SP20\_OS Advande Operating System Midterm

**Q1. Timer Interrupts**

Imagine a system without timer interrupts. The time value is read from a hardware register. What would be the problem with CPU Virtualization on such a machine?

(4 points)

Without timer interrupts, the OS cannot perform CPU virtualization through context switch. If there’s no timer interrupts, the OS cannot take control of CPU until the running process yields its control, and thus a context switch cannot happen. If the running process is malicious or buggy, or simply taking very long time to run, the OS loses control of CPU and other processes cannot be run.

Also, since the time value is read from a hardware register, it may be changed by a process which is currently running on CPU. The ability to change that hardware register should be a privileged operation, which can only happen in kernel mode.

**Q2. Context Switches**

If we had a system where each thread was guaranteed to run on its own physical core, would this eliminate the need for the context switch mechanism? Explain your answer.

(4 points)

Yes, if we can guarantee each thread has its own physical core, there’s no need of context switch. The idea of having context switch is to virtualize CPU, giving the illusion that there’s infinite number of cores, so that more processes can run “simultaneously”. If each thread has its own core, then we simply have all processes running in parallel for sure, and there’s nothing to switch.

But obviously, it is common that the number of processes running is way larger than the number of cores a computer has. Having one physical core for each potential process is unrealistic because 1. The price can be very high and 2. More importantly, we need idle cores waiting for potential processes to run and it is a very inefficient usage of hardware.

**Q3. TLB**

Imagine a system where the TLB did not have protection bits; the space saved is used for more TLB entries. The page table still has protection bits. There is no address-space-ID in the TLB (that indicates which address space/process each translation belongs to). Is the memory isolation between processes still there? How does removing the protection bits affect security?

(8 points)

No, if there is no TLB protection bits and no address-space-ID, then memory is no longer isolated between processes. The protection bits check whether a process can have access to certain entries of TLB (and subsequently the page tables). Since page tables are per-process based, if there’s no protection bits on TLB, the virtual address can essentially be translated into wrong physical addresses (which belongs to other processes). This is thus a strong violation of security through isolation of memory access. Address-space-ID offers another way/protection to make sure TLB knows which process a given entry belongs to, and since we assume that there’s no address-space-IDs, then it’s very dangerous to not having protection bits.

One solution can be that every time there’s a context switch, TLB flushes everything out of its space, so that no process can access other processes’ page tables and thus physical memory. But this is very costly in terms of time/efficiency.

**Q4. Scheduler**

Imagine a multi-level feedback queue scheduler where all processes are considered to be batch processes when they enter the system. If a process does not use its full CPU quantum, its priority is increased. The highest priority jobs run at any given time, with round-robin policy being used among jobs with same priority. How does this scheduler perform for interactive and batch jobs?

(8 points)

This algorithm will keep interactive and batch jobs at highest priority, thus performing well for those jobs. The idea is that the interactive and batch jobs have more I/O requests or things like that and can yield the use of CPU back to OS more often. Since they usually yield CPU control before its quantum is finished, they tend to not occupy the CPU and then keeping them in the highest priority won’t hurt performance metrics such as turnaround time compared to the long running, non-interactive jobs. Also, since they are interactive/batch jobs, keeping them in high priority gives better user experience as users don’t wait very long without seeing any responses.

**Q5. LRU**

Given a cache of size three which uses the LRU policy, please show the state of the cache (which elements are in that cache) after it processes each one of these accesses: 1, 2, 3, 1, 4, 1, 3, 5, 3.

(4 points)

LRU stands for Least-Recently Used, meaning that if there’s no empty space in cache, the least recently used object will be flushed out, replaced by the new one.

Accessing 1: Cache: 1 (1 added)

Accessing 2: Cache: 1, 2 (2 added)

Accessing 3: Cache: 1, 2, 3 (3 added)

Accessing 1: Cache: 1, 2, 3 (1 is in cache)

Accessing 4: Cache: 1, 4, 3 (the LRU is 2, replace 2 with 4)

Accessing 1: Cache: 1, 4, 3 (1 is in cache)

Accessing 3: Cache: 1, 4, 3 (3 is in cache)

Accessing 5: Cache: 1, 5, 3 (the LRU is 4, replace 4 with 5)

Accessing 3: Cache: 1, 5, 3 (3 is in cache)

**Q6. Locks**

Implement a blocking lock and unlock operation using a compare-and-swap.

int lock(int\* lock\_ref);

int unlock (int\* lock\_ref);

Compare and swap: CAS(int \*location, int old\_value, int new\_value) returns old\_value;

Rough pseudo-code is fine, it does not have to be code that will compile.

(4 points)

int lock(int\* lock\_ref) {

return CAS(lock\_ref, 0, 1);

}

int unlock (int\* lock\_ref) {

return CAS(lock\_ref, 1, 0);

}

Int CAS(int \*location, int old\_value, int new\_value) {

if(\*location == old\_value) {

\*location = new\_value;

}

return old\_value;

}

**Q7. Atomic hardware primitives**

Why is it impossible to create a lock operation without an atomic hardware primitive such as compare-and-swap? Explain with an example.

