yz709-ConcDist-sup1

Question 1

Exercise 1 : (*includes Q0 from s0.pdf*)
List the essential similarities and differences between parallel programming and distributed systems. Are parallel or distributed programs always faster than their sequential equivalent?

- Similarities:
  - Scalability increased horizontally, both aimed at improving performance:
    - In parallel programming, we use either multiple cores to increase the throughput
    - In distributed systems, we set up multiple servers to increase the throughput.
    - Both allow multiple instances (e.g., cores, servers, etc.) to execute tasks simultaneously.
  - Synchronisation of the system is critical:
    - In parallel programming, a master clock synchronised all the processors
    - In distributed systems, a synchronisation algorithm is implemented to ensure all servers run at the same time
- Differences:
  - Parallel programming:
    - Implemented at the application layer or in the operating system
    - It only occurs in a single computer
    - Threads can have shared memory on the same computer

        - Processors communicate with each other via a bus
    - Distributed programming:
        - Implemented at the physical layer or virtually
        - Distributed programming involves some forms of parallel programming to get things done, but parallel programming may only be applied on a standalone computer
        - Each computer has its memory, so no shared memory is involved
        - Computers communicate with each other via the network
        - Better at fault tolerance because we would introduce redundancy in the system to prevent cases when some servers go down
- Parallel or distributed programs may not always be faster than their sequential equivalent because:
    - Managing and allocating tasks to different cores or servers would introduce overheads
    - It is much harder to debug because the final results involve the effects from multiple instances

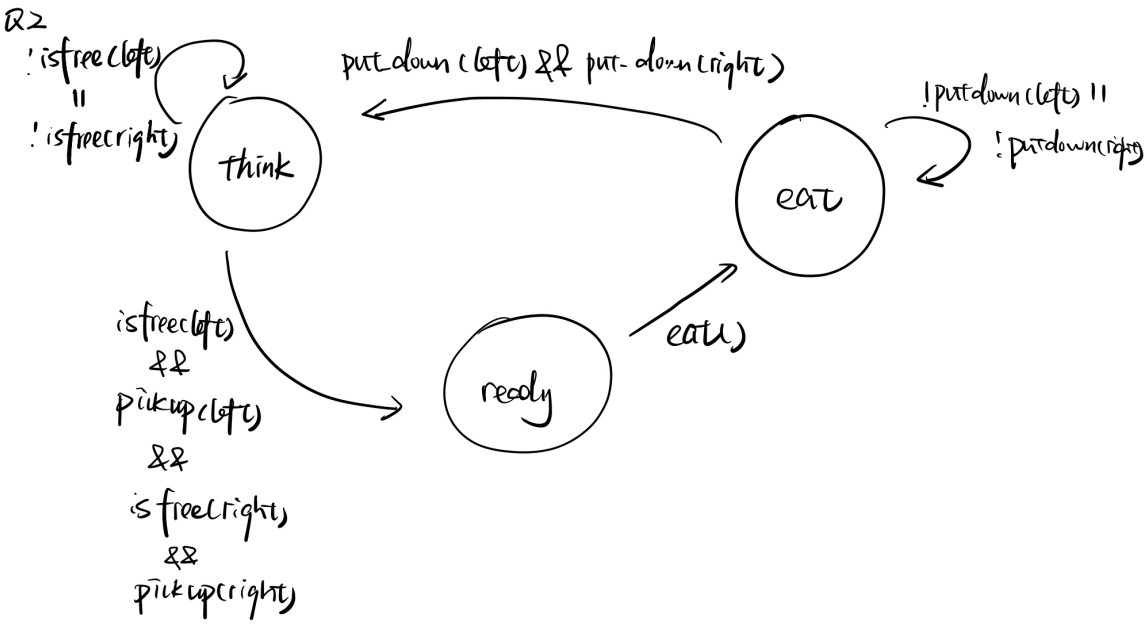
Comments:

