Chalmers | Göteborgs Universitet
Sandro Stucki, Computer Science and Engineering

Principles of Concurrent Programming TDA384/DIT391

Wednesday, 21 August 2019, 14:00–18:00

(including example solutions)

Exam set and supervised by: Sandro Stucki (sandros@chalmers.se, 076 420 8639)
Examiner: K. V. S. Prasad (prasad@chalmers.se)

Q1. Filing cabinet *(10p)*

The following pseudo code models a filing cabinet with the capacity to hold 100 files. (See Appendix A.1 for a full Java version of the code.)

```
1  class LockedCabinet {
2    int[] files = { 1, 1, ..., 1 };  // one hundred '1's
3    Lock lock = new ReentrantLock();
4
5    void checkout(int index) {
6      lock.lock();
7      while (files[index] <= 0) { lock.unlock(); lock.lock(); } // busy wait
8      files[index] = files[index] - 1;
9      lock.unlock();
10   }
11
12   void store(int index) {
13     lock.lock();
14     files[index] = files[index] + 1;
15     lock.unlock();
16   }
17 }
```

The field `files[`i`]` indicates the status of the i-th file: `files[`i`]` = 1 means that the file is currently stored in the cabinet, `files[`i`]` = 0 means the file has been checked out. All 100 files are initially stored in the cabinet. The methods `checkout()` and `store()` are used to checkout and return the file at position `index`. If a thread tries to checkout a file that is not in the cabinet, it waits for the file to be returned. To avoid multiple threads accessing the cabinet simultaneously, all read and write instructions to `files` are guarded by a `lock`.

(Part a) Why does the **`while`** loop on line 7 repeatedly acquire and release the `lock`? What could go wrong if the calls to `lock()` and `unlock()` in the loop were removed? Describe a scenario where this would happen. *(2p)*

Solution of 1.a: Without these instructions, calls to `checkout()` could deadlock. Assume, for example, a system with two threads t and u, where thread t has just successfully checked out the file at position `index`, so that `files[index] == 0`, and neither thread holds the `lock`. Next, thread u tries to checkout the file at the same position. It calls `lock.lock()`, obtains the lock, and enters the busy loop, waiting for the file to be returned. But because u is holding the `lock`, no other thread can return the file, so the value of `files[index]` remains 0, and u remains stuck in the loop. If t tries to return the file, it blocks indefinitely on `lock`, and neither thread can make progress.

(Part b) Assume that two threads t and u concurrently access the filing cabinet. Thread t calls `store(0)`, while thread u simultaneously calls `checkout(0)`. Can any *data races* occur in this situation? If so, give an example. Otherwise, explain why the code is free from data races. *(2p)*

Solution of 1.b: there are no data races. Recall the definition of a data race given in the course:

> A *data race* occurs when two concurrent threads
>
> - access a shared memory location,
> - at least one access is a write,
> - the threads use no explicit mechanism to prevent the accesses from being simultaneous.

In this case, the concurrent threads t and u try to access the shared memory location `files[0]`. Thread t writes to `files[0]` on line 14, thread u reads `files[0]` on line 7 and writes to it on line 8. However, all three accesses are explicitly guarded by `lock`, so no data races can occur.

A semaphore-based implementation

Consider the following alternative, semaphore-based implementation of a filing cabinet.

```
1 class SemaphoreCabinet {
2   Semaphore[] files = {
3     new Semaphore(1), ..., new Semaphore(1)  // one hundred semaphores
4   };
5   void checkout(int index) { files[index].acquire(); }
6   void store(int index)    { files[index].release(); }
7 }
```

Instead of an array of integers and a lock, `SemaphoreCabinet` uses an array of *semaphores* to keep track of files. The value of the semaphore `files[`i`]` indicates the status of file i. A value of 1 means the file is in the cabinet, 0 means it has been checked out. To checkout a file, a thread must acquire the corresponding semaphore; to return a file, it releases the semaphore. (See Appendix A.2 for a full Java version.)

(Part c) Assume we are given one instance `lc = new LockedCabinet()` of the lock-based implementation, and one instance `sc = new SemaphoreCabinet()` of the sema-

phore-based implementation. What is the maximum number of threads that can call `store()` on the lock-based instance `lc` concurrently without blocking? What about the semaphore-based instance `sc`? Justify your answers. *(2p)*

Solution of 1.c: the maximum number of concurrent, non-blocking calls to `store()` on `lc` is one because the `LockedCabinet` class uses a single lock to guard all calls to `store()`. Because the class `SemaphoreCabinet` uses individual semaphores for each file instead, every file can be released independently, so the maximum number of concurrent non-blocking calls to `sc.store()` is 100.

Assume that there are a hundred threads, each executing the following code:

```
for (int j = 0; j < 10000; ++j) { c.checkout(id); c.store(id); }
```

where `c` is either an instance of `LockedCabinet` or of `SemaphoreCabinet`, and `id` is the thread identifier (`id` $= 0, 1, \ldots, 99$).

(Part d) On a system with $N = 100$ CPU cores, which choice of implementation for `c` will make the program complete faster, `LockedCabinet` or `SemaphoreCabinet`? How does your answer change if there is only $N = 1$ CPU core? For each case, give a rough estimate of the speedup/slowdown of the semaphore-based implementation compared to the lock-based implementation ($1\times$, $10\times$, $100\times$, $\ldots$). Justify your answers. *(4p)*

Solution of 1.d: the semaphore-based solution is about $100\times$ faster for $N = 100$ CPU cores; for $N = 1$ core, the two solutions will have roughly the same performance.

