Process Synchronisation (Part II)

Eike Ritter [1]

Modified: November 2, 2012

Operating Systems with C/C++
School of Computer Science, University of Birmingham, UK

[1] Based on material by Matt Smart and Nick Blundell

Outline

Recap

- Previously, we looked at:
  - The problem of accessing shared data
  - In particular, we saw how race conditions can result in variable updates being overwritten, causing inconsistent results
  - We saw how such race conditions can be avoided by guarding the critical sections of our code with synchronisation mechanisms
  - We saw how it is important for a particular mutual exclusion mechanism to ensure that access is granted fairly to all waiting threads/processes and that threads are not waiting unduly.
  - We saw how atomic CPU instructions, such as TestAndSet, which let us read and modify a value in one CPU cycle, can be used with lock variables for enforcing mutual exclusion.

But There's a Bit of a Problem. . .

- Consider the simple mutual exclusion mechanism we saw in the last lecture:

```
do {
  while (TestAndSet(&lock)) ; // wait until we successfully
                             // change lock from false to true
    [critical section]
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

- This guarantees mutual exclusion but with a high cost: that while loop spins constantly (using up CPU cycles) until it manages to enter the critical section.
- This is a huge waste of CPU cycles and is not acceptable, particularly for user processes where the critical section may be occupied for some time.

Sleep and Wakeup

- Rather than having a process spin around and around, checking if it can proceed into the critical section, suppose we implement some mechanism whereby it sends itself to sleep and then is awoken only when there is a chance it can proceed
- Functions such as `sleep()` and `wakeup()` are often available via a threading library or as kernel service calls.
- Let's explore this idea

Mutual Exclusion with sleep(): A first Attempt

- If constant spinning is a problem, suppose a process puts itself to sleep in the body of the spin.
- Then whoever won the competition (and therefore entered the critical section) will wake all processes before releasing the lock - so they can all try again.

```
do {
  while (TestAndSet(&lock)) { // If we can't get the lock, sleep.
    sleep();
  }
    [critical section]
  wake_up(all); // Wake all sleeping threads.
  lock = FALSE; // Release lock
    [remainder section]
} while (TRUE);
```

Mutual Exclusion with sleep(): A first Attempt

- Is this a satisfactory solution?
- Does it achieve mutual exclusion?
- Does it allow progress (*i.e.* stop non-interested processes from blocking those that want to get it)?
- Does it have the property of bounded waiting?

Towards Solving The Missing Wakeup Problem

- Somehow, we need to make sure that when a process decides that it will go to sleep (if it failed to get the lock) it actually goes to sleep without interruption, so the wake-up signal is not missed by the not-yet-sleeping process
- In other words, we need to make the check-if-need-to-sleep and go-to-sleep operations happen atomically with respect to other threads in the process.
- Perhaps we could do this by making use of another lock variable, say `deciding_to_sleep`.
- Since we now have two locks, for clarity, let's rename `lock` to `mutex_lock`.

A Possible Solution?

```
while (True) {
  // Spinning entry loop.
  while (True) {
    // Get this lock, so a wake signal cannot be raised before we actually sleep.
    while(TestAndSet(&deciding_to_sleep));

    // Now decide whether or not to sleep on the lock.
    if (TestAndSet(&mutex_lock)) {
      sleep();
    } else {
      // We are going in to critical section.
      deciding_to_sleep = False;
      break;
    }
    deciding_to_sleep = False; // Release the sleep mutex for next attempt.
  }
    [critical section]
  while(TestAndSet(&deciding_to_sleep)); // Don't issue 'wake' if a thread is
                                         // deciding whether or not to sleep.
  mutex_lock = False; // Open up the lock for the next guy.
  wake_up(all); // Wake all sleeping threads so they may compete for entry.
  deciding_to_sleep = False;
    [remainder section]
}
```

- Have we, perhaps, overlooked something here?

Sleeping with the Lock

- We've encountered an ugly problem — in fact, there are several problems with the previous example that are all related.
    - As we said before, we need to decide to sleep and then sleep in an atomic action (with respect to other threads/processes)
    - But in the previous example, when a thread goes to sleep it keeps hold of the `deciding_to_sleep` lock which sooner or later will result in deadlock!
    - And if we release the `deciding_to_sleep` lock immediately before sleeping, then we have not solved the original problem.
- What to do, what to do…

Sleeping with the Lock

- The solution to this problem implemented in modern operating systems, such as Linux, is to release the lock *during* the kernel service call `sleep()`, such that it is released prior to the context switch, with a guarantee there will be no interruption.
- The lock is then reacquired upon wakeup, prior to the return from the `sleep()` call.
- Since we have seen how this can all get very complicated when we introduce the idea of sleeping (to avoid wasteful spinning), kernels often implement a sleeping lock, called a *semaphore*, which hides the gore from us.
- See `http://tomoyo.sourceforge.jp/cgi-bin/lxr/source/kernel/semaphore.c#L75`

Semaphores

- Synchronisation tool, proposed by E. W. Dijkstra (1965), that
  - Simplifies synchronisation for the programmer
  - Does not require (much) busy waiting
    - We don't busy-wait for the critical section, usually only to achieve atomicity to check if we need to sleep or not, *etc.*
  - Can guarantee *bounded waiting time* and *progress*.
- Consists of:
  - A semaphore type `S`, that records a list of waiting processes and an integer
  - Two standard atomic ($\leftarrow$ very important) operations by which to modify `S`: `wait()` and `signal()`
    - Originally called `P()` and `V()` based on the equivalent Dutch words

Semaphores

- Works like this:
  - The semaphore is initialised with a count value of the maximum number of processes allowed in the critical section at the same time.
  - When a process calls `wait()`, if count is zero, it adds itself to the list of sleepers and blocks, else it decrements count and enters the critical section
  - When a process exits the critical section it calls `signal()`, which increments count and issues a wakeup call to the process at the head of the list, if there is such a process
    - It is the use of ordered wake-ups (*e.g.* FIFO) that makes semaphores support bounded (*i.e.* fair) waiting.

Semaphore as General Synchronisation Tool

- We can describe a particular semaphore as:
  - A Counting semaphore - integer value can range over an unrestricted domain (*e.g.* allow at most N threads to read a database, *etc.*)
  - A Binary semaphore - integer value can range only between 0 and 1
    - Also known as mutex locks, since ensure mutual exclusion.
    - Basically, it is a counting semaphore initialised to 1

Critical Section Solution with Semaphore

