CS 1217 Operating Systems, Spring 2023 End Term Exam

Date: **Wednesday, 3rd May 2023**

Maximum Marks - **130**

Total time **- 90 Minutes (1.5 Hours)**

Instructions : Read carefully before proceeding

1. Be as precise as possible in your answers. Make sure to articulate the answers to the best of your abilities. Please note that the answers are about content and not about volume.
2. This is a closed book, closed notes, closed Internet and closed devices (including smartwatches) exam
3. If you make any additional assumptions other than the ones stated in the question, please state them clearly in your answer.
4. **Plagiarism will result in a fail grade for the entire course**. All the work that you submit should be your own.
5. Ideal exam taking strategy would be to go over the entire question paper before you begin. Identify the questions that you can formulate a good answer for and attempt those first.

Q #	Max Points	Earned Points
1	5	
2	5	
3	15	
4	15	
5	10	
6	5	
7	5	
8	10	
9	10	
10	15	
11	10	
12	25	
Total		

Q1. On a CPU that has a 32-bit architecture, assume that the TLB can hold 32 entries. Each entry has a Virtual Page Number and the associated Physical Page number. The Physical Page size is 8 KB. What is the amount of memory that can be addressed by the TLB, assuming only 75% of the entries in the TLB are valid? (**5 points**)

192 KB. You'll have to start with calculating the number of valid entries in the TLB, and then figuring out the capacity that they can address.

Q2. Describe the difference between software- and hardware-managed TLBs and how they affect kernel's TLB and page fault handling. Briefly describe the pros and cons of each approach. (**5 points**)

A software-managed TLB will generate a TLB exception if a page translation needed to resolve a virtual address is not loaded into the TLB. In contrast, a hardware-managed TLB will search the currently-loaded page tables itself to try and locate a valid translation and load it into the TLB if necessary. On a hardware-managed TLB, the kernel never sees TLB faults since they are resolved by hardware directly; only page faults are raised and resolved by the kernel. In contrast, a software-managed TLB sees both TLB and page faults.

The main advantage of a hardware-managed TLB is that hardware is faster at loading TLB entries and can avoid the overhead of trapping into the kernel. The disadvantage is that the kernel must implement its page tables in a way that the hardware can understand. The kernel may also lose some visibility into page usage if the hardware managed TLB handler doesn't set some state on the page table entry to indicate that it has been faulted on and used. Advantages of the software-managed approach include more flexibility for the operating system in how to structure its page tables.

Q3 For this question, let's discuss the concept of Copy on Write (CoW) applied to memory management of processes. Many times, a call to fork() is immediately followed by a call to exec(). In these cases, the replication of the parent's address space in the child is typically an unnecessary overhead. However, in many other cases, the child process carries on the work of the parent. In this case, replication of the address space of the parent in the child is clearly not something that can be done away with. To the contrary, it is actually useful.

CoW is an overhead reduction strategy that relies on the observation that fork() makes an identical copy of the parent's address space. So afterward, we know that for two virtual addresses in the parent and child, VA_{Parent} and VA_{child}, if $VA_{Parent} = VA_{Child}$ at the time fork() is called then the pages have identical contents. As long as the parent and the child are just reading from the page, they can share the same page. In other words, both the parent and the child process pages can point to the same physical page, which holds the actual contents. Problems arise only when one of them tries to write to a page. It is at this juncture that a copy of the page is created and the parent and the child get their own copy of the page.

First, describe the steps that would have to be taken for the OS to implement such a scheme.

While copy-on-write is a clever mechanism to use after fork(), there is a possibility that it will not identify *all* possible page sharing opportunities. Explain why not.

Second, describe a system for identifying page-sharing opportunities missed by copy-on-write. You need to be specific about how your system works: how it identifies identical pages, how they are merged, and how it ensures that pages that should be private, stay private. (**15 points**)

Copy-on-write after fork() will only identify shared pages that emerge from the parent-child relationship. So, for example, if /bin/whatever forks another copy of /bin/whatever, then many pages may be shared via COW. However, if Alice and Bob independently start separate copies of /bin/whatever, then the same pages will not be shared.

A system for identifying shared pages that do not emerge from fork() has to have same way of determining that two pages are, in fact, the same. You can imagine multiple ways of doing this: computing a per-page hash, for example, which uses the page contents directly; or annotating each page with the file that it came from, which would allow identical pages from /bin/whatever to be identified. File annotations would not have this problem, but could not identify pages that happen to be identical even though they came from different files. Question to ponder about : How likely is that to happen?