Since there is no code executed between the calls to `checkout()` and `store()`, the threads will spend all their time either acquiring and releasing locks/semaphores, or waiting (if they are blocked). As explained in Part c, the lock-based solution allows only one thread to execute `store()` concurrently without blocking, and the same is true for `checkout()`. In a system with a hundred CPU cores, all threads can in principle run in parallel, but most will be blocked if the lock-based implementation is used (contention is high). The semaphore-based implementation, on the other hand, allows all threads to run in parallel (there is no contention). Hence the semaphore-based implementation should be up to $100\times$ faster.

If there is only $N = 1$ core, there is no potential for parallelism and most threads will spend their time waiting, even if they are not trying to acquire a lock or semaphore. The two implementations should therefore not perform significantly differently in that scenario.

Code and transition table for Q2 and Q3

The code below models a pair of threads t and u collaborating to solve two tasks. See Q2 for a detailed explanation of the code, and Appendix A.3 for the full Java program.

```
boolean[] done = { false, false }; Lock[] lock = { ... };
```

thread t		thread u	
1	`while (true) {`	`while (true) {`	1
2	`  lock[t].lock();`	`  lock[u].lock();`	2
3	`  if (!done[t]) {`	`  if (!done[u]) {`	3
4	`    done[t] = true;`	`    done[u] = true;`	4
5	`  } else {`	`  } else {`	5
6	`    lock[u].lock();`	`    lock[t].lock();`	6
7	`    if (!done[u]) done[u] = true;`	`    if (!done[t]) done[t] = true;`	7
8	`    else break; // exit loop`	`    else break; // exit loop`	8
9	`    lock[u].unlock();`	`    lock[t].unlock();`	9
10	`  }`	`  }`	10
11	`  lock[t].unlock();`	`  lock[u].unlock();`	11
12	`} // t terminates`	`} // u terminates`	12

Below is an incomplete state transition table for this program. Each state consists of the current values of the program counters (pc_t, pc_u), and the arrays `done` (d_t, d_u) and `lock` (l_t, l_u) for each thread. The value of l_i is the thread holding `lock[i]`. In the table, pc_t and pc_u only take the values 2, 6, 9, 11 or 12; $pc_t = 2$ means the *next* line executed by t is 2.

	current state s = (pc_t, pc_u, d_t, d_u, l_t, l_u)	$s'(t)$ = next state if t moves	$s'(u)$ = next state if u moves
s_1	(2, 2, F, F, –, –)	(11, 2, T, F, t, –) = s_2	(2, 11, F, T, –, u) = s_3
s_2	(11, 2, T, F, t, –)	(2, 2, T, F, –, –) = s_5	(11, 11, T, T, t, u) = s_4
s_3	(2, 11, F, T, –, u)	(11, 11, T, T, t, u) = s_4	(2, 2, F, T, –, –) = s_6
s_4	(11, 11, T, T, t, u)	(2, 11, T, T, –, u) = s_7	(11, 2, T, T, t, –) = s_8
s_5	(2, 2, T, F, –, –)	(6, 2, T, F, t, –) = s_9	(2, 11, T, T, –, u) = s_7
s_6	(2, 2, F, T, –, –)	(11, 2, T, T, t, –) = s_8	(2, 6, F, T, –, u) = s_{10}
s_7	(2, 11, T, T, –, u)	(6, 11, T, T, t, u) = s_{11}	(2, 2, T, T, –, –) = s_{13}
s_8	(11, 2, T, T, t, –)	(2, 2, T, T, –, –) = s_{13}	(11, 6, T, T, t, u) = s_{12}
s_9	(6, 2, T, F, t, –)	(9, 2, T, T, t, t) = s_{14}	(6, 11, T, T, t, u) = s_{11}
s_{10}	(2, 6, F, T, –, u)	(11, 6, T, T, t, u) = s_{12}	(2, 9, T, T, u, u) = s_{15}
s_{11}	(6, 11, T, T, t, u)	no move	(6, 2, T, T, t, –) = s_{16}
s_{12}	(11, 6, T, T, t, u)	(2, 6, T, T, –, u) = s_{17}	no move
s_{13}	(2, 2, T, T, –, –)	(6, 2, T, T, t, –) = s_{16}	(2, 6, T, T, –, u) = s_{17}
s_{14}	(9, 2, T, T, t, t)	(11, 2, T, T, t, –) = s_8	no move
s_{15}	(2, 9, T, T, u, u)	no move	(2, 11, T, T, –, u) = s_7
s_{16}	(6, 2, T, T, t, –)	(12, 2, T, T, t, t) = s_{19}	(6, 6, T, T, t, u) = s_{18}
s_{17}	(2, 6, T, T, –, u)	(6, 6, T, T, t, u) = s_{18}	(2, 12, T, T, u, u) = s_{20}
s_{18}	(6, 6, T, T, t, u)	no move	no move
s_{19}	(12, 2, T, T, t, t)	no move	no move
s_{20}	(2, 12, T, T, u, u)	no move	no move

Q2. Collaborating threads – state transitions *(20p)*

The pseudo-code given on page 4 models a pair of threads t and u collaborating to solve two tasks. Each task is represented by a Boolean `done[i]` indicating whether task i has been completed. Each thread is assigned a task and attempts to complete that task first. Once it has completed its task, it attempts to steal and complete the task assigned to the other thread. If it discovers that both tasks have been completed already, it terminates.

To ensure that only one thread works on a given task, the array `done` is guarded by a pair of locks. Before a thread can read or modify `done[i]`, it must acquire `lock[i]`.

