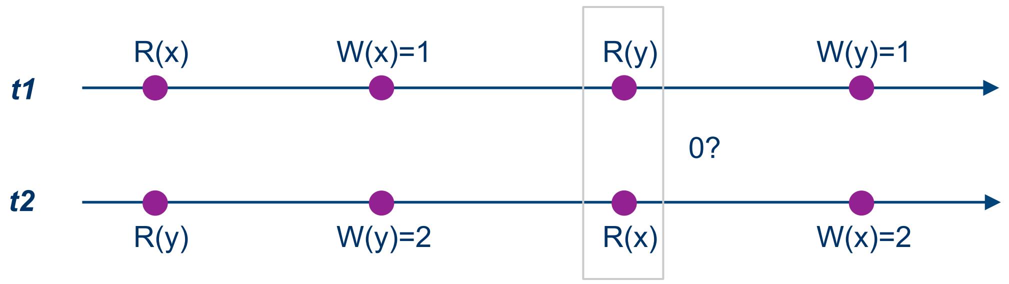
Qualifying exam: operating systems, 1/5/2015

Part 1. Consistency

Consider the program fragment below left. Assume that the program containing this fragment executes **t1**() and **t2**() on separate threads running on separate cores. They run concurrently as depicted in the timeline below. Assume a modern multicore system with a shared memory and lock() and unlock() primitives. Answer (a), (b), (c) below, noting any additional assumptions you make.

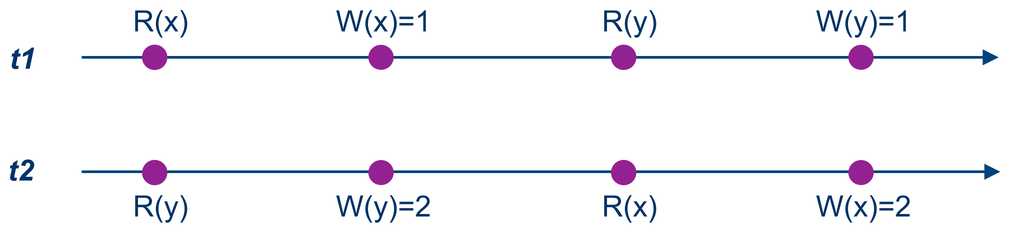
```
int x=0, y=0;

t1() {
    int i, j;
    i = x;    /* R(x)  */
    x = 1;    /* W(x) */
    j = y;    /* R(y)  */
    y = 1;    /* W(y) */
    …
}

t2() {
    int i, j;
    i = y;     /* R(y)  */
    y = 2;     /* W(y) */
    j = x;     /* R(x)  */
    x = 2;     /* W(x) */
    …
}
```



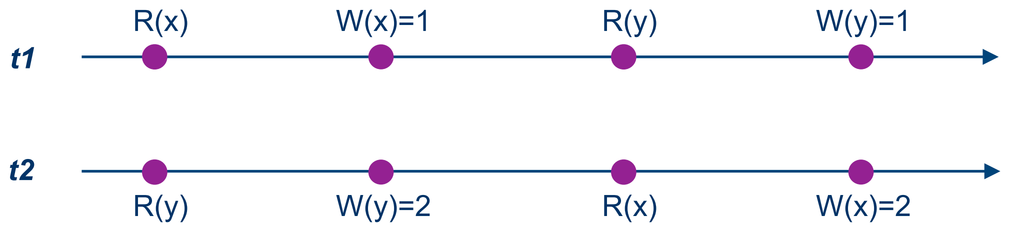
(a) Is there any **sequentially consistent** execution of **t1** and **t2** that could yield **R(y) = 0 in t1** and **R(x) = 0 in t2**? Why or why not? The point here is to demonstrate an understanding of sequential consistency.

Part 1. (Continued)

(b) Annotate the timeline below to show how to use locks to ensure a **sequentially consistent** execution with respect to the operations on **x** and **y**. Illustrate the resulting partial order of events by adding directed arcs between events to indicate Lamport **happens-before** relationships, and annotating the events with values for logical clocks and vector clocks.



