Principles of Computer Systems
Final Exam

20-Dec-2018

This exam has 5 questions, totaling 100 points. You have 105 minutes to answer them, which means you earn about 1 point per minute of work – please consider spending on each question no more minutes than the number of points attributed to it.

If you exit the room during the exam, you will have to turn in your exam, and you will not be permitted to return to the room until the end of the exam. Please plan accordingly.

You are allowed to have any amount of printed material you like (books, papers, notes) but no laptops, tablets, cellphones, etc. are permitted during the exam. You must take the seat assigned by the course staff and present your CAMIPRO card to the staff upon request.

Do not open the exam until instructed to do so.

Your name: _____ SCIPER: _____

Question 1 *(10 points)*

Say you are designing a link-layer technology for a new type of wireless link, and you need to pick the maximum transfer unit (MTU) size. Describe the steps you would go through to choose the MTU.

Answer:

The inputs provided are the probability of one bit being corrupted and a model of how subsequent bits are affected in the transmission. Based on this, compute the probability that a sequence of N bits (i.e., a packet of size N bits) gets corrupted. Solving for that probability being less than the target yields a maximum N, and then we set MTU to this value.

How to choose the target? Consider the maximum tolerable delay, which determines the maximum acceptable number of retransmissions, which then determines the target probability of corruption.

Question 2 *(15 points)*

The TCP congestion control algorithm uses packet loss to decide when there is network congestion. A friend of yours says this is a layering violation.

(a) Is your friend right or not? Explain.

(b) Describe a scenario where relying on packet loss to decide when there is network congestion does not work well and explain why. What would be a better approach to do congestion control in the scenario you described?

Answer:

a. Yes this is a layering violation, because TCP's congestion control algorithm makes the assumption that congestion leads to packet loss, and this is a detail specific to layers underneath TCP. In fact, depending on which link and/or network layer is used, this might not even hold.

   No, this is not a layering violation. We assume that congestion always leads to packet loss independent of the underlying link layer. So, there is no knowledge assumed.

   (5 points for either of the two, if justified well enough)

b. One example scenario where packet loss does not work is lossless network fabrics. For instance, in RoCE (a network protocol for doing RDMA over an Ethernet network), the Ethernet network is typically configured via Ethernet flow control or priority flow control to make it lossless. In such cases, TCP can deal with congestion by leveraging ECN marking similarly to DCTCP or DCQCN. Not doing this leads to sever performance problems due to the mismatch between the assumed cause of congestion and the real cause of congestion.

   (5 points for the scenario, 5 points for the solution)

Question 3 *(25 points)*

In class we discussed the Meltdown and Foreshadow attacks. A friend of yours says that the Meltdown attack demonstrates that Intel processors violate the atomicity of read operations promised in the Intel specs, because a process can see (and abuse) the side-effects of a read even if the read fails.

(a) What is (all-or-nothing) atomicity?
(b) Is your friend right or wrong? Explain.

Answer:

a) All-or-nothing atomicity is the property of a sequence of actions, whereby all of the component actions either complete as a unit, or the effects are not visible through the API provided by the system to its clients. Consider, for example, an ACID database transaction: it either completes and becomes visible to all future transactions, or no transactions ever see the effects. Visibility, in this context, refers to what can be seen through the system API – in the case of the transaction, multiple actions perform visible effects and populate the undo/redo log, but these effects are not observable through the SQL interface unless the transaction is committed.

b) Your friend is incorrect, by the following reasoning:
   i) Intel x86_64 processors do not actually guarantee the atomicity of memory reads, merely the fact that the MMU will disallow any accesses to memory which is not mapped or does not have the correct permissions.
   ii) When executing an illegal read as part of a Meltdown attack, the effects of the illegal read are not committed to the architectural state of the program, since no memory was updated. It only changes the micro-architectural state (the cache). The latter is similar to what goes into a database's undo/redo log and temporary changes vs. the former, which is similar to what is visible through the SQL interface.
   iii) Therefore, the MMU's guarantee that it will disallow rogue memory accesses is not violated, and the illegal reads were indeed checked (and failed) atomically.
   iv) The attack recovers the state from the cache using a side channel attack. Some think of the side channel as an interface, others don't even consider it an interface, but the bottomline is that it is not the system's API.