Page 4 also contains an incomplete state transition table modeling the behavior of the two threads. To keep the table small, only program positions relevant to the concurrent behavior of the program are tracked: lines where a lock is acquired (2 and 6) or released (9 and 11), and the end of the threads (line 12). Remember that $pc_t = 2$ means the *next* line executed by t is 2.

Your task is to complete the state transition table and prove or disprove that the code is free from *deadlocks* and *starvation*.

(Part a) Fill in the missing fields in rows s_5, s_9, s_{11}, s_{14}, s_{16}, s_{18}, s_{19} and s_{20} of the transition table. Make sure your solution is consistent with the names given for the missing fields in rows s_2, s_7, s_{13} and s_{17}. Don't write directly into the table on page 4. Instead, write down the complete rows as $s_i : (X, Y, \dots),\ (U, V, \dots)\dots$ on a separate sheet of paper with the rest of your solutions. *(8p)*

Solution of 2.a: see completed table on page 4.

(Part b) Does the table contain any final states? If so, list all of them, otherwise explain what the absence of final states means for the program. *(1p)*

Solution of 2.b: there are no final states. Only states where *both* threads have terminated are final states. The states s_{19} and s_{20} are candidates because, in each case, one of the threads has terminated. But, in both cases, the other thread is blocked (rather than terminated), so neither state is final. This means that at least one of the threads does not terminate (and hence the program does not terminate either).

(Part c) Are any of the fields in the state tuples redundant? Is it possible to remove any of the variables $pc_t, pc_u, d_t, d_u, l_t, l_u$ without collapsing two previously distinct states? *(2p)*

Solution of 2.c: the fields l_t and l_u are redundant since their information is contained in the values of pc_t and pc_u. We have

$$l_t = \begin{cases} - & \text{if } pc_t = 2 \text{ and } pc_u \in \{2, 6, 9\}, \\ u & \text{if } pc_t = 2 \text{ and } pc_u \in \{11, 12\}, \\ t & \text{otherwise,} \end{cases}$$

and similarly for l_u.

Deadlock

A thread is *blocked* in state s if it cannot move from s even though it has not yet terminated. For example u is blocked in s_{12}. A *deadlock* occurs if the program reaches a state where both threads are blocked.

(Part d) Can the program deadlock? Prove your answer from the state transition table. If your answer is yes, also give a trace (i.e. a sequence of states starting from the initial state s_1) that exhibits the deadlock. *(2p)*

Solution of 2.d: there is a deadlock, as witnessed by the state s_{18}. In this state, neither thread can move because each thread is waiting for a lock held by the other thread. Any trace ending in s_{18} exhibits the deadlock. For example, $(s_1, s_2, s_4, s_7, s_{13}, s_{16}, s_{18})$.

Starvation

Remember that a thread is *starving* if it is blocked indefinitely while other threads can make progress or terminate. We say a thread has become *immortal* if it is starving while all other threads have terminated.

(Part e) Can either of the threads become immortal? Prove your answer from the state transition table. If your answer is yes, also give a trace (i.e. a sequence of states starting from the initial state s_1) that exhibits starvation. *(2p)*

Solution of 2.e: both threads can become immortal. Thread u is immortal in state s_{19}, thread t is immortal in state s_{20}. In each case, the immortal thread is waiting on a lock held by the terminated thread. Any trace that ends in one of these states exhibits starvation. For example, $(s_1, s_2, s_4, s_7, s_{13}, s_{16}, s_{19})$.

Releasing locks on termination

Assume that line 8 in the code for threads t and u is changed from

```
8  else break; // exit loop
```

to

```
8  else { lock[t].unlock(); lock[u].unlock(); break; }
```

As a consequence, the values of states s_{19} and s_{20} must be updated, and 3 new states have to be added to the transition table.

...	...	...	...
s_{19}	(12, 2, T, T, –, –)	no move	(12, 6, T, T, –, u) = s_{21}
s_{20}	(2, 12, T, T, –, –)	(6, 12, T, T, t, –) = s_{22}	no move
s_{21}	(12, 6, T, T, –, u)	no move	(12, 12, T, T, –, –) = s_{23}
s_{22}	(6, 12, T, T, t, –)	(12, 12, T, T, –, –) = s_{23}	no move
s_{23}	(12, 12, T, T, –, –)	no move	no move

(Part f) Fill in the missing fields in rows s_{21}–s_{23} in the updated state transition table. Make sure your solution is consistent with the updated rows s_{19} and s_{20}. *(3p)*

Solution of 2.f: see table above.

(Part g) Do your answers to Parts d and e change after the code update to line 8? If so, prove your new answers from the updated state transition table. *(2p)*

Solution of 2.g: deadlocks can still occur, but neither thread can become immortal now. The update has not changed state s_{18}, which still witnesses a deadlock (and all traces leading to s_{18} remain unchanged). The states s_{19} and s_{20}, which previously witnessed the immortality of u and t, no longer do so because neither thread is blocked in the updated states. In fact, there are no longer any states in the updated table where one thread is blocked while the other thread has terminated, so there are no immortal threads.

Q3. Collaborating threads – logical reasoning *(8p)*

Consider again the collaboration problem described in its original form on page 4 (*not* the updated version from Q2.f). See Q2 for an explanation of the code.

Here in Q3, you must argue from the program, not from the state transition table (though you may seek inspiration from it). You get full credit for correct reasoning, whether you use formal logic, everyday language, or a mixture. Formulas and logical laws make your argument concise and precise, and help you keep track. With everyday language, be careful not to be fuzzy, or to mistake wishful thinking for proof. Appendix B reviews briefly the notation of propositional logic and linear temporal logic.