(c) Annotate the timeline below to show how to use locks to ensure a **serializable** execution of **t1** and **t2** (in the sense of atomic transactions). Illustrate the resulting partial order of events as in (b).



Part 2. Concurrency

A **monitor** is a classic abstraction for concurrency control using the four operations on the right. Monitors appear in modern programming languages including Java and C#, and as linked mutexes and condition variables in Modula-2 and pthreads. The pseudo-code on the right is a flawed implementation of monitors using semaphores. (P is down, and V is up). This question asks you to summarize what is wrong with the code.

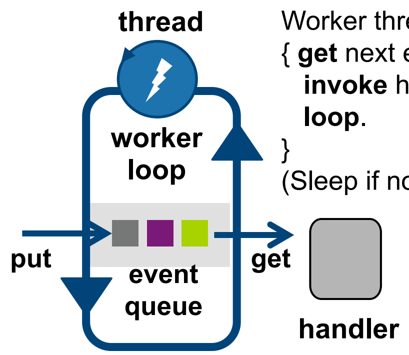
Write a list of **four distinct correctness properties** that this solution violates even when used correctly. Outline a fix for each one. Feel free to mark the code.

```
semaphore mutex(0);        /* init 0 */
semaphore condition(1);    /* init 1 */

acquire() {
    mutex.p();
}

release() {
    mutex.v();
}

wait() {
    condition.p();
}

signal() {
    condition.v();
}
```

Part 3. Threads and scheduling

This part asks you to write code to synchronize a standard event/request queue (a thread pool). Any kind of pseudocode is fine as long as its meaning is clear. You may assume standard data structures, e.g., linked lists: don't write code for those.

Threads place event records on the queue using the **put** method. (E.g., **put** might be called by a network connection handler.) A pool of worker threads **get** event records from the queue and a call handler routine to process each event. When a handler completes, the worker calls **get** again for the next event. The workers sleep if the queue is empty. **Note**: I am only asking you to code up the **put** and **get** methods, and **not** any thread/code that calls those methods.



Worker thread **do**:
{ **get** next event;
  invoke handler;
  loop.
}
(Sleep if no events.)

(a) Implement **put** and **get** using monitors (or mutex + condition variable).

```
put (Event e)
{








}
```

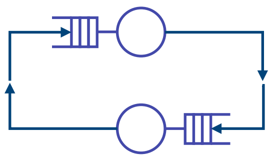
```
get() returns Event e
{








}
```

Part 3. (Continued)

(b) Modify your answer to (a) to reduce **average response time**. Suppose there are three types of incoming events with different average service demands: the handler for type A events takes one time unit to execute, type B events take 2 units, and type C events take 3 units. You may assume each event record is tagged with exactly one type code (A, B, or C). Your solution should be free from **starvation** and **deadlock**. Code is optional: if you can explain your solution without writing code, that is sufficient.

Part 4. Performance

Suppose I have a service that runs on a server S and receives requests from a network. Each request on S reads a randomly chosen piece of data from a disk attached to S, and also does some computing. On average, a request arriving when S is idle spends T time units computing on the CPU and T time units waiting for data from the disk.



Now suppose I redeploy the service on a new server S'. S' is the same as S, except that the CPU is 20% faster and S' has two disks, each containing a copy of the data.

(a) Sketch the throughput and mean response time of S and S' as the request arrival rate increases. Sketch the graphs, label the axes, and annotate key features of the graphs. What are the peak throughputs of S and S' at saturation? Note any other assumptions you need to answer the question.

Throughput	**Mean response time**

Part 4. (Continued)

(b) Now suppose I add memory to S' so that it can cache half of the data stored on the disks. What impact does this change have on the peak throughput? What impact does it have on the utilization of the disks? Again, note any additional assumptions made in your answer.

Part 5. Virtualization

What is the purpose of **copy-on-write** (COW)? Outline a scenario in which a virtual memory system uses COW. List a sequence of events that occurs to trigger a COW operation and to complete the operation. Outline (or sketch) the data structures that the OS needs to recognize a triggering scenario and to complete the operation.