Tutoriat 4 (Operating Systems)
CPU Scheduling

Contents

1 Introduction

When two threads update the same data simultaneously in an uncontrolled manner, it can lead to incorrect results and data corruption. As an example, consider the variable *count* and the machine code implementations of the instructions **count++** and **count−−**:

count++

```
register = count
register = register + 1
count = register
```

count−−

```
register = count
register = register - 1
count = register
```

Intuitively, two threads that execute *count++* and *count−−* at the same time should not ultimately change the value of the variable *count*. However, the simultaneous execution of these operations will produce a random ordering of the machine code instructions; for example, consider *count = 5*:

```
register1 = count           [register1 = 5]
register1 = register1 + 1   [register1 = 6]
register2 = count           [register2 = 5]
count = register1           [count = 6]
register2 = register2 - 1   [register2 = 4]
count = register2           [count = 4]
```

As can be seen, the final result is wrong; the value of *count* is 4 instead of 5. Such a situation where multiple threads are executing operations on shared data and the final outcome is determined by the order in which the threads are manipulating the data is known as a **race condition**. Say that 100 threads execute the code below; what are the possible *minimum* and *maximum* values for the variable *count*?

```
for i in range(100):
    temp = count
    temp = temp + 1
    count = temp
```

A **critical section** is a segment of code in a multi-threaded application that performs operations on shared data. When two or more threads enter their critical sections at the same time, race conditions may occur; this is known as the **critical section problem**. There are various solutions for dealing with this issue; such a solution should meet the following three properties:

- **Mutual exclusion:** if a thread is currently executing in its critical section, then no other thread can execute in its critical section at that moment.
- **Progress:** If no thread is executing in its critical section and some threads wish to enter their critical sections, then only those threads that wish to enter their critical sections can participate in the decision of which thread will enter its critical section next.
- **Bounded waiting:** there is a limit on the number of times that other threads are allowed to enter their critical sections after a thread has made a request to enter its critical section and before that request is granted. Thus, no thread will wait indefinitely to enter its critical section.

2 Peterson's solution

This is a solution to the critical-section problem when only two threads T_i and T_j are involved. These threads share the following two variables:

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

The variable *turn* states which thread should execute in its critical section. If $turn == i$, then the thread T_i is allowed to enter its critical section. The *flag* array indicates whether a thread is ready or not to enter its critical section; if $flag[i] == true$, then T_i is ready. The code of these threads is as follows:

Thread T_i

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

    while (flag[j] && turn == j);

    /* critical section */

    flag[i] = false;

    /* remainder section */
}
```

Thread T_j

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

    while (flag[i] && turn == i);

    /* critical section */

    flag[j] = false;

    /* remainder section */
}
```

At first, both threads set $flag[i] = true$ and $flag[j] = true$, indicating that they are both ready to enter their critical sections. The next two instructions are $turn == i$ and $turn == j$; both assignments can be executed at the same time, but one of the values will be overwritten. The value that remains will determine which thread executes first. Once that thread exits its critical section, it will set its $flag$ to $false$, allowing the other thread to enter its critical section. To prove that this solution is correct, the three properties must be checked:

- **Mutual exclusion:** if thread T_i is currently in its critical section, then $turn == i$ and $flag[i] = true$, which means that T_j is busy waiting in the loop. Thus, only one thread can execute its critical section at a time.
- **Progress:** if T_i wishes to enter its critical section and T_j does not, then $flag[j] == false$ and T_i will be able to bypass the loop and enter. If both threads wish to enter their critical sections, one of the values for *turn* will be overwritten, and one of the threads is guaranteed to enter its critical section.
- **Bounded waiting:** thread T_i cannot enter its critical section more than once before T_j gets a chance to, because once T_i exits its critical section, it immediately sets $flag[i] = false$, allowing T_j to exit the loop and start executing.

3 Hardware support for synchronization

Peterson's solution is considered to be **software-based** since it is not specific to an operating system and it does not require any special instructions. The following three subsections are focused on **hardware instructions** that provide support for solving the critical-section problem. These operations can be used directly as synchronization tools or can form the foundation of some more complex synchronization mechanisms.

3.1 Memory barriers

Consider the two following variables that are shared by two threads:

```
boolean flag = false;
int x = 0;
```

Also consider the code of the two threads:

Thread 1

```
while (!flag); /* waiting */
print x;
```

Thread 2

```
x = 100;
flag = true;
```

Normally, the printed value should be 100. However, since the code of T_2 does not have any data dependencies (the instructions *x = 100* and *flag = true* are independent operations that do not rely on each other's results), it is possible that a processor may, for some reason (e.g., various optimizations), reorder these instructions during execution; thus, *flag = true* would get executed before *x = 100.* In this case, it is possible that T_1 could print 0 instead of 100. Another possibility would be the reordering of the instructions in T_1, printing the value of x before executing the loop.