In your reasoning you may assume the following basic invariants about t and u:

1. the threads are only ever observed at the following locations:
   $pc_t \in \{2, 6, 11, 9, 12\}$ and $pc_u \in \{2, 6, 11, 9, 12\}$;
2. the `done` flags are Boolean-valued: either d_t or $\neg d_t$ and similarly for d_u.
3. the possible values of the locks l_t and l_u are $l_t \in \{-, t, u\}$ and $l_u \in \{-, t, u\}$;
4. at any given time, the program counters, and locks have exactly one value:
   $pc_t = i \wedge pc_t = j$ if and only if $i = j$, and similarly for pc_u, l_t, and l_u.

Consider the following propositions:

- $B_t \equiv (pc_t = 2 \wedge l_t \neq -) \vee (pc_t = 6 \wedge l_u \neq -)$,
- $B_u \equiv (pc_u = 2 \wedge l_u \neq -) \vee (pc_u = 6 \wedge l_t \neq -)$,
- $W \equiv (pc_t = 6 \wedge pc_u = 6)$,
- $D \equiv B_t \wedge B_u$,
- $F_t \equiv B_t \rightarrow \Diamond(pc_t = 12)$.

(Part a) Describe the propositions B_t, D and F_t in words. Which concurrency properties do D and F_t correspond to? *(3p)*

Solution of 3.a: B_t says that the thread t is trying to acquire `lock[`t`]` or `lock[`u`]` but the respective lock (l_t or l_u) is already locked. In other words, B_t says that t is *blocked*. Similarly, B_u says that u is blocked. D says that t and u are simultaneously blocked, which means that there is a *deadlock*. F_t says that, if t is currently blocked, it will eventually terminate, i.e. move to line 12, which implies *starvation freedom* for t.

(Part b) Explain briefly why the following invariants hold at any point during the execution of the program:
(I1) whenever $pc_t = 6$, we must have $l_t = t$;
(I2) whenever $pc_u = 6$, we must have $l_u = u$. *(1p)*

Solution of 3.b: (I1) holds because, for thread t to arrive at line 6, it must have executed line 2, and hence it must hold the lock `lock[`t`]`. (I2) holds by a similar argument for thread u.

(Part c) Show that, at any point in the execution of the program, if W holds, so does D. (*Hint:* you may use the invariants from Part b.) *(2p)*

Solution of 3.c: here is an example proof by contradiction.

Assume W holds, but D does *not*, that is, either t or u can make a move. We proceed by case-analysis on these two options (t moves, u moves).

First assume t can make a move, then

$$\neg B_t \quad \equiv \quad (pc_t \neq 2 \vee l_t = -) \wedge (pc_t \neq 6 \vee l_u = -).$$

From W we know that $pc_t = 6 \neq 2$, so we can simplify the above to

$$(\mathrm{T} \vee l_t = -) \wedge (\mathrm{F} \vee l_u = -) \quad \equiv \quad \mathrm{T} \wedge l_u = - \quad \equiv \quad l_u = -$$

which means `lock[`u`]` is unlocked. But from W we know that $pc_u = 6$, so $l_u = -$ contradicts (I2).

Similarly, if we assume that u can move, we arrive at a contradiction to (I1). Hence, D must hold, after all.

(Part d) Give a counterexample to F_t, i.e. describe a program run where B_t becomes true, but $pc_t = 12$ can *never* become true afterwards. *(2p)*

Solution of 3.d: this amounts to giving a scenario where t starves. One such scenario is for u to execute two iterations of the loop and terminate before t starts its first iteration. In so doing, u first completes its own task, then steals t's task and, in the process, locks both `lock[`u`]` and `lock[`t`]`. At that point, $l_t = l_u = u$, $pc_t = 2$, and $pc_u = 12$, so B_t holds (t is blocked). Since u has terminated and t is blocked, neither thread can move, and $pc_t = 12$ will never come true.

Q4. Fork/join parallelism *(10p)*

The following recursive Java method computes the sum of the values stored in an array `arr` between index `start` and index `end` (excluding `arr[end]`).

```java
int sum(int[] arr, int start, int end) {
  if (end <= start) return 0;
  else              return arr[start] + sum(arr, start + 1, end);
}
```

Based on this method, Billy has implemented the following naive fork-join task for summing up numbers in an array. (See Appendix A.4 for a full code listing.)