Question 2

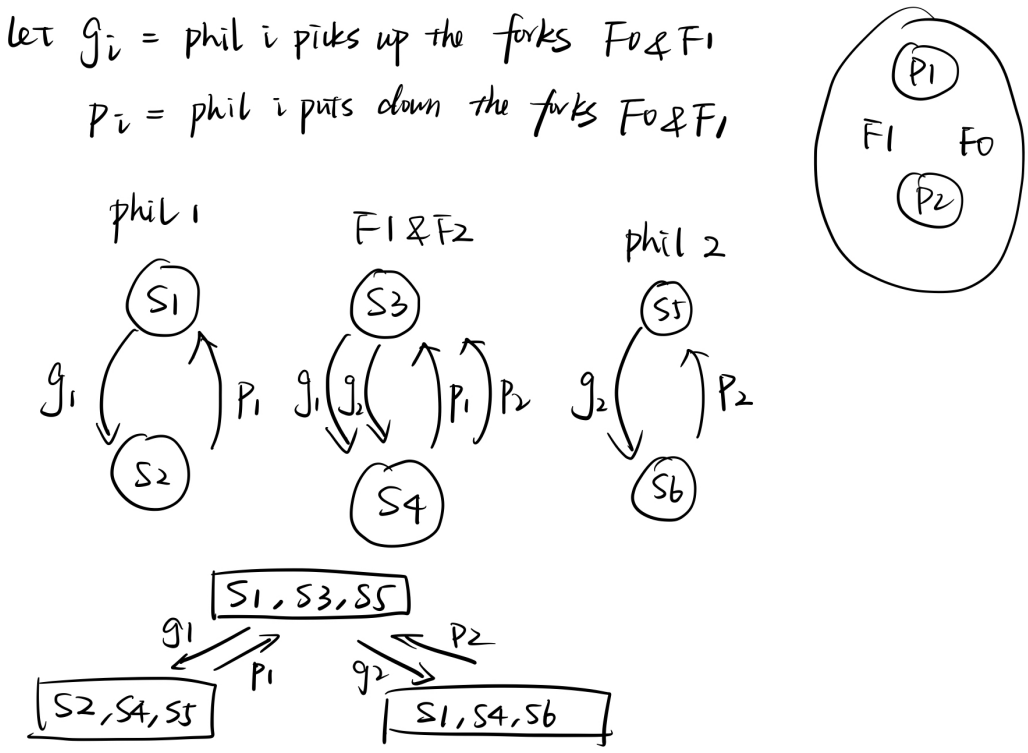
Exercise 2 : (*adapted from Q2 in s0.pdf*)
Draw an *abstract* FSM for the dining philosophers problem. It should have three states and use the following conditions and operations: `is_free(fork)`, `pick_up(fork)`, `put_down(fork)`, and `eat()`. For `fork`, use either `"left"` or `"right"`.

Then, draw the *concrete* FSMs for two philosophers where you replace fork with `"0"` or `"1"` depending on their position on the table. Finally, create the product automaton for the two philosophers and mark deadlock states.

- Abstract FSM:



- Concrete FSM of two philosophers:



Hence states S1, S3, S6 are unreachable – deadlock states
S1, S4, S5
S2, S3, S5
S2, S3, S6
S2, S4, S6

Comments:

Question 3

Exercise 3 :
Study the *Lamport Bakery Algorithm* from the lecture (slide 57/231). Convince yourself that all three lines containing `Enter[...]` are important. For each of the three lines,

1. Assume that line is missing from an implementation.
2. Provide an example that shows a violation of the promised guarantees.

```
void lock(tid) // tid=thread identifier
{
 Enter[tid] = true;
 Number[tid] = 1 + maxj(Number[j]);
 Enter[tid] = false;
 for (j in 0..Ntid-1)
 { while (Enter[j]) continue;
   while (Number[j] && (Number[j], j)<(Number[tid], tid)) continue;
 }
}
void unlock(tid)
{
 Number[tid] = 0;
}
```

- If the first `Enter[tid] = true` is missing:
  - If a thread with `tid = 1` enters but blocks when it gets the number, we are uncertain about its entry in `Enter[1]`.
  - When another thread with `tid = 2` enters, in the for loop, it might skip the thread with `tid = 1` or may not, leading to starvation of the thread with `tid = 2` as it need to wait until `tid = 1` finishes executing.
- If the second `Enter[tid] = false` is missing:
  - Then every `Enter[tid]` would be true; in the for loop, we would skip every `tid`, leading to race conditions if a later thread attempts to get the resources a previous thread is occupying.
- If the third `Enter[j]` is missing:
  - Then if a previous thread has `Enter[j] = true`, maybe because it blocks or progressed very slowly, not skipping it would lead to starvation as we must wait until `j` is finished processing and set `Enter[j] = false`.

Comments:

Question 4

Exercise 4 : (*Q5 from s0.pdf*)
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.

