Concurrent and Distributed Systems - 2019–2020

Supervision 0: Get Started Questions (DJ Greaves)

* Star denotes optional/advanced exercise.

Q0 Parallel Programming

List or tabulate the essential similarities and/or differences between parallel programming and distributed systems.

Q1 LL/SC FSM Diagram

How might you draw compare-and-swap or LL/SC interactions in an FSM view. Draw diagram for one thread. Describe product construction properties in a few words.

Q2 Philosophers FSM

Draw out FSM product for 2 or 3 philosophers.

Q3 CBMC Example (*)

* Start working towards applying the CBMC model checker on the Beer Fridge Stocking problem with two people. I will show you how to create undetermined input (with declared but undefined functions) and multiple people threads (using posix pthread_create) as lectures progress. To start, just try to capture one or two of the properties that we would like to verify.

Q4 LL/SC Behaviour on Context Switch (*)

* Consider why an OS based over the LL/SC mechanism should issue an instruction during a context switch to clear any pending LL that has not been cleared by an SC.

Q5 Replication of State

Why is replication of state generally undesirable (two reasons) in computer systems? The standard producer/consumer solution, as lectured, uses two queue pointers and two semaphores. How much replication of state is there? Discuss whether any replication is good or bad style.

Q6 P-C Relaxation

Can we modify the generalized P-C solution, as lectured, to allow concurrent access by 1 producer and 1 consumer by adding one more semaphore?

Q7 MRSW Monitor

Sketch an MRSW monitor implementation (L04, slide 24).

(Beginners will find this a very hard exercise. But it is well worthwhile, so I have not put an asterisk.)

The monitor code suggested in the lecture was for signal-and-continue semantics and it was (roughly) as follows:

```
monitor ProducerConsumer {
   int in, out; item_t buf[N];
   condition notfull, notempty;

procedure produce(item) {
  while ((in-out) == N) wait(notfull);
  buf[in % N] = item;
  if ((in-out) == 0) signal(notempty); // NB signal_and_continue
  in = in + 1;
  }

procedure item_t consume() {
  while ((in-out) == 0) wait(notempty);
  item = buf[out % N];
  if ((in-out) == N) signal(notfull); // NB signal_and_continue
  out = out + 1;
  return(item);
  }
/* init */ { in = out = 0; }
}
```

Th example code from the lecture, a queue, had a fairly lightweight consume operator. A better example has more intensive lookup behaviour in the read operation. So rather than a simple FIFO, we now consider something like an associative dictionary or tree where the consume operator takes an argument and involves some digging around.

Of course, having all of the read/consume/lookup code in a monitor will not give the desired advantage of the MRSW paradigm. We need the lookup code to be runnable by multiple threads at once and for the monitor to provide ancillary thread guarding to stop simultaneous reading and writing. So implement as a monitor (that has internal condition variable(s)) the mrsw_lock needed to serve the following thingy that has a datastructure which holds data of type datum_t:

```
class mutable_thingy
{
  mrsw_lock lock;
  SortedListOrSomething<datum_t> storedv;

  datum_t reader(arg)  {
    lock.start_read();
    datum_t rr = storedv.search_somehow(arg);
    lock.end_read();
    return rr;
  }
```

```
  void writer(datum_t nv) {
    lock.start_write();
    storedv.augment_somehow(nv);
    lock.end_write();
    }
}
```

Q8 Co-routines (*)

(a) * Early versions of the Windows Operating System (pre circa 1998) lacked most features that would be expected to be found in an operating system: it was essentially a GUI-shell. List the minimal abstractions expected from an operating system. Explain why a GUI-shell might require threads at least.

(b) * Concurrency can be provided without a sophisticated scheduller using a co-routine package. A set of co-routines time-share on a voluntary basis by calling yield() which blocks the current thread that will resume again (by returning from yield()) after all other threads have been run. There is no pre-emption. The other essential API call provided by a co-routine package is a create_thread(void (*f)(void)) which is passed the address of a C routine (that normally contains an infinite loop whose body contains calls to yield()). [The syntax for passing a function to a function in C is a little off-putting, but should eventually be familiar to those reading the C course.] Sketch or describe the code for a co-routine package, being clear about the central datastructure needed and how the stack pointer should be handled. If you want to do a full C implementation, you should first be familiar with longjump.

(c) * Early versions of Windows provided co-routines as the only concurrency mechanism. What problems would this have led to and for what application scenario might it be common to use co-routines today?