Using its identity function, the samepage daemon would periodically scan through pages loaded in memory. (I guess you could do ones in the swap partition too, but it's not clear how helpful that is since the goal is to reduce memory usage.) When it finds two that are identical, it do several things. First, both pages must be marked as read-only, to prevent writes during the process. Second, one of the pages has to have its PTE updated to map the virtual address to the physical address of the second page. The page should also be marked as shared between the two processes so that eviction and swap out work correctly. (Modern kernels usually have existing mechanisms to support page sharing between processes, since this can be established as a form of IPC.) Finally, the extra page can be freed. To ensure that pages stay private, we recycle the copy-on-write idea: mark them as read-only and split at the first write.

The same approach has also been used on memory-constrained devices such as smartphones, and in particular on Android which runs many virtual machines and therefore may have many duplicated pages.

Q4. In class, we played the game of "Policy vs. Mechanism". In this question, we will play "Stupid or Clever", which is built along similar lines. In the next set of questions, you will be provided with a few design choices for various types of operating system subsystems. You should classify each design decision as either Stupid or Clever, and provide arguments in support of your answer. (**15 points**)

(a) Commit the data of all files to the commit log before committing the metadata information.

Stupid: The overhead of logging the data twice will be too high. The log will now be very big, which is not advisable.

(b) Implementing a memory management subsystem of the OS that is completely agnostic to the details of the underlying architecture.

Good idea for OS design from a software engineering perspective, but bad since the performance will not be optimized. Will take arguments in favor of either, if the argument is corrrect.

(c) The Mach virtual memory system, developed in the 80's, was divided into a machine dependent (pmap) layer and a machine independent layer. The machine independent layer implements a logical virtual memory system with a logical page size (that must be a multiple of the physical page size) and the machine dependent layer translates the logical structure into the physical structures of a particular machine, updating hardware mappings. (**15 points**)

Ans: Clever. Many types of processors are around. This can help to not have a processor architecture specific implementation of the virtual memory system.

Q5 Is there a possible interleaving of the function calls as enumerated in the following code segments lead to deadlock? Answer "yes" or "no" with brief explanations about when (if) deadlocks can happen (**10 points**)

Part 1

Process 0:	**Process 1:**
lock1.acquire();	lock1.acquire();
lock2.acquire();	lock2.acquire();
lock1.release();	lock1.release();
lock2.release();	lock2.release();

Ans : No deadlock possible, because there is no circularity of requests. Both processes grab the locks in the same order.

Part 2

Process 0:	**Process 1:**
lock1.acquire();	lock2.acquire();
lock2.acquire();	lock1.acquire();

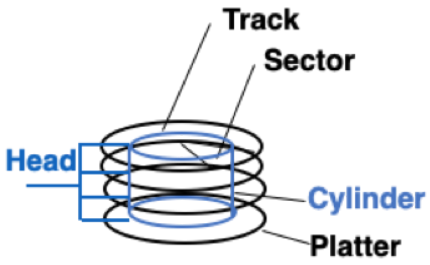
```
lock1.release();        lock1.release();
lock2.release();        lock2.release();
```

Ans: Deadlock is possible, because there is circularity. If process 0 acquires lock1 and then process 1 acquires lock2, both will be stuck holding one lock while waiting for the other.

Q6 Provide one example each of an exception that would terminate a running process and an exception that would not. Describe the sequence of steps that happen when an exception occurs. (**5 points**)

Multiple examples covered in class, any one describing the correct set of steps will be accepted.

Q7 Imagine that you are incharge of optimizing the access latency of large, multi-GB multimedia files that will be stored on spinning HDDs with multiple platters. The disk interface allows the data to be written to it in 128 KB chunks, which is also the size of a block for this HDD. The file system in conjunction with the disk controller writes the individual blocks of a file to available free blocks, wherever it can find them across the disk. First explain how this mechanism would lead to high latencies for file lookups. Next, imagine that you have been tasked with optimizing this design. How can you modify the existing file system to use the concept of cylinders to reduce the average access latency to these multimedia files? Below is the conceptual representation of the disk to help you visualize the HDD. (**5 points**)



A simple observation is that media files, especially videos tend to be read sequentially. Typically, you'll start watching a movie from the beginning and watch it to the end. To make sure that the "next" piece of the video is accessible with low latency, we have to make sure that the movement of the head is reduced as much as possible. There are two possible optimizations

1. On a single platter, try to write large chunks of video data in blocks that are sequentially laid out on the same track.
2. Next, when free blocks are found on a platter, try to find more blocks that lie on the same cylinder as shown in the figure. In the ideal case, when writing file data, we should write the data blocks across all the 4 platters of the cylinder as shown in figure.

