes.

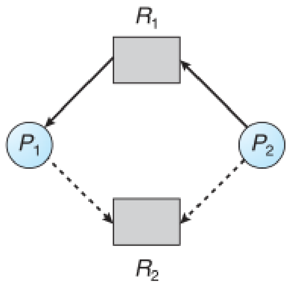
Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some older and still avoid a deadlock. A system is in a safe state only if there exists a **safe sequence**.

A sequence of processes <P_1,P_2,…P_n> is safe if, for each Pi, the resource requests that Pi can still make can be satisfied by the currently available resources plus the resources held by all P_j with $j < i$. In this situation, if the resources that Pi need are not immediately available, it can wait until all P_j have finished. When they have finished, Pi can obtain all of its needed resources, run, return its allocated resources and terminate. If no such sequence exists, the system state is said to be unsafe.

Given the concept of a safe state, we can define avoidance algorithms that ensure the system will never deadlock. The idea is to remain in the safe state: we start being in it and we grant requests to processes only if the allocation leaves the system in a safe state. Note that if a process requests a resource that is currently available, it may still have to wait. Thus, resource utilization may be lower than it would otherwise be.

Resource-Allocation-Graph Algorithm If we have a resource-allocation system with only one instance of each resource type, we can use a variant of the resource-allocation graph. We introduce a new type of edge, called a **claim edge**:



$P_i \rightarrow R_j$ indicates that process P_i may request resource R j at some time in the future (represented in the graph by a dashed line).

Now suppose that process P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph. We check for safety by using a **cycle-detection algorithm**. An algorithm for detecting a cycle in this graph requires an order of n 2 operations, where n is the number of processes in the system. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. In that case, process P_i will have to wait for its requests to be satisfied.

Banker's Algorithm Less efficient but applicable to a system with multiple instances of each resource type.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement this (*n* is the number of processes in the system and *m* is the number of resources types):

1. `Available[m]` Indicates the number of available resources of each type.
2. `Max[n][m]` Defines the maximum demand of each process.
3. `Allocation[n][m]` Defines the number of resources of each type currently located to each process.
4. `Need[n][m]` Indicates the remaining resource need of each process.