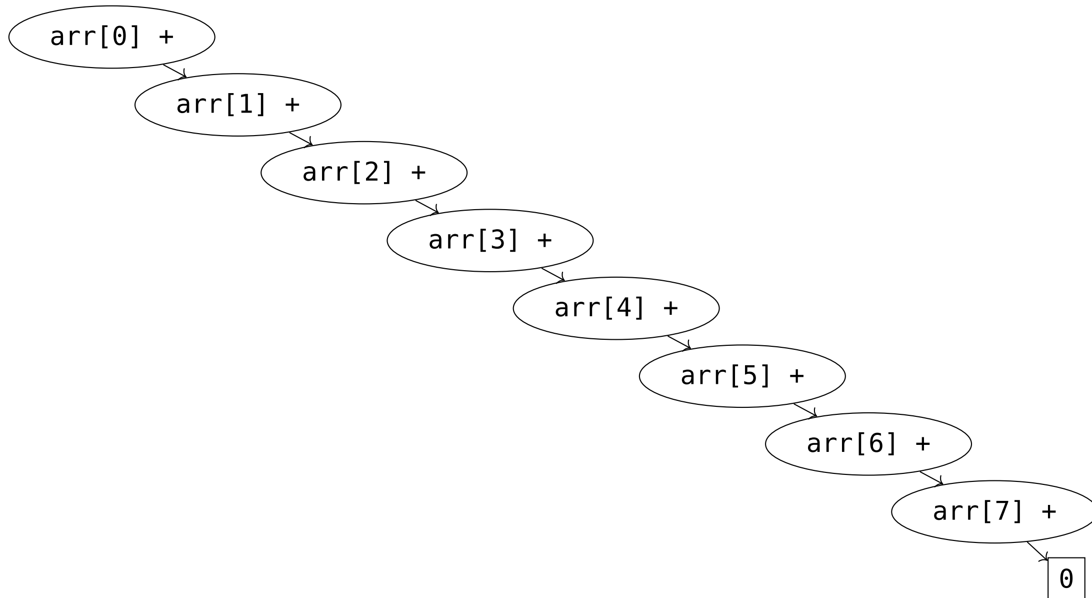
```java
class SumTask extends RecursiveTask<Integer> {
  protected int[] arr;
  protected int start, end;

  SumTask(int[] arr, int start, int end) {
    this.arr = arr; this.start = start; this.end = end;
  }

  @Override public Integer compute() {
    if (end <= start) return 0;
    else {
      SumTask t = new SumTask(arr, start + 1, end);
      t.fork();
      int s = t.join();
      return arr[start] + s;
    }
  }
}
```

(Part a) Draw the dependency graph for a run of `SumTask(arr, 0, 8)` assuming `arr` has exactly 8 elements. Each node should represent a task instance, with leaf nodes representing the base cases (where `end <= start`). Edges should represent parent-child relationships between tasks. How many nodes does the graph contain? *(2p)*

Solution of 4.a: there are $8 + 1 = 9$ nodes.



(Part b) What is the approximate runtime of `SumTask(arr, 0, 8)` assuming each task takes about one unit of time to perform its computation? What is the maximum number of tasks that can be executed in parallel in this implementation (excluding parent tasks waiting for a child task to finish)? How can these numbers be inferred from the dependency graph? *(4p)*

Solution of 4.b: the total runtime is approximately 9 time units – the depth of the dependency graph – and at most one task can run in parallel – the width of the dependency graph.

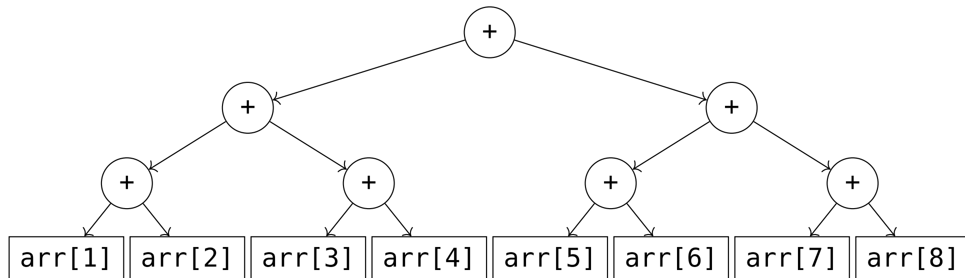
After taking a class on concurrent programming, Billy adjusts the `compute()` method of `SumTask` as follows:

```
1  @Override public Integer compute() {
2    if      (end <= start)     return 0;
3    else if (end == start + 1) return arr[start];
4    else {
5      int mid = (start + end) / 2;
6      SumTask t1 = new SumTask(arr, start, mid);
7      SumTask t2 = new SumTask(arr, mid, end);
8      t1.fork();
9      t2.fork();
10     int s1 = t1.join();
11     int s2 = t2.join();
12     return s1 + s2;
13   }
14 }
```

(Part c) What is the total number of tasks created by the updated implementation when running an instance of `SumTask(arr, 0, 8)`? Assume this instance is running on a machine with 8 or more CPU cores. Does the new implementation improve the total runtime

compared to the naive one? If so, why? Justify your answers by drawing the dependency graph for `SumTask(arr, 0, 8)` using the new implementation of `compute()`. *(4p)*

Solution of 4.c: there are 15 tasks (= number of nodes in the dependency graph). The total runtime is now approximately 4 time units (= depth of the graph) which is a clear improvement. This is because more tasks can run in parallel now: the maximum number of parallel tasks improved from 1 to 8 (= width of the graph).



Q5. Event counter *(12p)*

The following Erlang function implements the event handler of a so-called *event counter*.

```
1  handler(Goal, Done, Clients) when Done == Goal ->  % Goal reached
2    [C ! {done, N} || {C , N} <- dict:to_list(Clients)],
3    handler(Goal, 0, dict:new());                     % reset server state
4  handler(Goal, Done, Clients) ->                    % Goal not yet reached
5    receive
6      {event, From} ->
7        From ! ok,
8        NewClients = dict:update_counter(From, 1, Clients),
9        handler(Goal, Done + 1, NewClients)
10   end.
```

An event counter is a server that receives and counts `event` messages from clients. The server keeps track of the total number of received events using the argument `Done`, and of the number of events received from particular clients using the dictionary `Clients`. When `Done` reaches a predefined number `Goal`, the server sends a `done` message to all its clients and resets its state (except for the `Goal` argument, which never changes).

Clients interact with the server by calling the functions `event/0` and `wait/0`.

```
11 event() ->
12   counter ! {event, self()},
13   receive ok -> ok end.
14
15 wait() ->
16   receive {done, N} -> N end.
```

Clients notify the server of events by calling `event()`. When a client calls `wait()`, it blocks until it receives a `done` message from the server.