(8 points)

It is because normal instructions like read() and write() are separate calls. The running thread/process may be context-switched out in between those two calls, i.e. checked the lock but not updated the lock yet, and thus another thread/process can still access the lock and may potentially update it before OS context-switching back to the first thread/process.

For example, there’s Thread1 and Thread2, each performing the lock operation using read() and write(), and the following happens in time order:

Thread1: read() and check the lock;

OS performs context switch, from Thread1 to Thread2;

Thread2: read() and check the lock;

Thread2: write() and successfully update the lock;

OS performs context switch, from Thread2 to Thread1;

Thread1: write() and also successfully update the lock; (\*\*here Thread1 should not “lock”, but because its check and update operations happened separately, it bypassed the whole idea of having a lock)

Therefore, we must have one atomic hardware primitive, so that it never pauses in the middle of check and update.

**Q8. Paging or Segmentation**

Imagine this particular scenario: A company wants to run a single batch-oriented process which uses all of the virtual address space. Performance is the most important factor to consider. In this scenario, should paging or segmentation be used? Explain your decision.

(4 points)

Paging should be used. Segmentation solves internal fragmentation, where there’s waste in between stack and heap in address space; paging concerns external fragmentation, where there are pieces of memory allocated here and there in physical memory and new requests for big chunks of memory will fail. Since this process knows that it will use all the virtual address space, there’s no need to worry about internal fragmentation and thus we don’t need segmentation. Instead, we should use paging to minimize external fragmentation, which in this case might be caused by segmentation.

Also, since performance is the most important factor, segmentation is much faster than paging, as it querying through a much smaller base and bounds pairs as opposed to large page tables. Therefore, by saving a lot of time on address translation, the performance under segmentation is much better.

**Q9. Paging or Segmentation (Part 2)**

Among segmentation and paging, which one is faster? Answer for two cases: one where there is no TLB, one where there is the TLB. Explain your answer.

(4 points)

If there’s no TLB, segmentation is faster. For a given process, segmentation only requires three pairs of base and bounds (and also some other bits), respectively for code, stack and heap. Thus, it’s fast to translate a virtual address (VA) to a physical address (PA). In contrast, paging breaks memory into smaller pieces and thus requires page tables (PTs) that are much larger than just three pairs of base and bounds. And the result is when performing VA to PA translation, more search work needs to be done in the page tables (either linear PTs or more complicated tree structures). Therefore, segmentation is faster. (Here we assume that we are only discussing efficiency in terms of speed, and thus ignoring other pros and cons of segmentation vs paging.)

With TLB, paging may be faster than segmentation, but I wouldn’t argue that it’s always the case. This is because segmentation base and bounds pairs are always very small compared to PTs, so even if paging is improved by TLB, it may still be slower than segmentation. Paging concerns more about managing physical memory orderly and reduce external fragmentation; if purely about speed compared to segmentation, it may not win even with the help of TLB. But still, with TLB, paging can be pretty fast, because TLB offers a cache for recently used memory accesses and avoid querying through page tables to find physical address. The performance of TLB is much faster than pure PTs by magnitudes, but it can hold much fewer information compared to PTs both due to cost and physics rules.

**Q10. 32 bit to 64 bit**

Imagine we are porting a system (that uses paging) from using 32-bit virtual addresses to 64-bit virtual addresses. What needs to change in memory virtualization (ignore the other changes that will be required)? Can we do this entirely in software? Why or why not?

(8 points)

No, we cannot do this merely in software. Since we use paging, the page table structure will be very different, which requires not only software changes, but also changes in the hardware help we need in page tables. For the sake of efficiency and protection, a lot of paging is done through hardware. When we translate the virtual address into physical address through paging, the hardware needs to get the first few digits (virtual page number, VPN) and go to the page tables to get the corresponding physical address and then apply the offset to get the exact physical address. 32-bit and 64-bit may have very different scheme regarding the size of VPNs/PPNs and offsets. For example, in the lecture of Multi-Level page tables, Professor gave the example that for 32-bit, there might be a 10-bit Page Directory, a 10-bit Page Table, and a 12-bit offset, whereas in 64-bit, there might be a 12-bit unused, four levels of 9-bit Page Tables (Global, Upper, Mid, Lower) and a 12-bit offset. These are not things we can easily handle in software, but rather to change the behaviors of hardware.

**Q11. Free**

When you pass a pointer obtained from malloc() to free(), how does it know how much memory to free?

(4 points)

First, malloc() and free() are not system calls themselves, but rather just library calls. Within them, there will be system calls actually performing memory related operations. Second, those system calls figure out how much to free with the help of hardware. Each piece of memory, when allocated, comes with a header, which includes the size of this piece of memory and likely some other information, such as a magic number. When we call malloc(), a header is automatically created and a piece of memory of desired size (plus the size of header) is allocated; when we call free(), the pointer (the argument of free()) is updated to the actual beginning of that piece of memory (i.e. pointing to the beginning of header) and the size information is there, so that the full piece of memory can be released.