Consider a server with the following concurrency structure. As incoming requests arrive, a network handler queues them for processing by a fixed set of servicer threads. Each request \( r \) has a field \( r.\text{op} \) whose value identifies the type of operation requested. Each servicer processes queued requests for a specific operation type \( t \) in sequence. Each completed request is processed by one servicer.

You are asked to write code for two interacting methods (procedures or routines) at the heart of this server. You may assume common data structures (e.g., queues); do not write code for those.

The two methods:
1. The network handler: `incoming(Request \( r \))`. Queue an incoming request \( r \) for processing.
2. Request dispatcher: `Request \( r = \text{getRequest(Operation } t)\)`. Dispatch a queued request matching operation \( t \) to the calling servicer.

Each servicer runs for a fixed operation type \( T \), and executes a simple loop: `do { \( r = \text{getRequest}(T); \text{processAndReply}(r); \) }

These methods may execute concurrently on multiple cores. Servicers should wait for a request if none are queued. Write a version of the two methods for each of three approaches to synchronization:

A. Restricted monitors only. You may use monitors (locks/mutexes) and conditions (condition variables), but each monitor may have at most one associated condition. For example, the Java language has this restriction.
B. Monitors only. Each monitor may have multiple conditions. This may permit a more efficient implementation than for P1.A.
C. Semaphores only.
P2. Server performance

Consider a server S that functions as described in P1. Suppose there are three operation types: t1, t2, t3. Requests have a mean CPU service demand of D milliseconds, and require no I/O. S has 12 cores, each of which can run one thread at a time. These questions in this part all have short answers, but the last few may take 2-3 sentences. Feel free to illustrate with a figure.

1. Suppose S has two servicers for each operation type. What is the maximum server utilization for a request mix (workload) containing requests of types t1 and t2 only?
2. How many servicers should S create to handle requests at maximum throughput, regardless of the request mix?
3. Estimate the maximum request throughput of S with the suitable number of servicers (as in P2.2).
4. Given the suitable number of servicers, if S receives requests at 50% of the maximum throughput, estimate the average server utilization.
5. Given the suitable number of servicers, and a workload in which requests arrive at some rate \( \lambda \) and requests of type t2 are twice as frequent as requests of type t1 or t3, estimate what share of CPU time is spent on requests of type t2.
6. Suppose S has only one servicer for requests of type t1, and that that thread is blocked awaiting a request 50% of the time under some given workload. Estimate the throughput for requests of type t1 in that scenario.
7. Assuming the scenario of P2.6 meets the assumptions of standard queuing models (e.g., exponentially distributed service demand and inter-arrival times), estimate the mean response time for requests of type t1 in that scenario.
8. Suppose the operator of S wants to ensure that S expends no more than one-third of server CPU time on each of the three operations, regardless of workload. How many servicer threads should it create for each request type to meet this objective?
9. The approach to performance isolation in P2.8 has a key drawback for unbalanced request mixes. Describe the problem with reference to a specific example.
10. If S receives requests at a rate higher than its maximum throughput over a period of time (regardless of how it is configured), what impact does this have on memory utilization? How should the network handler routine address this situation?
11. Briefly enumerate some internal factors that might result in performance that differs from your “estimate” for an idealized server.
P3. Close to the metal: Meltdown

Meltdown is a vulnerability (a security flaw) in many existing CPUs. When discovered in 2018, it enabled an untrusted user process on standard operating systems (OS) to read all of machine memory. This part walks through some elements of a software exploit (attack code) for Meltdown. Each question (except P2.8) requires a short answer (1-2 sentences) with suitable terminology.

1. Give an example of a harm that might result if a user process can read all of machine memory.
2. What OS mechanism typically prevents a user process from reading all of machine memory?
3. A Meltdown exploit requires that the kernel is mapped into the virtual address space even when a core is executing in user mode. Why might an OS leave its kernel mapped in user mode, i.e., what benefit does it provide?
4. A Meltdown exploit is most effective when all of machine memory is mapped into kernel virtual memory (VM). Why might an OS establish such a mapping, i.e., what benefit does it provide?
5. A typical Meltdown exploit executes a loop containing an instruction sequence like the one shown below, incrementing the KVA (kernel virtual address) on each iteration to read kernel VM one byte at a time. The load instruction at line 1 is disallowed: a core in user mode cannot complete a read on kernel VM. How does an OS disable such access?
6. What happens when the disabled load instruction fails: what/where is the next instruction to execute? (A Meltdown exploit might use any of several approaches to suppress or recover from this event; ignore all that.)
7. How large is the array beginning at the address value of the symbol _base, i.e., what is the highest offset from _base that the store instruction at line 3 could address and write?
8. The Meltdown vulnerability results from a behavior of speculative execution in certain processors, in which the load and store affect the CPU cache state before the CPU detects that the load at line 1 must fail. Assuming they do, sketch an algorithm for the attack program to determine the value of the byte at KVA. You may assume a register CCR that contains the value of the core’s cycle counter, and a constant threshold value T such that a load that takes more than T cycles is a cache miss.

```
1. load byte @KVA → R0 ; attempt a load from a Kernel Virtual Address
2. shl R0, 0xc          ; register shift-left
3. store 42, @[_base+R0] ; store to array starting at _base in user VM
```
P4. Virtual memory and the hardware/software interface

Modern 64-bit machines have large address spaces. Moreover, these address spaces may be sparse (big holes). The page tables that map a large and sparse virtual memory may have a substantial memory cost.

1. Illustrate a typical page table structure that enables reasonably compact maps. Draw a picture. Illustrate how fields of a virtual address might be used to index the page table.

2. Consider a small C program that simply prints some values and then exits. How much machine memory is consumed for page tables to map the virtual memory of a process to run this program? Give a concrete estimate and justify it.