Part 1. Multicore memory models

Consider the program fragment below left. Assume that the program containing this fragment executes \texttt{t1()} and \texttt{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.

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

Suppose that \texttt{t1:R(y)} returns 0. Then \texttt{t1:R(y) \rightarrow t2:W(y)=2}, since sequential consistency (SC) ensures a total order on accesses to memory, at least to the same location. But SC also ensures that all memory accesses in each thread execute in program order, so \texttt{t1:W(x)=1 \rightarrow t1:R(y)}. Similarly, \texttt{t2:W(y)=2 \rightarrow t2:R(x)}. Since \rightarrow is transitive, this implies that \texttt{t1:W(x)=1 \rightarrow t2:R(x)}. Since there are no other intervening write operations on \texttt{x}, this implies that \texttt{t2:R(x)} must return 1.

For this question I accepted almost any coherent argument that made clear that either read could yield 0, but that it is a contradiction for both to yield 0. For full credit the argument should make some reference to both properties of SC (since both are required complete the argument.
(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 $y$.

See discussion on next page.

(c) How many possible **synchronization orders** exist for the example above? Pick one and illustrate it on the timeline above by drawing arrows for the **happened-before** relationships. If you prefer you may illustrate it on the timeline below.

See discussion on next page.
Multicore memory models ensure sequentially consistent executions programs that are “properly labeled” (fully synchronized), i.e., that use locks correctly.

For 1b it is necessary to show how to use locks correctly. You do that by putting lock() and unlock() operations on the timeline. Also be clear about how many mutexes are involved.

Any locking discipline is OK, as long as all accesses to a given location (x or y) are under the scope of the same lock. The simplest answer is to use a single lock and lock everything down: both t1() and t2() lock() at the start and unlock() at the end. The second-least-complicated answer is to use two locks, one for x and one for y. Then t1() locks x for its first half, unlocks, then locks y for its second half, and t2() is similar.

Then 1c asks you how many synchronization orders are possible using your locking discipline in 1b. If you gave the simplest answer (one global lock) to 1b, then 1c has a simple answer: two. Either t1 or t2 acquires the global lock first. If you used two locks, then the answer is four: either t1 or t2 goes first for lock x, and either t1 or t2 goes first for lock y. All 2*2=4 combinations are possible.

I got a number of answers with factorials and such that apparently assumed that the synchronization order is a total order. But it is not: it is a partial order.

To complete 1c it is necessary to draw all of the happened-before (à) arrows. You need six arrows, three on each line, to show that à preserves program order. And you need one à from one line to the other each time a lock is passed from one thread to the other.
Part 2. Threads and concurrency control

This problem asks you to write the core of a program to issue and service disk requests. As always: “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.”

There are $N$ requester threads that issue requests and one servicer thread to service the requests. Requesters issue requests by placing request objects on a queue. The queue is bounded to at most $MAX$ requests: a requester waits if the queue is full. Each requester is permitted to have at most one pending request. The servicer handles requests only when the queue is full or all living requesters have a pending request.

Write procedures for the requesters and servicer. You will need to use some control concurrency: use monitors, mutexes, condition variables, and/or semaphores (your choice).

```c
int N, MAX; /* global pre-initialized constants */
Queue rq; /* a queue with put/get operations */

/* Requester threads call this for each request */
void request(RequestObject robj)
{
    // This problem appeared in lab #1, and the solution is the same.
}

/* Code for the servicer thread */
void servicer()
{
}
```
Part 3. CPU protection and OS structure

In a Dune-enabled Linux system, as described in the Dune paper, the various pieces of software run in one of four different CPU protection levels (states) depicted in the diagram below. This part asks several questions relating to this system and diagram.

(a) Annotate the diagram with words or phrases to indicate what kind of software runs in each of the four states, and why: what are the advantages of each state as a context to run that software?

Linux processes that don’t use Dune.
- Fully isolated (sandbox, lockbox).
- No access to privileged machine functions.
- Safe!

Host kernel (hypervisor or VMM).
- Full access to machine functions.
- Fully privileged.
- Controls everything.

Untrusted code running within Linux processes that run in “Dune mode”.
- Runs on ring 3.
- Fully isolated (sandbox, lockbox) from the rest of the Dune process, and from everything else.
- No access to privileged machine functions.

Linux processes that run in “Dune mode”.
- Runs on ring 0.
- Fully isolated (sandbox, lockbox).
- Access to privileged machine functions.
- Safe! But also empowered to manage their own faults, interrupts, page tables, etc.
- Access to system call interface in host kernel via “hypercalls”.

