Concurrent and Distributed Systems - 2022–2023

CS2: Semaphores, Generalised producer-consumer, and Priorities. (Rev A)

* Star denotes optional/advanced exercise. Estimated times to complete excluded starred sub-parts.

Q1 Semaphores/Deadlock

Six parts, each taking about one minute.

(a) Semaphores are initialised to a value — 0, 1, or some arbitrary n. For each case, list one use case for which that initialisation would make sense.

(b) Where semaphores are being used for **mutual exclusion**, write down two fragments of pseudo-code, to be run in two different threads, that can experience deadlock.

(c) Deadlock may occur when its four necessary conditions (mutual exclusion, hold-and-wait, no preemption, circular wait) are met. Describe a situation in which two threads, making use of semaphores for **condition synchronisation** (i.e., conveying an inter-thread event or message), can deadlock.

(d) In Figure 1, `items` and `spaces` are used for condition synchronisation, and `guard` is used for mutual exclusion. If we remove `guard`, why will this implementation become unsafe in the presence of multiple consumer threads or multiple producer threads?

(e) Semaphores are introduced, in part, to improve efficiency under contention around critical sections by preferring thread re-schedulling to spinning. Describe a situation in which this might not be the case.

(f) The implementation of semaphores themselves depends on two classes of operations: increment/decrement of an integer, and blocking/waking up threads. As such, semaphore operations are themselves composite operations. What might go wrong if `wait()`'s integer operation and scheduler operation are non-atomic? How about `signal()`?

Q2 Contention

The implementation from Figure 1 suffers from unnecessary contention between producers and consumers due to the shared guard lock. [This is simply a repeat of a question on Sheet S0 and the slide itself.].

(a) Provide pseudo-code for an implementation that eliminates that contention. Done already.

(b) There are two situations where 'in == out' and a third situation where they are not equal. For proof of correctness, we must make a three-way case split and argue that the solution is correct in each case. Make the necessary statements that together convince us that the solution is fully correct. Approximately three sentences required.

```
item_t buffer[N]; int in = 0, out = 0;
spaces = new Semaphore(N);
items  = new Semaphore(0);
guard  = new Semaphore(1);   // for mutual exclusion

// producer threads
while(true) {
  item_t item = produce();
  wait(spaces);
  wait(guard);
    buffer[in] = item;
    in = (in + 1) % N;
  signal(guard);
  signal(items);
}

// consumer threads
while(true) {
  wait(items);
  wait(guard);
    item_t item = buffer[out];
    out =(out+1) % N;
  signal(guard);
  signal(spaces);
  consume(item);
}
```

Figure 1: Pseudo-code for a producer-consumer queue using semaphores.

Q3 Queue Management

Note that there are two forms of queuing in the code of Figure 1: threads may be in a semaphore queue and items may be queuing in the buffer.

(a) Discuss under what conditions the two forms of queuing will tend to get exercised and whether having such queues arising is a good or bad indication (*eg*. is something overloaded?). By 'conditions' we mean the relative rates and burstiness of production and consumption. Mention *backpressure*. Three or four sentences.

(b) (*) Most-recently used (MRU) thread management prefers the most-recently schedulled thread over less-recently schedulled threads.

(*) The MRU policy could be implemented natively by a threads and semaphores library or it it can be implemented using application-level programming on top of an existing library whose policy may not be known or controllable. Discuss which approach to implementing MRU might be preferable.

Q4 Work items or consumers with different priorities.

[Doing all of Q4 and all of Q5 in full will likely take a very long time. Your supervisor will likely recommend you read all parts of Q4 and Q5, making sure you understand what is being asked, but only expect you to complete the full answer to a few sub-parts, taking at most an hour or so.]

In general, a priority can be denoted with a schedulling priority allocated to a producer or consumer thread or it could be indicated per qeueued work item with a label (*eg.* a numeric field in a record). The general situation is potentially complex, perhaps requiring lexicographical or weighted comparisons, so we'll only consider simple cases. For example, a given producer might only generate work items with one fixed priority, which narrows the space down, or consumers might only accept items of a given priority, which again narrows it down. In the next few questions you shall implement some common, simple design patterns using just semaphores for concurrency control.

(a) Consider the case where the items passed from producers to consumers have two different priorities. The priority of a work item is given by a simple predicate (e.g. inspecting a field in the item's header). Consumers are equally able to handle both priorities, but it is preferred that all higher-priority items are passed to consumers before any lower-priority ones.

Provide code in the style of Figure 1 that implements the required queuing discipline. *Hint: you could use two circular buffers, but note that an idle consumer thread can only be blocked on one semaphore*.

Does any potential for priority inversion exist?

(b) Now consider that producers and work items are identical in priority terms but that consumers are statically labelled with three different preferences: 1=HI, 2=MID, 3=LOW. The consumers can handle more than one priority but the preference reflects that some consumers are better or cheaper etc..

The queue implementation in Figure 1 made no effort to prioritise which consumer threads receive items, so now provide code that, when dequeuing an item, hands it to a consumer with the highest priority of those currently available (i.e. idle).

Your implementation should not need to know the number of producers and consumer threads in advance (i.e. it is not know at compile time or explictly stored in your code). You can assume that "peeking" at semaphore values and queue lengths is allowed.

(c) What changes would your code for these solutions require if monitors were used? Is this a good idea? [Feel free to answer all the following questions using monitors if you prefer, provided available parallelism is not reduced.]

(d) (*) Some semaphore libraries provide arrays of semaphores. When you have studied processor architecture in more detail, what consideration should we make about the spacing of entries in such an array? What useful operations could be provided on a complete array of semaphores?

Q5 Work distribution without priorities

(a) Sometimes it is important that consumers are given a fair share of the available work. Round-robin work distribution distributes work fairly across all consumer threads. What possible definitions of 'fairly' are there? Can we tell whether the Figure 1 code implements round-robin? Define round-robin, taking into account that consumption times may vary and not every consumer thread will be idle at every arbitration decision.

Modify the provided code or provide your own pseudo-code for an implementation where we can be sure that service will be round-robin. *Notes:* the number of consumer threads can be fixed at compile time; additional state to record the last-dispatched consumer identity may be required. Round-robin arbitration should only be applied to those that are valid contenders: those that are idle or busy with other work should be disregarded during the arbitration decision.

(b) For each of the work distribution disciplines in this question and the last two, describe a scenario in which that scheme might usefully be used in order to improve performance, and explain why it helps.

Q6 Priority inversion

A system using the generalised producer-consumer implementation in Figure 1 suffers from priority inversion.

(a) A portion of the priority inversion arises from low-priority producers starving high-priority consumers waiting for one another via 'guard'. Is this likely to be a significant portion? (*)What feature found in many implementations of a mutex might make a mutex better for mutual exclusion in this circumstance, rather than a semaphore? (*)Why can't a semaphore provide the same facility?

(b) Another contribution to priority inversion can arise from low-priority consumers starving high-priority work generation through backpressure. Describe a situation where this is happening. How could this problem be exacerbated if the buffer size is small and can it be addressed without using further queues or queues with larger buffers?

(c) A final portion of the priority inversion arises from the *classic* form, where high-priority producers or consumers are starved by medium-priority work elsewhere. Describe two example scenarios. (*)Can these problems be easily mitigated?