Process Synchronisation (Part I)

Eike Ritter [1]

Modified: October 31, 2012

Lecture 9: Operating Systems with C/C++
School of Computer Science, University of Birmingham, UK

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

Outline

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Our solution in the IPC lecture only allowed us to put at most `BUFFER_SIZE-1` items in the buffer...

Old Producer Process Code

```
// Produce items until the cows come home.
while (TRUE) {
  // Produce an item.
  Item next_produced = [some new item to add to buffer];

  // Wait one place behind the next item to be consumed - so we don't
  // write to items that have yet to be consumed.
  while (((in + 1) % BUFFER_SIZE) == out); // <- Spin on condition

  // Store the new item and increment the 'in' index.
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

Old Consumer Process Code

```
while (true) {
  // Wait until there is something to consume.
  while (in == out); // <- Spin on condition
  // Get the next item from the buffer
  Item next_item = buffer[out];
  [process next_item]

  // Increment the out index.
  out = (out + 1) % BUFFER_SIZE;
}
```

Improving the Bounded Buffer

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills all the buffer slots, rather than `BUFFER_SIZE-1`.
  - We can do so by having an integer `count` that keeps track of the number of full buffers. Initially, `count` is set to `0`. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item.
- When we looked at this problem in the earlier lecture, we didn't consider problems of concurrent modification of shared variables - the focus of this lecture.

Producer

```
while (true) {
  /* produce an item and put in nextProduced */
  while (count == BUFFER_SIZE); // wait if buffer full
  buffer [in] = nextProduced; // store new item
  in = (in + 1) % BUFFER_SIZE; // increment IN pointer.
  count++; // Increment counter
}
```

Consumer

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

Race Condition

There is something wrong with this code!

- `count++` could be compiled as a sequence of CPU instructions:
  - `register1 = count`
  - `register1 = register1 + 1`
  - `count = register1`
- `count--` could be compiled likewise:
  - `register2 = count`
  - `register2 = register2 - 1`
  - `count = register2`

Race Condition

- Consider that the producer and consumer execute the `count++` and `count--` around the same time, such that the CPU instructions are interleved as follows (with `count = 5` initially):
  - Prod.: `register1 = count` $\{$`register1` $\rightarrow$ `5`$\}$
  - Prod.: `register1 = register1 + 1` $\{$`register1` $\rightarrow$ `6`$\}$
  - Cons.: `register2 = count` $\{$`register2` $\rightarrow$ `5`$\}$
  - Cons.: `register2 = register2 - 1` $\{$`register2` $\rightarrow$ `4`$\}$
  - Prod.: `count = register1` $\{$`count` $\rightarrow$ `6`$\}$
  - Cons.: `count = register2` $\{$`count` $\rightarrow$ `4`$\}$
- With an increment and a decrement, in whatever order, we would expect no change to the original value (`5`), but here we have ended up with `4`?!?!.

Solution Criteria to Critical-Section Problem

- So we must find a solution to protect against concurrent modification of data in what is termed the *critical section* with the following criteria:
  - **Mutual Exclusion** - If process P_i is executing in its critical section (*i.e.* where shared variables could be altered inconsistently), then no other processes can be executing in their critical sections.
  - **Progress** - no process outside of the critical section (*i.e.* in the *remainder section*) should block a process waiting to enter.
  - **Bounded Waiting** - A bound must exist 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 (*i.e.* it must be fair, so one poor process is not always waiting behind the others).
- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the N processes.

Peterson's Solution

- Two process solution.
- Assume that the CPU's register LOAD and STORE instructions are atomic; that is, cannot be interrupted.
  - With the extreme optimisation of modern CPU architectures this assumption no longer holds, but this algorithm is of historical relevance.
- The two processes share two variables:
  - `int turn;`
  - `Boolean wants_in[2];`
- The variable `turn` indicates whose turn it is to enter the critical section.
- The `wants_in` array is used to indicate if a process is ready to enter the critical section. `wants_in[i] = true` implies that process P_i is ready!

But firstly: A Naïve Algorithm for Process P_i

```
is_in[i] = FALSE; // I'm not in the section
do {
  while (is_in[j]); // Wait whilst j is in the critical section
  is_in[i] = TRUE; // I'm in the critical section now.
    [critical section]
  is_in[i] = FALSE; // I've finished in the critical section now.
    [remainder section]
} while (TRUE);
```

What's wrong with this?

Peterson's Algorithm for Process P_i