See Appendix A.5 for a full code listing.

(Part a) List all the message types processed by the server. For each type, describe

- the *name* of the message (e.g. "**ok**");
- the *sender* and *receiver* of the message (e.g. "from server to client");
- the *payload* of the message (e.g. "the sender PID" or "no payload");
- what *causes* a message to be sent (e.g. "a client calls the `event` function"). *(3p)*

Solution of 5.a: There are three types of messages:

- `{done, N}` – sent from the server to clients when `Done` has reached `Goal`; `N` is the number of events registered from the respective client since the last server reset;

- `ok` – sent from the server to clients to acknowledge the reception of an event (no payload);

- `{event, From}` – sent from a client with PID `From` to the server when the client calls `event()`.

(Part b) What is the meaning of the number returned by a call to `wait()`? *(1p)*

Solution of 5.b: `wait()` returns the number of events sent from the calling client to the server since the last reset of the server state (i.e. since the server was created, or since the last reached `Goal` events).

Assume an event counter with `Goal=2` has just been initialized and two processes `P` and `Q` are spawned from the main process using the following code:

```
Main = self(),  % remember PID of main process
P = spawn(fun() -> event(), event(), Main ! {self(), wait()} end),
Q = spawn(fun() -> event(),          Main ! {self(), wait()} end),
event().        % event generated by the main process
```

Here are two possible sequences of messages received by the `Main` process after `P` and `Q` terminate:

1. `{P,2}`, `{Q,1}`.
2. `{Q,1}`, `{P,1}`.

(Part c) For each sequence, give a corresponding sequence of messages received by the *server* that explains the messages seen by `Main`. Your answer should be of the form $(M_1, D_1), (M_2, D_2), \ldots$, where each M_i is a message received by the server and D_i is the value of `Done` before M_i is processed by the server. *(2p)*

Solution of 5.c: Here are two example sequences (others are possible):

1. (`{event,P}`, `0`), (`{event,P}`, `1`), (`{event,Q}`, `0`), (`{event,Main}`, `1`).

2. (`{event,P}`, `0`), (`{event,Q}`, `1`), (`{event,Main}`, `0`), (`{event,P}`, `1`).

Resetting the server

Sometimes it is necessary to reset the event counter. For this, we introduce a new type of message – `reset` – that can be sent to the server from any other process. `reset` messages are handled as follows.

By the server: when the server receives a message of the form `{reset, P}` from process `P`, it first responds to `P` with an `ok` message. It then sends a `reset` message to all its clients (those listed in the `Clients` dictionary). Finally, it resets its internal state (`Done` and `Clients`).

By clients: clients that receive a `reset` message while waiting for a `done` message from the server (in a call to `wait()`) exit the `wait` function and return `error` to indicate that the server was reset without reaching its goal.

You are now asked to write a short piece of code. You may use Erlang, pseudo code, or a mixture of both. Syntax details, such as punctuation, are not important, but an experienced Erlang programmer should be able to understand your code. Your code *must use functional style*: use recursion, list comprehensions or higher-order functions instead of mutable state or loops. You are not expected to write more than 20 lines of code in total.

(Part d) Implement the reset mechanism according to the above specification. Describe the necessary changes to the functions `wait/0` and `handler/3`. You don't need to repeat parts of the code that do not change. Describe also the new function `reset/0` used to reset the server. Calls to `reset()` should block until an acknowledgment from the server has been received. *(6p)*

Solution of 5.d: here is an example implementation:

```
wait() ->
  receive
    {done, N} -> N;
    reset     -> error   % add clause handling 'reset' messages
  end.

reset() ->
  counter ! {reset, self()},
  receive ok -> ok end.

handler(Goal, Done, Clients) when Done == Goal -> ... % no changes
handler(Goal, Done, Clients) ->  % Goal not yet reached
  receive
    {event, From} -> ... % no changes
    {reset, From} ->
      From ! ok,
      [C ! reset || {C, _} <- dict:to_list(Clients)],
      handler(Goal, 0, dict:new())  % reset server state
  end.
```

Q6. Lock-free filing cabinet *(10p)*

Recall the semaphore-based filing cabinet implementation `SemaphoreCabinet` from Q1 (page 2). See Appendix A.2 for the full Java version.

```
1 class SemaphoreCabinet {
2   Semaphore[] files = {
3     new Semaphore(1), ..., new Semaphore(1)  // one hundred times
4   };
5   void checkout(int index) { files[index].acquire(); }
6   void store(int index)    { files[index].release(); }
7 }
```

Here in Q6, you will be asked to implement a lock-free filing cabinet and to compare its performance to the original `SemaphoreCabinet`.

Remember that the `AtomicInteger` class from the Java standard library provides a `compareAndSet` method with the following signature.

```
boolean compareAndSet(int expectedValue, int newValue)
```

(Part a) Describe a lock-free implementation of `SemaphoreCabinet`. Your solution should use a CAS (compare-and-set) synchronization primitive, such as Java's `AtomicInteger`, but no semaphores or locks. Describe all fields and methods of the class that need to be added, changed or removed. You may use either Java code or pseudo code to do so. *(6p)*

Solution of 6.a: replace the array of `Semaphores` with an array of `AtomicIntegers` and use compare-and-set (CAS) operations when updating the file values:

- for `store()`, repeatedly attempt incrementing the array cell at `index` until the CAS operation succeeds;
- for `checkout()`, keep reading the array cell (without locking) until it is positive, then attempt decrementing it using CAS – if CAS fails, start over.