Some people answered this question with reference to virtual machine systems, which use the same architectural foundations. That was generally OK.
(b) Now annotate the diagram again to indicate transitions (i.e., control transfers) between states that occur in the system as it runs. Add words or phrases to label those transitions and indicate what kind of events trigger those state transitions.
Part 3. CPU protection and OS structure (continued)

(c) Software running in each of the four CPU protection states runs within a virtual address space (VAS) defined by page table maps stored in memory. Annotate the figure again as follows. First, for each of the four states, indicate how many VASs use that state (e.g., 0, 1, N), and what events cause the creation and destruction of a VAS for that state. Second, for each state, draw arrows to the other state(s) whose software sets up and maintains the page tables for those VASs.
Part 4. Library OS

Please keep answers short and direct. “Answers are graded on content, not style.” It is not necessary to write essays with complete sentences: use lists, phrases, or whatever is easy and expressive. Please do not restate any part of the question.

The paper on the Drawbridge library OS mentions that previous library OS systems (e.g., Exokernel) focus on “providing applications with fine-grained, customized control of hardware resources, such as page tables, network packets, and disk blocks.” We also discussed how Scheduler Activations uses this concept to enable a high-performance thread system as a library (“ULTS”): the user-level thread library and kernel interact to allocate CPU cores (“processors”) to the application.

(a) Summarize what events might cause an application to gain or lose a processor core in a Scheduler Activations system. How does the thread library learn of changes in the number of cores allocated to the application?

- events to gain: kernel decides to allocate a core to this ULTS. Could be prompted by any transition to kernel, e.g., another process exits, or a timer interrupt + preemption. The ULTS uses a syscall to ask for more cores, and if it has asked, it gets one if and when the kernel decides to give it.

- events to lose: kernel decides to take one away. (see above)

Note that the application does NOT lose a core just because one of its threads blocked, or faulted, or exited. This is a fundamental point about Scheduler Activations. Similarly, it does not gain a core just because it created a thread, or a sleeping thread woke up.

- learn of changes: kernel upcalls to ULTS on one of its cores, passing any context information as needed.
Part 4. Library OS (continued)

Drawbridge, the paper says, has “differing goals (security, host independence, and migration)”, and therefore its design differs from Exokernel and other library OS systems that emphasized per-application control over raw hardware resources. Instead, its focus is to offer “higher-level abstractions” that “make it easier to share underlying host OS resources such as buffer caches, file systems, and networking stacks with the library OS”. In particular, Drawbridge applications can “share resources including the screen, keyboard, and mouse”.

(b) Discuss how Dune balances these apparently competing sets of goals. How does Dune’s design exemplify each of these structuring philosophies? How does it differ from them?

I was looking for anything that showed you understand Dune and that was responsive to the question.

The answer I expected was: this is *exactly* what Dune is designed for:

“per-application control over raw hard resources”: YES! When a process is running in Dune mode, it has access to “privileged CPU features” (per the title of the paper). It can create any kind of OS model or environment using the cores it has.

“share underlying host resources”: YES! A process running in Dune mode can call the host kernel through the usual Linux system call interface.

“Differs”: well, resources held by a Dune process are virtualized, and can be taken away without warning or notification. That is true of Scheduler Activations as well, but in a more limited way: if the application is still active, it is always notified of a change in its allocation.
Part 4. Library OS (continued)

(c) Is Singularity a “library OS”? Why or why not? How is it same? How is it different?

I was looking for anything that showed you understand Singularity and that was responsive to the question.

The answer I expected is: (briefly)

same: processes (SIPs) typically run in the same hardware protection domain as the OS (kernel). So a system call is very fast, just like a library call.

different: Singularity has a protection structure that closely mirrors conventional operating systems. It has a distribution of functions that closely mirrors conventional operating systems. What is different is the reliance on language protection, which (typically) removes the need for protecting the operating system “the hard way” (using hardware-based memory protection).

Notes on grading: my marks are hard to read. Sorry.
- A “check” means I generally liked you answer.
- A horizontal line means you did not give me what I wanted, but you said something else that was worth something close to half credit.
- A back slash means you said something that I couldn’t understand as a correct answer or even a partial credit answer.
- There may also be various pluses and minuses.
- All problems are equally weighted.
- All parts of each problem are equally weighted.
- Grading is not an exact science. Feel free to ask. Feel free to check my math.
This page intentionally left blank. Feel free to use it.