```
do {
  wants_in[i] = TRUE; // I want access...
  turn = j; // but, please, you go first
  while (wants_in[j] && turn == j); // if you are waiting and it is
                                    // your turn, I will wait.
    [critical section]
  wants_in[i] = FALSE; // I no longer want access.
    [remainder section]
} while (TRUE);
```

- When both processes are interested, they achieve fairness through the `turn` variable, which causes their access to alternate.

Peterson's Algorithm for Process P_i

- Interesting, but:
  - Aside from the question of CPU instruction atomicity, how can we support more than two competing processes?
  - Also, if we were being pedantic, we could argue that a process may be left to wait unnecessarily if a context switch occurs only after another process leaves the remainder section but before it politely offers the turn to our first waiting process.

Synchronisation Hardware

- Many systems provide hardware support for critical section code.
- Uniprocessors - could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Delay in one processor telling others to disable their interrupts
- Modern machines provide the special atomic hardware instructions (Atomic = non-interruptible) `TestAndSet` or `Swap` which achieve the same goal:
  - `TestAndSet`: Test memory address (*i.e.* read it) and set it in one instruction
  - `Swap`: Swap contents of two memory addresses in one instruction
- We can use these to implement simple locks to realise mutual exclusion

Solution to Critical-section Problem Using Locks

- The general pattern for using locks is:

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

TestAndndSet Instruction

- High-level definition of the atomic CPU instruction:

```
boolean TestAndSet (boolean *target) {
  boolean original = *target; // Store the original value
  *target = TRUE; // Set the variable to TRUE
  return original: // Return the original value
}
```

- In a nutshell: this single CPU instruction sets a variable to TRUE and returns the original value.
- This is useful because, if it returns FALSE, we know that only our thread has changed the value from FALSE to TRUE; if it returns TRUE, we know we haven't changed the value.

Solution using TestAndSet

- Shared boolean variable `lock`, initialized to false.
- Solution:

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

- Note, though, that this achieves mutual exclusion but not *bounded waiting* - one process could potentially wait for ever due to the unpredictability of context switching.

Bounded-waiting Mutual Exclusion with TestandSet()

- All data structures are initialised to FALSE.
- `wants_in[]` is an array of waiting flags, one for each process.
- `lock` is a boolean variable used to lock the critical section.

Bounded-waiting Mutual Exclusion with TestandSet()

```
boolean wants_in[], key = FALSE, lock = FALSE; //all false to begin with

do {
  wants_in[i] = TRUE; // I (process i) am waiting to get in.
  key = TRUE; // Assume another has the key.
  while (wants_in[i] && key) { // Wait until I get the lock
                                  // (key will become false)
    key = TestAndSet(&lock); // Try to get the lock
  }
  wants_in[i] = FALSE; // I am no longer waiting: I'm in now

  [*** critical section ***]

  // Next 2 lines: get ID of next waiting process, if any.
  j = (i + 1) % NO_PROCESSES; // Set j to my right-hand process.
  while ((j != i) && !wants_in[j]) { j = (j + 1) % NO_PROCESSES };

  if (j == i) { // If no other process is waiting...
    lock = FALSE; // Release the lock
  } else { // else ....
    wants_in[j] = FALSE; // Allow j to slip in through the 'back door'
  }
  [*** remainder section ***]
} while (TRUE);
```

If your head hurts. . .

- Don't feel too bad if your head hurts a little after looking at this stuff
  - Solutions to synchronisation among concurrent processes/threads are not obvious
  - Synchronisation bugs arise in many professional software products, since they may take many millions of permutations to manifest themselves - but when they do, they can cause all sorts of havoc.
  - Since some programmers may not be fully aware of the reasons for such bugs, they tend to under- or over-compensate, making code vulnerable or inefficient.
- Have a good look through the examples in this lecture and the book, since synchronisation is very important for this and many other areas of computer science, such as distributed systems and computer networking.

Summary

We looked at:

1. The Critical-Section Problem
   - Background
   - Race Conditions
   - Solution Criteria to Critical-Section Problem
   - Peterson's (Software) Solution
2. Synchronisation Hardware
   - Hardware Supported Locks
   - TestAndSet Solution
   - Bounded-waiting Solution using TestAndSet

Next OS lecture: **More on Synchronisation!**