Here is an example implementation in Java:

```
// No more locks or semaphores!
private AtomicInteger[] files = new AtomicInteger[100];
public LockFreeCabinet() {
  for (int i = 0; i < 100; ++i) { files[i] = new AtomicInteger(1); }
}

void checkout(int index) {
  int value;
  do {
    value = files[index].get();
  } while (value <= 0 || !files[index].compareAndSet(value, value - 1));

// An alternative solution using two nested loops:
```

```
//
//  do {
//    do { value = files[index].get(); } while (value <= 0);
//  } while (!files[index].compareAndSet(value, value - 1));
}

void store(int index) {
  int value;
  do {
    value = files[index].get();
  } while (!files[index].compareAndSet(value, value + 1));
}
```

Let c be a filing cabinet that is accessed by N different threads executing the following code:

```
1  for (int j = 0; j < 100; ++j) {
2    // switch between index = 0 and index = 1 in even/odd iterations
3    int index = j % 2;
4    c.checkout(index);
5    doWork();
6    c.store(index);
7  }
```

Assume that the method `doWork()` performs a non-trivial computation (>10^6 CPU cycles). Consider the following scenarios:

(Scenario 1) There are 32 CPU cores and $N = 32$ threads.

(Scenario 2) There are 2 CPU cores and $N = 100$ threads.

(Part b) Which of the implementations do you expect to perform better in Scenario 1, the semaphore-based one from Q1 or your lock-free version? In Scenario 2? Justify your answers for both scenarios. *(4p)*

Solution of 6.b: the lock-free version should perform better in Scenario 1, whereas the semaphore-based version from Q1 should perform better in Scenario 2.

Contention is high in both scenarios since there are at least an order of magnitude more threads than there are files (only the two files at `index` 0 and 1 are accessed). Therefore, most threads trying to checkout a file will be blocked. The semaphore-based implementation invokes the scheduler when this happens, resulting in considerable overhead (>1000 cycles), but once a thread is blocked it is moved to the wait queue where it will occupy minimal CPU resources. In the lock-free implementation, on the other hand, threads trying to checkout a file block by busy-waiting, which causes high CPU usage but minimal scheduling overhead. The CAS operation, while not free, cause negligible overhead when compared to scheduling (a few hundred CPU cycles).

In Scenario 1, there are as many CPU cores as there are threads, so high CPU usage (due to busy-waiting) will have minimal impact on overall runtime, but scheduling over-

head remains costly (>1000 cycles). Hence, we should expect the lock-free version to perform slightly better in Scenario 1.

In Scenario 2, there are few CPU cores and busy-waiting threads will block the execution of other threads until they are preempted by the scheduler or `doWork()` completes in another thread. This will result in significant overhead ($> 10^6$ cycles, if we have to wait for `doWork()` to complete). Thus, we expect the semaphore-based version to perform significantly better in Scenario 2.

Appendix A. Full code listings

Appendix A.1. Code for Q1

```
1  import java.util.concurrent.*;
2  import java.util.concurrent.locks.*;
3
4  class LockedCabinet {
5    final private int[] files = new int[100];
6    final private Lock lock = new ReentrantLock();
7    public LockedCabinet() {
8      for (int i = 0; i < 100; ++i) { files[i] = 1; }
9    }
10   public void checkout(int index) {
11     lock.lock();
12     try {
13       while (files[index] <= 0) { lock.unlock(); lock.lock(); }
14       files[index] = files[index] - 1;
15     } finally { lock.unlock(); }
16   }
17   public void store(int index) {
18     lock.lock();
19     try { files[index] = files[index] + 1; } finally { lock.unlock(); }
20   }
21   public static void main(String[] args) {
22     LockedCabinet lc = new LockedCabinet();
23     Thread[] ts = new Thread[100];
24     for (int i = 0; i < ts.length; ++i) {
25       final int id = i;
26       ts[i] = new Thread() {
27           @Override public void run() {  // thread code for Q1
28             for (int j = 0; j < 10000; ++j) {
29               lc.checkout(id); lc.store(id);
30             }
31           }
32         };
33       ts[i].start();
34     }
35     for (int i = 0; i < ts.length; ++i) {
36       try { ts[i].join(); } catch (InterruptedException ie) {}
37     }
38   }
39 }
```

Appendix A.2. Code for Q1 and Q6

```java
1 import java.util.concurrent.*;
2
3 class SemaphoreCabinet {
4   final private Semaphore[] files = new Semaphore[100];
5   public SemaphoreCabinet() {
6     for (int i = 0; i < 100; ++i) { files[i] = new Semaphore(1); }
7   }
8   public void checkout(int index) {
9     try { files[index].acquire(); } catch (InterruptedException ie) {}
10  }
11  public void store(int index) {
12    files[index].release();
13  }
14  public static void main(String[] args) {
15    SemaphoreCabinet sc = new SemaphoreCabinet();
16    Thread[] ts = new Thread[100];
17    for (int i = 0; i < ts.length; ++i) {
18      final int id = i;
19      ts[i] = new Thread() {
20          @Override public void run() {  // thread code for Q1
21            for (int j = 0; j < 10000; ++j) {
22              sc.checkout(id); sc.store(id);
23            }
24          }
25        };
26      ts[i].start();
27    }
28    for (int i = 0; i < ts.length; ++i) {
29      try { ts[i].join(); } catch (InterruptedException ie) {}
30    }
31  }
32 }
```