```
Semaphore mutex; // Initialized to 1
do {
  wait(mutex); // Unlike the pure spin-lock, we are blocking here.
    [critical section]
  signal(mutex);
    [remainder section]
} while (TRUE);
```

Semaphore Implementation: State and Wait

We can implement a semaphore within the kernel as follows (note that the functions must be atomic, which our kernel must ensure):

```
typedef struct {
  int count;
  process_list; // Hold a list of waiting processes/threads
} Semaphore;

void wait(Semaphore *S) {
  S->count--;
  if (S->count < 0) {
    add process to S->process_list;
    sleep();
  }
}
```

Semaphore Implementation: State and Wait

- Note that, here, we decrement the wait counter before blocking (unlike the previous description).
- This does not alter functionality but has the useful side-effect that the negative count value reveals how many processes are currently blocked on the semaphore.

Semaphore Implementation: Signal

```
void signal(Semaphore *S) {
  S->count++;
  if (S->count <= 0) { // If at least one waiting process, let him in.
    remove next process, P, from S->process_list
    wakeup(P);
  }
}
```

But we have to be Careful: Deadlock and Priority Inversion

- Deadlock: Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let `S` and `Q` be two semaphores initialized to 1

```
Process 1            Process 2
  wait(S);             wait(Q);
  wait(Q);             wait(S);
  .                    .
  .                    .
  .                    .
  signal(S);           signal(Q);
  signal(Q);           signal(S);
```

- Priority Inversion: Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Good account of this from NASA programme: `http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html`

Classical Problems of Synchronisation and Semaphore Solutions

- Bounded-Buffer Problem
- Readers and Writers Problem
- Boring as it may be to visit (and teach about) the same problems over and over again, a solid understanding of these seemingly-insignificant problems will directly help you to spot and solve many related real-life synchronisation issues.

Our Old Friend: The Bounded-Buffer Problem

- The solution set-up:
  - A buffer to hold N items, each can hold one item
  - Semaphore `mutex` initialized to the value 1
  - Semaphore `full_slots` initialized to the value 0
  - Semaphore `empty_slots` initialized to the value N.

Producer

```
while(True) {
  wait(empty_slots); // Wait for, then claim, an empty slot.
  wait(mutex);
  // add item to buffer
  signal(mutex);
  signal(full_slots); // Signal that there is one more full slot.
}
```

Consumer

```
while(True) {
  wait(full_slots); // Wait for, then claim, a full slot
  wait(mutex);
  // consumer item from buffer
  signal(mutex);
  signal(empty_slots); // Signal that now there is one more empty slot.
}
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers - only read the data set; they do not perform any updates
  - Writers - can both read and write
- Problem:
  - Allow multiple readers to read at the same time if there is no writer in there.
  - Only one writer can access the shared data at the same time

Readers-Writers Problem

- The solution set-up:
  - Semaphore `mutex` initialized to 1
  - Semaphore `wrt` initialized to 1
  - Integer `readcount` initialized to 0

Writer

```
while(True) {
  wait(wrt); // Wait for write lock.
  // perform writing
  signal(wrt); // Release write lock.
}
```

Reader

```
while(True) {
  wait(mutex) // Wait for mutex to change read_count.
  read_count++;
  if (read_count == 1) // If we are first reader, lock out writers.
    wait(wrt)
  signal(mutex) // Release mutex so other readers can enter.

  // perform reading

  wait(mutex) // Decrement read_count as we leave.
  read_count--;
  if (read_count == 0)
    signal(wrt) // If we are the last reader to
  signal(mutex) // leave, release write lock
}
```

Summary

We looked at:

1. Inefficient Spinning
   - The Problem
   - Why not Sleep for a Bit?
   - The Missing Wake-up Problem
2. Semaphores
   - What's That?
   - Semaphore Implementation
   - Deadlock and Priority Inversion
3. Semaphore Examples
   - Semaphore Solutions to Synchronisation Problems
   - Bounded-Buffer Semaphore Solution
   - Readers and Writers Semaphore Solution