Q8 We have discussed how shared data structures in multi-threaded applications can lead to reduced performance for multi-threaded programs. Provide an explanation regarding why that happens.

Provided next is a snippet of code that does locking to implement correctness. Describe how you can perform more intelligent locking to improve the code's performance without sacrificing correctness or significantly increasing the amount of space needed to store the data. As a hint, you can assume that the loop frequently has to examine many entries before it finds one that is available. The code to remove entries is not shown, but you can assume that entries are periodically removed. (**10 points**)

```
1  struct item {
2      bool valid;
3      int value;
4  };
5
6  struct item array[32768];
7  struct lock * lockArray;
8
9  int saver(int bestValueEver)
10 {
11     bool failed = 1;
12     // Assume that arrayLock was properly initialized.
13     lock_acquire(lockArray);
14
15     for (int i = 0; i < 32768; i++) {
16         if (array[i].valid == 0) {
17             array[i].value = bestValueEver;
18             array[i].valid = 1;
19             failed = 0;
20             break;
21         }
22     }
23
24     lock_release(lockArray);
25     return failed;
26 }
```

On multicore systems extra locking can reduce concurrency in certain cases. Imagine two threads on separate cores are trying to save a **bestValueEver** using the provided code. Once one of them grabs the big lock and begins to walk the entire array, the other has to wait until it is complete. This is despite the fact that a lot of the work can be done without holding the lock. How? By using the read-lock-read-alter synchronization pattern. Here instead of using the lock to ensure correctness during the whole loop, we walk the array without holding the lock and use the current state of array[i].valid as a hint indicating that this entry might be free. At that point we have a candidate entry to examine further, and then we grab the lock. Once we have acquired the lock, we need to recheck the entry to make sure that it's still available, but if so we can save our **bestValueEver** and exit. There will be a few cases where concurrent access will result in two threads locking and checking the same entry, at which point only one should win and the

other must continue. But this is unlikely, and could be made even less likely by having each thread start their search at a random point in the array rather than at the beginning.

Instead of holding the lock for the entire loop, acquire it only when we think we have found an available entry but then need to double-check that this is actually true. (You could actually make this a bit slicker by only setting the valid flag inside the lock and then setting the value once you have marked it as in use.)