Appendix A.3. Code for Q2 and Q3

```
1 import java.util.concurrent.*;
2 import java.util.concurrent.locks.*;
3
4 class TaskThread extends Thread {
5   private int id;
6   public static boolean[] done = { false, false };
7   public static Lock[] lock = { new ReentrantLock(), new ReentrantLock() };
8
9   TaskThread(int id) { this.id = id; }
10
11   @Override public void run() {
12     int me = id;
13     int other = 1 - id;
14     while (true) {
15       lock[me].lock();
16       if (!done[me]) {
17         done[me] = true;
18         System.out.println("Thread " + me + " completed task " + me);
19       } else {
20         lock[other].lock();
21         if (!done[other]) {
22           done[other] = true;
23           System.out.println("Thread " + me + " completed task " + other);
24         } else {
25           break; // exit loop
26         }
27         lock[other].unlock();
28       }
29       lock[me].unlock();
30     }
31     System.out.println("Thread " + me + " done.");
32   }
33
34   public static void main(String[] args) {
35     TaskThread t0 = new TaskThread(0);
36     TaskThread t1 = new TaskThread(1);
37     t0.start(); t1.start();
38     try {
39       t0.join(); t1.join();
40     } catch (InterruptedException i) {}
41   }
42 }
```

Appendix A.4. Code for Q4

```
1  class SumTaskNaive extends RecursiveTask<Integer> {
2    protected int[] arr;
3    protected int start, end;
4
5    static int sum(int[] arr, int start, int end) {
6      if (end <= start) return 0;
7      else              return arr[start] + sum(arr, start + 1, end);
8    }
9
10   SumTaskNaive(int[] arr, int start, int end) {
11     this.arr = arr; this.start = start; this.end = end;
12   }
13   @Override public Integer compute() {
14     if (end <= start) return 0;
15     else {
16       SumTaskNaive t = new SumTaskNaive(arr, start + 1, end);
17       t.fork(); int s = t.join();
18       return arr[start] + s;
19     }
20   }
21   public static void main(String[] args) {
22     final int N = 200; int[] numbers = new int[N];
23     for (int i = 0; i < N; ++i) { numbers[i] = i; }
24     SumTaskNaive t = new SumTaskNaive(numbers, 0, N);
25     int res = ForkJoinPool.commonPool().invoke(t);
26     System.out.println("Sum: " + res);
27   }
28 }
29
30 class SumTask extends SumTaskNaive {
31   SumTask(int[] arr, int start, int end) { super(arr, start, end); }
32   @Override public Integer compute() {
33     if      (end <= start)     return 0;
34     else if (end == start + 1) return arr[start];
35     else {
36       int mid = (start + end) / 2;
37       SumTask t1 = new SumTask(arr, start, mid);
38       SumTask t2 = new SumTask(arr, mid, end);
39       t1.fork();          t2.fork();
40       int s1 = t1.join(); int s2 = t2.join();
41       return s1 + s2;
42     }
43   }
44 }
```

Appendix A.5. Code for Q5

```
1  -module(counter).
2  -export([init/1,event/0,wait/0,reset/0]).
3
4  % start the event counter process with a given Goal
5  init(Goal) ->
6    register(counter, spawn(fun () -> handler(Goal, 0, dict:new()) end)).
7
8  % notify the counter about an event
9  event() ->
10   counter ! {event, self()},
11   receive ok -> ok end.
12
13 % wait until the counter reaches its goal
14 wait() ->
15   receive {done, N} -> N end.
16
17 % synchronously reset the counter
18 reset() ->
19   todo.
20
21 % the server's event handler
22 handler(Goal, Done, Clients) when Done == Goal ->  % Goal reached
23   [C ! {done, N} || {C , N} <- dict:to_list(Clients)],
24   handler(Goal, 0, dict:new());  % reset server state
25 handler(Goal, Done, Clients) ->                    % Goal not yet reached
26   receive
27     {event, From} ->
28       From ! ok,
29       NewClients = dict:update_counter(From, 1, Clients),
30       handler(Goal, Done + 1, NewClients);
31     {reset, From} ->
32       todo
33   end.
```

Appendix B. Linear Temporal Logic (LTL) notation

1. An atomic proposition such as $q2$ (process q is at label $q2$) *holds for* a state s if and only if process q is at $q2$ in s.

2. Let ϕ and ψ be formulas of LTL. Formulas are either atomic propositions, or are built up from other formulas using the following operators: $\neg$ for "not", $\vee$ for "or", $\wedge$ for "and", $\rightarrow$ for "implies", $\Box$ for "always", and $\Diamond$ for "eventually". A convenient abbreviation is ϕ iff ψ (i.e., ϕ if and only if ψ) for $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$.

   These operators have the obvious meanings, but two differ from what might be your interpretation of the names. First, $\phi \vee \psi$ ("ϕ or ψ") is false iff both ϕ and ψ are false. This is an "inclusive or", so $\phi \vee \psi$ is also true if both ϕ and ψ are true. Second, $\phi \rightarrow \psi$ ("ϕ implies ψ") is false iff ϕ is true and ψ is false. So, in particular, $\phi \rightarrow \psi$ is true if ϕ is false. The meanings of the operators $\Box$ and $\Diamond$ are defined below.

3. A *path* is a possible future of the system, a possibly infinite sequence of states, each reachable from the previous state in the path. A state s *satisfies* formula ϕ if every path from s satisfies ϕ.

   A path π satisfies $\Box\phi$ if ϕ holds for the first state of π, and for all subsequent states in π. The path π satisfies $\Diamond\phi$ if ϕ holds for some state in π.

   Note that $\Box$ and $\Diamond$ are duals:

$$\Box\phi \equiv \neg\Diamond\neg\phi \qquad \text{and} \qquad \Diamond\phi \equiv \neg\Box\neg\phi.$$