A **memory barrier** (also known as a **memory fence**) is a special instruction that imposes an ordering constraint on memory operations (the operations that perform loads/reads and stores/writes), preventing unpredictable behavior caused by instruction reorderings. The rule enforced by memory barriers is as follows: *all memory operations issued* *before the barrier must complete and become visible to other processors before any memory operations issued after the barrier can begin to execute.*

To solve the issue with the above threads, ensure that for T_1 the value of *flag* is loaded before printing the value of x, and for T_2 the value of x is updated before the value of *flag* is updated. Thus, the solution requires using two memory barriers as follows:

Thread 1

```
while (!flag); /* waiting */
MEMORY_BARRIER
print x;
```

Thread 2

```
x = 100;
MEMORY_BARRIER
flag = true;
```

3.2 Hardware instructions

An **atomic instruction** is a single, indivisible operation that cannot be interrupted while it executes; the instructions are all either fully executed or they are not executed at all.

3.2.1 test_and_set

The *test_and_set* function reads the current value of a memory location (the value of the boolean variable *target*), sets it to a new value (*target* becomes *true*) and returns the original value (*target* could have been *true* or *false*).

test_and_set

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;

    *target = true;

    return rv;
}
```

Mutual exclusion using *test_and_set*

```
do {
    while (test_and_set(&lock));

    /* critical section */

    lock = false;

    /* remainder section */
} while (true);
```

The value of the boolean variable **lock** indicates whether a thread is executing in its critical section or not. If **lock == true**, a thread has ”acquired” the lock and is executing in its critical section; otherwise, no thread is executing in its critical section.

Notice the loop **while (test_and_set(&lock))**; when the lock is free (**lock == false**), the first thread to execute this instruction will acquire the lock, setting its value to **true** and entering its critical section. The other threads will be kept waiting in the loop, until the lock becomes free again; once that happens, another thread (we cannot know which one) will acquire the lock and enter its critical section, blocking the remaining threads again. Obviously, **mutual exclusion** is ensured, but **bounded waiting** is not; why?

3.2.2 compare_and_swap

The *compare_and_swap* function compares the value in a memory location to an expected value, and if the two values match, the value in the memory location is replaced with a new value; otherwise, nothing happens.

compare_and_swap

```
int compare_and_swap(int *value, int
    expected, int new_value) {
    int temp = *value;

    if (*value == expected) {
        *value = new_value;
    }

    return temp;
}
```

Mutual exclusion using *compare_and_swap*

```
while (true) {
    while (compare_and_swap(&lock, 0, 1) !=
        0); /* do nothing */

    /* critical section */

    lock = 0;

    /* remainder section */
}
```

The integer variable **lock** is the same as that of the *test_and_set* instruction (0 stands for $false$, 1 stands for *true*). Notice the loop **while (compare_and_swap(&lock, 0, 1) != 0)**: as long as the current thread expects the lock to be free (*expected == 0*) but the lock is actually acquired (*value == 1*), the thread remains blocked. The moment the real value indicates that the lock is free (*value == 0*), the thread acquires the lock (sets its value to 1), enters its critical section and then frees the lock.

Mutual exclusion is ensured with this solution. However, the **bounded waiting** requirement is not met, since we cannot know which thread will acquire the lock, and some of the threads could be kept waiting forever. A solution like the one below would satisfy all three requirements:

```
 1  while (true) {
 2      waiting[i] = true;
 3      key = 1;
 4
 5      while (waiting[i] && key == 1) {
 6          key = compare_and_swap(&lock, 0, 1);
 7      }
 8
 9      waiting[i] = false;
10
11      /* critical section */
12
13      j = (i + 1) % n
14      while ((j != i) && !waiting[j]) {
15          j = (j + 1) % n;
16      }
17
18      if (j == i) {
19          lock = 0;
20      }
21      else {
22          waiting[j] = false;
23      }
24
25      /* remainder section */
26  }
```

(LINES 2-3) For this approach, two additional variables are introduced: the boolean array **waiting** (which will be shared amongst all threads) and the integer variable **key** (which is local to every individual thread; an auxiliary variable that stores the result of the CAS operation). The main idea is that we continue to cycle through all threads to make sure that no thread waits forever to enter its critical section. If a thread wishes to enter its critical section, it means that *waiting[i] = true*.

(LINES 5-7) Initially, the lock is free, all threads wish to enter their critical sections, and the value of *key* is set to 1. Consider the instruction **key = compare_and_swap(&lock, 0, 1)**: if the lock is free (*lock == 0*), a thread will acquire it (*lock = 1*) and the old value of *lock* will be stored in the variable *key* (so *key = 0*). The value of *key* will be set to 0 for only the thread that acquired the lock; thus, that thread will exit the loop and the others will continue waiting.

(LINES 9-23) Assume that T_i is the thread that set *key* = 0 and exited the loop. It will set *waiting[i] = false* since it will no longer be waiting to enter its critical section. Once it executes and then exits its critical section, it will cycle through the threads, looking for the next thread that is waiting to enter its critical section. During searching, it may find a thread T_j with *waiting[j] = true*; in this case, it will set $waiting[j] = false$, allowing T_j to exit the busy waiting loop and enter its critical section. Notice how it does not have to set *lock* = 0; it is enough to set $waiting[j] = false$ to allow T_j to execute, and thus the lock will still be 1 because it will be "acquired" by T_j.

Assume that T_i cycled through the threads all the way back to $j = i$ and did not find a thread that wanted to enter its critical section; this means that $waiting[j] == false$ for all threads T_j. Consequently, the lock is freed by setting *lock* = 0, since there are no threads (at least at the current moment) that wish to enter their critical sections.

3.3 Atomic variables

In the beginning, we talked about the race condition that occurs when two threads execute *count++* and *count−−* at the same time. **Atomic variables** are variables for which the read, write, and update operations are guaranteed to be indivisible, preventing other threads from interfering. This solves race conditions by making sure that an operation is completed entirely or not at all, avoiding data corruption. Functions that manipulate atomic variables are often implemented using the atomic instruction CAS:

```
void increment(atomic_int *v){
    int temp;
    do {
        temp = *v;
    } while (temp != compare_and_swap(v, temp, temp + 1));
}
```

The do-while loop repeatedly tries to increment the atomic variable until it succeeds, since the operation can fail because of other threads interfering. First, the initial value of the variable is stored in *temp*. The CAS function is used to check if the variable's current value is equal to the expected value in *temp*; if so, the variable is incremented, and the loop ends. Otherwise, the loop continues.

4 Mutex locks

The solutions discussed earlier are not really straight-forward; there are other more high-level mechanisms that can be used to solve synchronization problems in a more accessible manner. One of these would be the **mutex lock** (where *mutex* stands for *mutual exclusion*).

When a thread enters its critical section, it **acquires** the lock. As long as the lock is acquired, other threads trying to acquire it will be blocked. Once the thread exits its critical section, it **releases** the lock, allowing another thread to acquire it and enter its critical section.

acquire

```
acquire() {
    while (!available);
    available = false;
}
```

release

```
release() {
    available = true;
}
```

A mutex lock is considered to be **contended** if a thread blocks when trying to acquire it; otherwise, it is **uncontended**. A contended lock could have **high contention** (many threads are trying to acquire it) or **low contention** (not that many threads are trying to acquire it). Highly contended locks can degrade the overall performance of a system. The code below illustrates a simple application of mutex locks:

```
while (true) {
    acquire(lock);

    /* critical section */

    release(lock);

    /* remainder section */
}
```

5 Semaphores

A **semaphore** is an integer variable that is used to manage access to a finite number of shared resources by multiple threads. Apart from initializing the semaphore, the only permitted operations are **signal/increment** and **wait/decrement**. Semaphores are useful for solving synchronization problems where there is a finite amount of resources and a greater number of threads that wish to access those resources and were invented by Edsger Dijkstra.

When a thread wishes to access a resource, it decrements the value of the semaphore by calling *wait* (or *decrement*). If the new value of the semaphore is ≥ 0, the thread will receive its resource and continue to run. Otherwise (the new value is < 0), there are no resources available and the thread blocks until a resource becomes available, that is, until another thread calls *signal* or (*increment*). A semaphore can be of two types: a **counting semaphore** (the values range over an unrestricted domain) or a **binary semaphore** (which only has values of 0 and 1).

5.1 Implementation

A simple way of defining the operations for the semaphore would be as follows (making use of busy waiting loops):

wait (decrement)

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

signal (increment)

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

The issue with this approach is wasted CPU time; the threads that are kept waiting in the loop are still active, consuming valuable CPU time by constantly running the condition in the loop. To resolve this issue, we can assign two attributes to a semaphore: an integer variable (the number of available resources) and a list of threads associated with the semaphore.

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

When there are no resources left and a thread calls *wait*, it is put to sleep and placed in the semaphore's list. When another thread calls *signal*, a thread is removed from the waiting list and awakened (unless the list is empty).

wait (decrement)

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

signal (increment)

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

Note that the implementations of the *sleep* and *wakeup* functions should be provided by the OS.

5.2 Semaphore utilization

Consider a semaphore initialized with the value 0. The code below makes sure that T_1 reads the line before it is printed by T_2:

Thread 1