Q9.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int32_t foo[1024];
5
6 int
7 main(int argc, const char * argv) {
8   buffer = (char *) malloc (10240); // You can assume that malloc succeeds.
9   int32_t bar[1024];
10  // <-- Here -->
11  struct fooer[24];
12 }
```

(**5 points**) Examine the simple program above. At the point in its execution indicated, at minimum how many 4K pages of memory will it require in each segment: code, data, stack and heap? Justify your answer (**5 points**). Note that you can ignore dynamically-loaded libraries, and assume that malloc does not consume any memory for its own data structures.

Ans : The process has:

- At least one code page: Most of the code mentioned above would easily fit in a page
- At least one data page: for the 4K foo array
- At least two stack pages: one for the 4K bar array, and at least one more for anything else on the stack. (There is at least one other thing on the stack: argv!)
- At least three heap pages for the 10K dynamically-allocated buffer.

Q10. Consider a simple system running a single process. The size of physical frames and logical pages is 16 bytes. The main memory (RAM) can hold 3 physical frames. The virtual addresses of the process are 6 bits in size. The program generates the following 20 virtual address references as it runs on the CPU: 0, 1, 20, 2, 20, 21, 32, 31, 0, 60, 0, 0, 16, 1, 17, 18,

32, 31, 0, 61. (Note: the 6-bit addresses are shown in decimal here.) Assume that the physical frames in RAM are initially empty and do not map to any logical page.

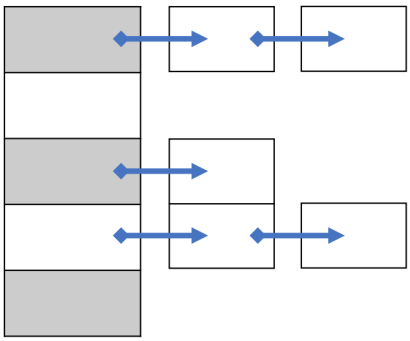
(a) Translate the virtual addresses above to logical page numbers referenced by the process. That is, write down the reference string of 20 page numbers corresponding to the virtual address accesses above. Assume pages are numbered starting from 0, 1, ...

(b) Calculate the number of page faults generated by the accesses above, assuming a FIFO page replacement algorithm. You must also correctly point out which page accesses in the reference string shown by you in part (a) are responsible for the page faults.

(c) Repeat part (b) above for the LRU page replacement algorithm (**15 points**)

(a) For 6 bit virtual addresses, and 4 bit page offsets (page size 16 bytes), the most significant 2 bits of a virtual address will represent the page number. So the reference string is 0, 0, 1, 0, 1, 1, 2, 1, 0, 3 (repeated again).
(b) Page faults with FIFO = 8. Page faults on 0,1,2,3 (replaced 0), 0 (replaced 1), 1 (replaced 2), 2 (replaced 3), 3.
(c) Page faults with LRU = 6. Page faults on 0, 1, 2, 3 (replaced 2), 2 (replaced 3), 3

Q11
Assume the data structure shown in the figure below, which is being accessed by a multi-threaded program. Assume that each one of the boxes on the left contains a pointer to another linked list. Each one of these lists can be accessed using a **get()** and **put()** interface. **Get()** allows a thread to remove an entry from a given list, while **put()** allows a thread to add an entry to a list. Given these assumptions, answer the following questions. (**10 points**)



.

a) How will a race condition be caused in the situation described above? Give a simple example.

Suppose put(5) and put(10) run in parallel. Both threads read and write to table[0]. The ordering of their execution might lead to a race condition.

a) Describe at least two ways in which you can use locks to avoid the race condition identified in part(a). Which one of the two will have better performance? Why?

Can have coarse grained locks (one lock for the entire structure) or fine grained locks (one lock per table). Typically, coarse grained locks have worse performance since they limit parallelism.

Q12 During this course, we have talked about how designing computer systems is an exercise in trade-off analysis. Anything that you want your system to have, has an associated price with it. Typically, computer systems are forced to make trade-offs in price, performance, energy consumption. For memory and storage systems, one can add (memory or storage) capacity trto the mis as well. Memory, which is typically made of DRAM, is fast (nano-second access latencies) but expensive (per GB). In addition, DRAM is volatile -- once your pull off the power cord, all the information in there is lost. HDDs on the other hand are cheap (per GB), non-volatile, but *terribly* slow (access latencies in milli-seconds). Flash is non-volatile as well, but much slower than DRAM and more expensive (per GB) than HDDs.

Now, imagine that you can acquire, very cheaply, a device with a terabyte (or more) of fast and byte-addressable (like memory) but non-volatile (like disk) storage. This device is made up of Non Volatile Memory (NVM) chips that overcomes some of the limitations of Flash. Now there are two ways in which these NVM chips can be used. The first is to use them as main memory, where they can replace DRAM as main memory. The other is to use them as storage devices, where we leave the DRAM untouched, but use these NVM devices as a part of the storage system. In yet another case, both the memory and the storage devices can be replaced with these NVM chips.

1. Present and motivate five different significant aspects of OS design that you would consider redesigning if you were designing an OS for a device with a single large and fast byte addressable NVM chips replacing both memory and the disk (5 points each)

Five may seem like a lot of things to think about, but there are so many ways that this could revolutionize OS design. Think through the various subsystems that currently manage or use memory and the storage system. You should also think about the various OS operations that move state back and forth between memory and the disk, and how those operations would get affected by the change in assumptions about system architecture. Think about how NVM and disk are managed differently and how you could unify management of a single NVM chip. Think about process startup and shutdown, installation and update, state maintenance, and the effect of software bugs. Think about reboot. Consider big parts of the OS that may no longer need to exist, but also about side effects of the volatile nature of memory that you may want to preserve on NVM systems. (**25 points**)

If you want the real complete solution to this question, look up this paper from the 13th Workshop on Hot Topics in Operating Systems (HotOS'11). The solution is essentially just paraphrasing their ideas.

Here's a set of eight, but there could be more.

