1 Multiprocessor Synchronization

Communication (in a distributed computing model) using shared memory model.
Focus – solving a single problem using a parallel computing model. Parallelism is used to increase speed of computation. In contrast to distributed computing, where the problem statement includes the division of the problem between processes and/or their coordination.

1.1 Comparison to Message Passing

In shared memory model (which will be the focus of this class) all processors have access to some common memory locations, which they access using read, write, and optionally more complex operations. In message passing model, each processor has a link to its neighbors. Correctness issues in message passing models can be more difficult to analyse.

It is possible to emulate shared memory using message passing – pass one message per write or read. This is very inefficient, but techniques like caching can improve this. It is also possible to emulate message passing using shared memory; just have areas set aside for message queues.

Impossibility result – consensus is impossible in a message passing model if a single processor fails. The equivalence of the two models is useful to apply these results to shared memory models.

Synchronizing single-cell reads or writes is much easier than synchronizing reads or writes to a collection of $k$ cells between multiple processors.

Example Processor $P_0$ writes to a memory location and $P_1$ reads from this location. If the processors are fully asynchronous, we need an unbounded shared memory because there is no guarantee that $P_1$ has received any given message that $P_0$ has sent, so no given message can be overwritten.

1.2 Properties of Concurrent Programs

Reasoning is inherently difficult because the ordering is not static.

Safety property bad things don’t ever happen
Liveness property  good things eventually happen

Mutual Exclusion  Control access to a shared resource between multiple processors such that only one processor accesses the resource (or critical section) at a time. The liveness properties in this problem are:

no deadlock  When a processor $P_i$ is trying to access the critical section, some processor $P_j$ will eventually be able to enter the critical section.

no starvation  When a processor $P_i$ is trying to access the critical section, processor $P_i$ will eventually be able to enter the critical section.

Example  Processors $a, b, c$ all want to access the criital section. If the critical section is given only to $a$ and $b$ (because they request access repeatedly), then no deadlock is satisfied but no starvation is not satisfied as $c$ never enters.

Increment counter  Find primes in the range $[n_1, n_2]$. Trivial division of the problem is not efficient because primes are more common in lower numbers. The best division of the problem is to assign numbers to processors, and have processors ask for the next number in sequence after they are finished. This requires a counter that can be incremented atomically.

The trivial implementation of this algorithm can repeat numbers because reads and writes can be interleaved arbitrarily; the value of the counter can even go down in the case where one processor is much slower than another.

1.3 Mutual Exclusion Implementation (1)

Processor 0
1: $flag_0 := 1$
2: Wait until $flag_1 = 0$
3: Critical section
4: $flag_0 := 0$

Processor 1
1: $flag_1 := 1$
2: while $flag_0 = 1$ do
3:  $flag_1 := 0$
4:  Wait until $flag_0 = 0$
5:  $flag_1 := 1$
6: end while
7: Critical section
8: $flag_1 := 0$

Mutual exclusion (safety)

Proof: By contradiction. Note that when a processor is in the critical section, its flag is set to 1. Assume a processor enters the critical section. The processor must have set its flag to 1, and after setting its flag, the processor also must have read a zero on the other processor’s flag. Therefore the other processor is not in its critical section at the moment.

Deadlock freedom (liveness)

Proof: Assume $P_0$ is trying to enter the critical section. It must have completed step 1, and can only be waiting in step 2. Eventually, $P_1$ will read $flag_0 = 1$ and will set its flag to 0 and not change again. This allows step 2 to complete and therefore enters the critical section.

Starvation freedom (liveness) It is possible for process $P_1$ to never enter the critical section if $P_0$ always has its flag set to 1 when the conditions of steps 2 and 4 are checked. This can happen if $P_1$ enters the critical section in a tight loop and $P_1$ never checks the flag during the times when it is zero.

Starvation-free version Create a priority bit which determines if you use algorithm 0 or algorithm 1; switch the priority bit at the end of each critical section’s execution.

1.4 Producer-Consumer Problem

Producer creates objects/resources of some type, and consumer will use this object. There is a limited storage space for the objects; safety requires that producer stop when the space is full and the consumer only consume objects that actually exist. Liveness conditions: if there is space available, the producer will be allowed to produce; if there are objects available, the consumer will be able to consume.

Additional simplifying condition: consumer consumes whenever there is something to consume, and producer only produces when there are no items in the storage space.

Producer
1: loop
2: Wait until $flag = 1$
3: Produce items
4: $flag := 0$
5: end loop
Consumer

1: loop
2: Wait until $flag = 0$
3: Consume all items
4: $flag := 1$
5: end loop

This algorithm is not suitable to replace a mutex because both producer and consumer wait for the other. This is not wait-free, and so has problems with fault tolerance. Wait-free algorithms will be covered in the future.

1.5 Readers/Writers Problem

Multiple processors need to be able to read and write to a shared object (which spans multiple cells). Readers can access the object concurrently, but writers need exclusive access so that readers don’t read partially written messages (or so that two written messages aren’t interleaved).