```
read line
signal(S)
```

Thread 2

```
wait(S)
print line
```

Take a look at the following code:

Thread 1

```
statement a1
statement a2
```

Thread 2

```
statement b1
statement b2
```

We want to ensure that the instruction a_1 is executed before b_2, and that b_1 is executed before a_2. The orders between a_1 and b_1 (or a_2 and b_2) are irrelevant for this exercise. This synchronization problem is known as a **rendezvous**. We can use two different semaphores **aArrived** and **bArrived**, both initialized with the value 0. A possible solution for this problem is as follows:

Thread A

```
        statement a1
        bArrived.wait()
        aArrived.signal()
        statement a2
```

Thread B

```
        statement b1
        bArrived.signal()
        aArrived.wait()
        statement b2
```

6 Condition variables

Imagine that a thread needs to access a shared resource, but only after a certain condition is met. For example, the *producer-consumer problem:* the consumer acquires a mutex lock to ensure mutual exclusion while manipulating the shared data. It checks whether the queue is empty or not (the condition); if the queue is not empty, the consumer proceeds, does its work, and releases the lock. Otherwise, the consumer must wait for the producer process to add an item. A naive approach to waiting would be to introduce a busy waiting loop, which wastes CPU resources and blocks the producer process from ever entering its critical section, since the consumer will be indefinitely holding the mutex lock.

A **condition variable** is a queue of waiting threads that is associated with a mutex lock and a condition, which aims to solve the issue presented earlier. When a thread finds its condition is not met (e.g., the buffer is empty), it calls the function *wait*, immediately putting the thread to sleep and releasing the mutex lock. Once a thread updates the shared resource (e.g., the producer adds an item to the buffer), that thread calls the function *signal* to wake up a waiting thread or *broadcast* to wake up all waiting threads. The awakened threads try to reacquire the mutex; only one thread will successfully reacquire it and the others will be put back to sleep.

7 Monitors

An **abstract data type (ADT)** is a conceptual model of a data structure that bundles shared data with the methods that operate on it (e.g., **stacks** and **queues** are abstract data structures with operations such as *pop*, *push*, and *isEmpty*; some common implementations for stacks and queues would be arrays and linked lists).

A **monitor** is an ADT that encapsulates the shared data with the methods that operate on it, acting as a high-level synchronization mechanism; it is an object, a class, or a module that bundles together the resources and the logic to control access to them. Only one thread can execute within the monitor's methods at a time, ensuring mutual exclusion. Monitors use a mutex lock and can also include *condition variables.* The basic overall structure of a monitor would be as follows:

```
monitor MonitorName {
    /* shared variable declarations */

    initialization_code(args...) {
        ...
    }

    function F1(args...) {
        ...
    }

    ...

    function Fn(args...) {
        ...
    }
}
```

8 Liveness

Liveness refers to a set of properties that a system must satisfy to ensure that processes/threads make progress during their execution life cycle. A process waiting indefinitely to enter its critical section and infinite loops are two common examples of ”liveness failures”. Two other examples would be **deadlocks** and **priority inversions**.

8.1 Deadlocks

A situation in which two or more threads are indefinitely waiting for an event that can only be caused by one of those waiting threads is known as a **deadlock** (and the threads are said to be **deadlocked**). For example, earlier we talked about condition variables; when a consumer process acquires the mutex lock, checks the buffer, and sees there are no items to consume, it must wait and release the lock to allow the producer process to place items in the buffer; however, if the consumer process were to wait while having the mutex lock, the two processes would get deadlocked. Take another fresh example:

Thread 1

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

Thread 2

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

This is a system consisting of two threads T_1 and T_2, each accessing two binary semaphores Q and S. If T_1 executes *wait(Q)* and then T_2 executes *wait(S)* and *wait(Q)*, then T_2 will be put to sleep, because it will be waiting for the resource from semaphore Q, which is currently taken by T_1. After that, T_1 executes *wait(S)* and gets put to sleep, as it will be waiting for the resource from the semaphore S, which belongs to T_2. In this way, both threads will wait indefinitely for each other to free their resources, entering a deadlocked state.

8.2 Priority inversion

Consider that a low-priority task L begins its execution and acquires a shared resource, such as a mutex lock, which is required by a high-priority task H. The task H becomes ready to run and preempts task L; it attempts to acquire the shared resource (the mutex lock) currently held by L. Thus, it blocks and waits for L to release the resource; now, L becomes the highest-priority runnable task (since H is blocked) and resumes its execution. However, a medium-priority task M enters the ready queue and preempts L; ultimately, M runs for a potentially long unbounded amount of time.

The issue is that the highest-priority task H is forced to wait for a medium-priority task M to finish, so that the low-priority task L can release the shared resource. This is known as a **priority inversion**, which is an example of liveness failure that occurs in a priority-based preemptive scheduling system through a sequence of events, such as the one described above, involving three or more tasks with different priority levels.