1. Goodbye paging and 4K page sizes. Quite obviously, a device without both disk and memory no longer needs to move pages from disk to memory. Remember our goal when choosing pages to evict? We wanted to make the system look like it had as much memory as the disk size, all of it as fast as memory. Mission accomplished! This is an entire large OS subsystem that we can largely remove. In addition, remember that we chose a page size for a variety of reasons that had a lot to do with the backing store. When swapping, large pages minimize the size of kernel data structures while also amortizing the cost of disk seeks inside the swap file (or disk). But we are no longer swapping. Obviously, however, page size still plays a role in terms of hardware-aided address translation via the MMU and TLB, but because the NVRAM is byte-addressable we can experiment with new pages sizes more suitable given this radical architectural change. Some memory allocation and protection mechanisms even eliminate pages entirely.

2. Unifying memory and file system protection. Because we no longer have a separation between the memory and the disk, we need to consider how to unify the protection models. Memory protection is hardware-enforced on page boundaries. In contrast, file systems provide more coarse-grained (file-level) but richer protection semantics, including the idea of ownership, group permissions, permission inheritance (via directories), and even the flexibility provided by capability-based access control lists (ACLs) on some file systems. Part of this richness, obviously, is because file system protection mechanism are implemented in hardware, rather than software. But now with a single large NVRAM chip serving as both memory and disk, we might want to consider a way to unify these. At minimum, we have to ensure that the protection semantics provided by hardware when the NVRAM is used "like memory" are more closely-matched by what the OS provides in software when the NVRAM is used "like a disk".

3. Unifying and memory and file system naming. Another fundamental difference between memory and file systems we need to reconsider is the idea of address spaces and how virtual addresses are translated. Address spaces essentially provide each process with a separate local name space—recall that a virtual address is meaningless without an address space to translate it inside, or alternatively without knowing what process is translating the virtual address. In contrast, file systems provide a global name space: /path/to/foo is the same file regardless of which process is opening or accessing the file. Unifying these two abstractions requires identifying and addressing this issue. One approach is to move to single address space machines, which have been previously explored

4. Installing, packaging, and launching apps. Today, executing an app essentially transforms it from one form—on disk—to another—in memory. This is the job of the ELF file loader during

exec(). A single NVRAM chip replacing both the disk and memory makes this unnecessary, and apps can be distributed in their ready-to-run form, eliminating the job of the ELF loader. Another big part of the OS gone! Although you still need to consider how to reconcile the addresses the process wants to use with the NVRAM that's available on the machine, but loadable libraries have approaches to doing this that could simplify the process. This change also fundamentally alters how applications save state. Currently processes have to write out to stable storage to save state and have developed elaborate and process-specific mechanisms for doing so. A big chunk of non-volatile memory makes that unnecessary, and allows them to essentially be stopped and restarted in any state without any additional process support. So this really completely eliminates the usual process of "starting up" and "shutting down" a process as we currently know it, with no impact on memory consumption. Cool! Finally, NVRAM also makes it extremely easy to move process setups from machine to machine, stored exactly at any point in their execution.

5. Application faults. At last, a downside! One of the interesting side effects of NVRAM is that what used to be impermanent (RAM) is now permanent. Starting up and shutting down processes isn't done only to let the OS know that you are done using a particular program and allow its resources to be freed—it also creates a chance for the process to reload its memory contents from a known good starting point frozen in the ELF file. This is particularly important if the process is restarting to recover from a fault. So removing this capability entirely isn't necessarily a great idea, since whatever state corruption led to the fault is now impossible to remove. There are a few ways to address this without forcing processes back to storing state "on disk". One way is to have users create snapshots or checkpoints of the process at good states, which could be stored in the (very large) NVRAM and reverted to after failures.

6. Decoupling power cycles and reboot. Another interesting effect of the single large NVRAM chip is that power cycles no longer have to trigger a reboot. If the device loses power, (almost) all of its state is permanently stored on the NVRAM, with the exception being any register state associated with its execution at that precise instant. (Although with a small amount of battery backup that could be written to the NVRAM after a sudden loss of power before the processor went offline.) So the idea that power cycles need to trigger a reboot on wall-powered devices no longer holds. On battery-powered devices the story is even simpler, since they already have ways to monitor their battery levels and don't need to be unprepared for most power outages—with the exception being the user suddenly ripping the battery out!

7. Reboot and data corruption. While this sounds like a plus, reboots for the OS can serve the same useful purpose as restarts do for processes: the chance to reload potentially-corrupted state from a known good starting point. So it probably makes sense to preserve some kind of reboot mechanism that can be triggered by users when needed that allows the OS to reinitialize state and recover from faults it may be unable to detect. But again, there is no need for reboots to be coupled in a way to power cycles, since the "memory" never loses state.

8. New sleep states. As previously mentioned, the overhead of entering a low-power sleep state is now limited to the cost of unloading the process registers into the NVRAM and then powering down the processor. On most mobile devices, the memory must either be moved to stable storage in order to be powered off—an expensive process—or kept in an active state to avoid losing contents, which consumes power. Our new device avoids this entirely and can sleep both much more quickly and completely and power on much more rapidly.