- Replication of states generally undesirable:
  - Concurrency and synchronisation issues as we need to update the states at the same time, avoid inconsistency among replications
  - Greater overheads since we need to keep track of the replications of states
- For the producer/consumer solution, the two semaphores are replicated in both the producer and the consumer thread. In contrast, the two queue pointers are not replicated because the `in` pointer is only used in the producer thread, and the `out` pointer is only used in the consumer thread.
- Too much replication is not a good style, but sometimes we have to have state replication among different threads to synchronise them and avoid race conditions. Hence, it is a good style to use the minimum amount of state replication while implementing a concurrent system.

Comments:

Question 5

Exercise 5 :
In Java programs, we sometimes need global objects (e.g. a central log writer). It is common to only initialise such a global object when it is needed for the first time, and to re-use the same instance as long as the program lives. This design pattern is called "Singleton pattern" and the calling code can get a reference to the global object via the static method `getInstance()`.

Consider the following code and the suggested three implementation variants for `getInstance()`. For each of these, discuss whether it is correct; if not, provide a counter-example. Also, discuss which variant is the most preferred one.

```
1  class MyGlobalSingleton {
2      private static MyGlobalSingleton instance = null;
3
4      public static MyGlobalSingleton getInstance() {
5      ...
6      }
7
8      // Version A (while loop approach)
9      public static MyGlobalSingleton getInstance() {
10         while (instance == Null) {
11             instance = new MyGlobalSingleton();
12         }
13         return instance;
14     }
15
16     // Version B (synchronise the entire method)
17     public static synchronized MyGlobalSingleton getInstance() {
18         if (instance == Null) {
19             instance = new MyGlobalSingleton ();
20         }
21         return instance;
22     }
23
24     // Version C (nested synchronisation)
25     public static MyGlobalSingleton getInstance() {
26         if (instance == Null) {
27             synchronized (MyGlobalSingleton.class) {
28                 if (instance == null) {
29                     instance = new MyGlobalSingleton();
30                 }
31             }
32         }
33     }
34 }
```

- Version A - does not work:
    - Two threads can call the method simultaneously, and in both cases, the `instance` variable would be `Null`, so two instances would be set up, breaking the singleton pattern.
- Version B - works:
    - Only one thread can access the method `getInstance()` at a time; every time, a call to the method would obtain a lock.

    - The method in version B prevents two threads from entering the same method `getInstance()`, but it has performance drawbacks as we need to get a lock even under standard cases when no instance set-up is required.
- Version C - works:
    - It is more efficient than version B because we only obtain a lock when setting up the instance.
    - It is similar to multiple reader, single writer design idea where there is no restriction when readers access the instance because they would not change the instance.

Comments:

Question 6

Exercise 6 : (Q0 from s1.pdf)

(a) Counting 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) Write down two fragments of pseudo-code, to be run in two different threads, that experience deadlock as a result of poor use of mutual exclusion.

(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 (e.g., in producer-consumer) can deadlock.

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

(e) Semaphores are introduced in part to improve efficiency under contention around critical sections by preferring blocking to spinning. Describe a situation in which this might not be the case; more generally, under what circumstances will semaphores hurt, rather than help, performance?

(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()`?

```
1       int buffer[N]; int in = 0, out = 0;
2       spaces = items = guard =
3       new Semaphore(N);
4       new Semaphore (0);
5       new Semaphore(1); // for mutual exclusion
6
7       // producer threads
8       while(true) {
9           item = produce ();
10          wait(spaces);
11          wait(guard);
12          buffer[in] = item;
13          in = (in + 1) \% N;
14          signal(guard);
15          signal(items);
16      }
17
18      // consumer threads
19      while(true) {
20          wait(items);
21          wait(guard);
22          item = buffer[out];
23          out =(out+1) \% N;
24          signal(guard);
25          signal(spaces);
26          consume(item);
27      }
```

- (a):
  - Initialised to 0: there are 0 items in the circular buffer initially
  - Initialised to 1: there is also one processor to be used
  - Initialised to N: there are N spaces in a circular buffer or N slots in a RAM

- (b):
  - If thread 1 reaches line 3 and thread 2 reaches line 10 simultaneously, thread 1 has obtained the lock x while thread has obtained the lock y, hence thread 1 would be waiting for thread 2 to release lock y, and thread 2 would be waiting for thread 1 to release lock x. Thus, we have a deadlock where states are waiting for other members to release a lock.

```
// thread 1
lock(x){
  lock(y); // line 3
  // critical section
  unlock(y);
}

// thread 2
lock(y){
  if (<cond>){ // line 10
    lock(x);
    // critical section
    unlock(x);
  }
}
```