Grading scheme:

Part (a) - 10 points for mentioning the following:

- a sequence of actions which complete as a unit or whose effects are invisible to future actions (8pts for complete/abort as a unit, 2 points for specifying that effects are invisible).
- Only 5 points if the answer focuses specifically on memory reads/writes (because this should not affect the answer of part a)

Part (b) - 15 points, broken down as follows:

- 2 points for saying no, friend is incorrect
- 10 points for the insight that the illegal reads actually did not update any program state that is visible with the same software API (the ISA) as the illegal reads

- 3 points for saying that the recovery of the state is through a different API and therefore the MMU's protection guarantee is not violated

Students who assumed that "the atomicity of reads promised in the Intel spec" referred to the MMU's protection check and all subsequent instructions, were awarded 6 points:

- Assume that a cache line **should** be (all-or-nothing) atomic; either it is allowed by the hardware if the protection checks pass, or it is disallowed and should do nothing at all.
  (2 points)
- However, the Meltdown attack demonstrates that even failed reads leave some state behind in the processor's L1 cache.
  (2 points)
- Therefore, the forbidden reads were **not** (all-or-nothing) atomic.
  (2 points)

Question 4 *(35 points)*

Consider a new low-cost CPU that has multiple cores, each core with its own cache, but no coherence/consistency guarantees (i.e., you may write to one cache on one core, and that value may never propagate on its own to the caches on other cores). A cache can be flushed in its entirety by privileged-mode code (i.e., by the operating system kernel). Say you want to run on this CPU a multi-threaded application in which data is shared between threads.

(a) Is it possible for the application to run correctly without any assistance from the OS (i.e., the OS does nothing to help compensate for the lack of cache coherence)?
(b) Is it possible for the OS to transparently provide to the application the illusion of a sequentially consistent memory (i.e., the memory hierarchy appears to execute all reads and writes to a single memory location in a total order that respects the program order of each thread)? In other words, could an application written under the assumption of sequentially consistent memory run on this OS and CPU correctly without any modification?
(c) What is the right balance of functionality between the application and the OS kernel when it comes to operating efficiently on this CPU? Propose a concrete system call API offered by the OS.
(d) Using the API you proposed above, sketch an implementation of a doubly-linked list that supports concurrent search and insert/delete operations from multiple threads simultaneously. (Ideally express the implementation in pseudocode, but you may describe it just in English as well)

Answer:

(a) No, it's not possible to correctly execute an application with true data sharing on this non-coherent CPU. There is no upper bound on how long it takes for writes by one thread to propagate to other cores to be read, and furthermore can potentially be overwritten.
(2 points for NO, 3 points for simple explanation)

(b) We assume that the flush operation flushes and invalidates the caches in all cores.
Yes, it is possible in the following way: the OS marks all the application memory as read-only. Thus, any attempt to write the memory will trap. Then, the kernel performs the write operation and flush the cache.
(2 points for YES, 3 points for flush after write, 5 points for mechanism)

(c) The above approach is inefficient. Instead of offloading the implementation entirely to the OS, the application can assist. The application should split the shared variables from the variables that are accessed only by the current thread. The local variables have a single reader and a single writer so there is no consistency issue. The application needs to flush the cache every time it writes to a shared variable. To do so, the OS exposes the primitive to flush the cache through a system call, e.g. `msync()`. The compiler can also assist the programmer with this by declaring shared memory locations, e.g., similarly to shared pointers in C++, that can only be updated through specific functions, e.g., `update_shared(ptr, val)` that flushes the cache after the update.
(3 points for exposing the primitive through the syscall, 3 points for calling msync

after writes, 4 points for splitting local from global variables)

(d) Given the mutex_lock()/mutex_unlock() and the msync() function calls, we assume a global lock for the entire list.
For search(), lock the entire list, and search for the corresponding value, and unlock. A call to msync() is redundant because writes() always call msync() that also invalidates all the caches.
For insert and delete(), lock the entire list, insert or delete a node, msync, and unlock. Similarly an msync() after the lock is redundant.
Alternatively, one can use finer grained locking where individual nodes are locked.
(3 points for right msync placements, 7 points for locking)

Other answers that are acceptable:

- Splitting up/statically partitioning the shared data among threads, making it private. RPCs still need to be implemented over shared memory though.
- The OS can expose a transactional memory API to the applications, and implement the underlying system that tracks conflicting addresses in the kernel. Committing transactions need to flush their caches and updates back to main memory, and concurrent transactions accessing the same location (with at least one access being a write) require one of the transactions to abort.

Question 5 *(15 points)*

Name two of Lampson's hints that are reflected in the design of the Internet (viewed as a system) and explain how they have contributed to its success. If you can think of many that apply, pick the most interesting ones.

Answer:

Any principle could earn sufficient points if explained well. Some examples include:

- End-to-end
- Don’t generalize
- Keep basic interfaces stable
- Use a good idea again
- Cache answers

Grading scheme: 1 point for naming a principle, 13 points for explaining (6.5 each)