- (c):
  - When thread 1 enters and acquires `sem1`, thread 2 enters at the same time and acquires `sem2`. Then both threads cannot progress as they are waiting for the other resource to be signalled.
  - Hence we have a circular wait, since the other three requirements for a deadlock has already been fulfilled (mutual exclusion, hold-and-wait and no preemption), we now reached the deadlock case.

```
sem1 = new Semaphore(1);
sem2 = new Semaphore(1);

// thread 1
while (true) {
  wait(sem1);
  wait(sem2);
  // critical section
  signal(sem2);
  signal(sem1);
}

// thread 2
while (true) {
  wait(sem2);
```

```
    wait(sem1);
    // critical section
    signal(sem1);
    signal(sem2);
}
```

- (d):
  - If two consumers try to remove items simultaneously or two producers try to put items into the buffer simultaneously, we would have race conditions, and updates may be problematic.
  - For instance, thread 1 enters the consumer thread and reach `item = buffer[out]`, but haven't updated `out = (out + 1) % N` and at the same time, thread 2 enters and reaches `item = buffer[out]` while the out still holds the old value.
- (e):
  - Blocking threads has more overhead than spinning because the operating system has to keep track of all calls to wait and signal semaphores. With a multi-core processor and many locks only held for a short period, if we constantly block threads, the time blocking them may be greater than the time they hold the lock, which implies it is not worth blocking them but allows them to spin.
  - Semaphore may lead to priority inversion; if two threads with different priorities are all waiting for a resource and the lower priority one comes first, it might be waked up first when the resource is available.
  - The order of wait and signal operations needs to be executed correctly to avoid deadlock, so it is not practical for large scale and may lead to the loss of modularity.
- (f):
  - The non-atomic operation would lead to inconsistency, which further introduces bugs. `wait()` needs to decrement the integer semaphore and execute or block the thread; while `signal()` needs to increment the integer semaphore and wake up threads.
  - Suppose we have a buffer of N spaces, thread 1 calls `wait(spaces)`. If `wait(spaces)`'s integer operation and scheduler operation are not atomic, then after we decremented the integer of the semaphore, another thread (e.g., thread 2) might enter and invoke the `signal(spaces)` operation and

increment the integer of the semaphore; when the previous thread resumes, the semaphore integer hasn't changed, but the scheduler doesn't know that, which implies the data we filled into the buffer would be lost afterwards.

- If we have a buffer of N spaces with no items inside, threat 1 calls `signal(items)` to fill one space. But if `signal()`'s integer operation and scheduler operation are not atomic, then after we incremented the integer semaphore, another thread (e.g., thread 2) enters and calls `wait(items)`, since the `items` semaphore is updated to 1, but there are no items exist in the buffer, we encountered a bug.

> 💡 Comments:

Question 7

Exercise 7 :
Consider the following scheduling algorithms for deciding which customer should be picked next:

Round-robin Tasks are distributed following a given order (wrapping around to the first when each customer has been served).

Priority-based Tasks are distributed first to high-priority consumers, then to lower-priority customers.

Most-recently-used Tasks are given to the most recently finished customer.

For each of these scheduling algorithms:

1. Describe how you can implement them and what data structures you need. A high-level discussion is fine, but you can also provide pseudocode.
2. Describe a scenario where it is a good choice.

- Round-robin:
  - A queue of consumers, each one has the same number of quantum.
  - Dequeue the head element, execute it and enqueue it back to the queue if it requires more CPU time.
  - A good choice if we have consumers of the same types.
- Priority-based:

    - Each consumer will have a corresponding priority number; we have a priority queue for the consumers ordered based on their priorities.
    - Dequeue the head element, execute it and preempt it by putting it back to a suitable position in the priority queue if another higher priority one requests execution.
    - A good choice when we have a range of different consumers, and if we want to increase responsiveness, we could give the consumers involving user interactivity a higher priority. But that leads to starvation of lower priority consumers.
- Most-recently-used:
    - Each task will have a corresponding timestamp; we have a priority queue for the consumers ordered based on their timestamp, with recent ones having a higher priority.
    - Dequeue the head element using a FIFO execution style. We cannot preempt any consumers before they finished executing.
    - A good choice when we want to make the common case fast, as the most recent consumers would be the ones we execute frequently. But that leads to starvation of less recently used consumers.

Comments: