Solutions to Assignment 3

Written Exercises

W1. Explain how sharing memory over a network (as you have done with Mneme) is similar to sharing files over a network. Briefly discuss the similarities and differences.

In modern operating systems, files are typically represented as a long sequence of bytes that can be accessed randomly. Memory is also a long sequence of bytes that can be accessed randomly. In fact, the current trend in operating systems is to blur the line between memory and files—many operating systems cache the contents of frequently used files in memory and swap infrequently used memory areas out to disk. In fact, some operating systems (such as Mach) even use the same paging algorithm to regulate both cached files and memory contents.

The key difference lies in how data is accessed. Access to memory is very unstructured, and random access is the norm. Access to files is more structured, through calls to read, write and lseek (or equivalent functions). File access is usually sequential—random access is permitted via the lseek function, which can move the current position around, but it remains a more cumbersome mechanism than the direct random access available for memory.

Some modes of access to files have no analogue in memory access. For example, it is possible to open a file in "append mode" where all data will be written to the end of the file, growing the file as needed.

In addition, the size of a shared memory area is limited by the logical address space of the processor, whereas the size of a file is limited by the filesystem API and the filesystem implementation.

Finally, files are usually considered to be persistent, whereas memory contents are ephemeral. When a process exits, its memory contents are lost, so we might think it reasonable that when the last process finishes using a shared memory area, it should be destroyed. On the other hand, local files are kept around until deleted by the user so we expect the same from a networked filesystem.

W2. Your original implementation of Mneme allocates pages of memory to one process at a time, making the acquisition of a page by a process akin to the acquisition of a mutex lock. However, it is possible to adopt a strategy for page acquisition akin to the readers/writers problem. In fact, it is possible to go beyond traditional readers/writers solutions (which disallow reads while one process is writing) and develop a solution that allows readers to continue to read "stale" data until the current writer has written new data.

(a) Explain how such a scheme would work by describing the flow of requests and data between clients and the shared-page server.

Obtaining a page for read/write access works as it did in earlier versions of Mneme—a writable page can only be "checked out" to one client at a time. Thus, when a process needs to write to a page, the Mneme library sends a request to the shared-memory server to check out the page. If the page is not checked out to another process, it is marked as checked out and the page contents are sent to the requester

(if the page is already checked out to another client, the requester must wait its turn). Sometime later, the Mneme library should expire the page and check it back into the server so that other processes can read the updated data and any waiting writers can have access.

If a client requires read-only access to a page, we allow the Mneme library to fetch the page contents from the shared-memory server, even if the page is checked out for writing. If the page is checked out, the server will send stale data but the stale data cannot cause any inconsistencies, provided that eventually the stale data will be replaced with up-to-date data.

Read-only pages must also be expired because otherwise changes to memory contents would never propagate to readers. There are two obvious ways to perform these expirations: the first method is to set a timeout as we do for read/write pages—when the timeout expires, we mark the page as invalid to ensure that it will be fetched again from the server. The second method is for the server to tell clients when the page has been updated on the server and so is no longer valid.

Readers do not change the pages they fetch from the server, so there is no need for them to send any messages to the server when the page expires.

Unfortunately, while this readers/writers extension to Mneme works for normal access, it can cause problems for memory that has been locked with shmem_lock. Users of shmem_lock expect updates to locked memory to be atomic—if a program shmem_locks a 10 KB area, modifies it, and then shmem_unlocks it, they should expect that no other processes will be able to see the area in a partially updated state. But if the area falls across a page boundary, and shmem_unlock checks in the two pages in two transactions, there will be a moment when the server will hold partially-updated data. Since we have extended the server to allow reads at any time, the server could send that partially-updated data to readers, violating the semantics of shmem_lock. One solution to this problem is to have shmem_lock lock out readers; another solution is to have shmem_unlock update the pages on the server in a single atomic transaction.

(b) Explain why this kind of solution to the readers/writers problem is not usually possible.

Normally each reader and writer does not have its own private copy of the data being accessed; instead they manipulate a shared structure.

In addition, we are not attempting to provide any of the usual guarantees of the readers/writers problem. With real shared memory, there can be multiple readers and multiple writers all accessing the memory simultaneously. The only reason we do not allow multiple writers write to the same page simultaneously in Mneme is that because updates are made to a local copy of the page and then passed back to the server, it would be too difficult to provide proper consistency, not because multiple simultaneous writes should not be allowed.

w3. What is meant by the term *busy waiting?* What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answers.

When a task cannot continue until an event occurs, and so repeatedly polls to see if the event has occurred, it is said to be *busy waiting*. The key idea is that the task continues to consume processor time by repeatedly checking the condition. Examples of busy waiting include software synchronization algorithms, such as the bakery algorithm, and *spin locks* using hardware synchronization primitives such as test_and_set.

The alternative to busy waiting is *blocked waiting* (also known as *sleeping waiting*), where a task sleeps until an event occurs. For blocked waiting to work, there must be some external agent that can wake up the task when the event has (or may have) occurred.

Processes also wait when they are ready to run (not waiting for any event) but the processor is busy executing other tasks.

Busy waiting cannot be avoided altogether. Some events cannot trigger a wakeup; for example, on Unix a process cannot "sleep until a file is modified," because the operating system does not provide any mechanism to automatically wake up the process when the event occurs; some amount of repeated polling is required. (Windows NT, on the other hand, does allow this kind of wakeup.)

Also, blocking synchronization primitives such as semaphores need to use a lower-level synchronization technique in their implementation. On a uniprocessor machine, techniques such as disabling interrupts can enforce a critical section, but on multiprocessor machines, disabling interrupts is inadequate and some form of busy-waiting synchronization must be used.

Generally, busy waiting is mostly avoidable on uniprocessor machines, but is mostly unavoidable on multiprocessor machines. However, even on multiprocessor machines busy-waiting is best used for very short waits and limited to operating system code.

w4. Explain the difference between logical and physical addresses.

A logical address is a memory address used inside programs running on the machine—the logical memory of a program is the memory it may access. A physical address an the actual location in the physical memory of the machine—there is a direct correspondence between physical addresses and locations on the memory chips of the machine. Logical memory may not be present in physical memory at all. A logical address might refer, say, to a segment that has been swapped out. The correspondence between logical addresses and physical addresses need not be fixed—the operating system might move a partition, a segment, or a page during the lifetime of the process and have that move be invisible to the process.

w5. Explain the difference between internal and external fragmentation.

Internal fragmentation occurs when allocations are quantized, such as having to be a multiple of 8192 bytes, or a power of two. A program wishing to allocate 10,000 bytes on a system where allocations must be a power of two will waste 6384 bytes due to internal fragmentation. Internal fragmentation can be reduced by reducing the amount of quantization.

External fragmentation occurs when free space is not concentrated in one place, but scattered as pockets (or *holes*). If we have a 10 KB hole in one area, and a 7 KB hole somewhere else, we will not be able to satisfy a request for 16 KB of contiguous space even though we have 17 KB of free space available. External fragmentation can often be remedied by using *compaction* to collect all the free space together.

Thus, as their names suggest, internal fragmentation is wasted space "on the inside" of an allocated area, and external fragmentation is is space wasted between allocated areas.

w6. Why are page sizes always powers of two?

The hardware needs to turn a logical address into a page number and offset. The calculation is pageno = addr **div** PAGE_SIZE and offset = addr **mod** PAGE_SIZE. Both **div** and **mod** are complex operations unless PAGE_SIZE is a power of two. But if PAGE_SIZE is a power

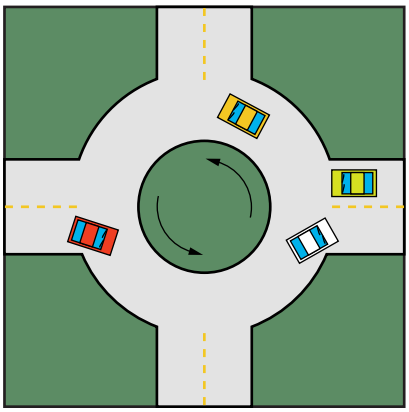

Figure 1: A Traffic Circle.

of two, the **div** operation can be replaced with a right shift and the **mod** operation can be replaced with an **and** operation, both of which can be performed rapidly.

w7. What complications arise if an operating system that manages memory using segmentation allows processes to increase the size of their segments?

If a segment is immediately followed by a large enough hole (free space), we could allow processes to increase the size of their segments. However, the situation becomes complicated if there is not a large enough hole following the segment for the requested increase, but there is sufficient free space elsewhere in memory: Should the request be denied, or should memory be reorganized?

Neither solution is very satisfactory. Denying requests could lead to a situation where a program which needs 1 KB more memory on a machine with 16 MB free and is denied its request. Reorganising memory is difficult because it is not clear what the best reorganization would be. Should all free space be concentrated into one area, or should it be evenly distributed so that each segment has room to grow? Whichever choice we make, we may prove to be wrong and leave the operating system shuffling and reshuffling memory over and over.

w8. On the road, four-way stops are (theoretically) prone to deadlock. Traffic circles (aka roundabouts) are an alternative to four-way stops. In a traffic circle, all vehicles reach their destinations by driving around the traffic circle in a counterclockwise direction. Vehicles wanting to enter the circle must yield to vehicles already driving around the circle (i.e., vehicles on their left). An example of a traffic circle is shown in Figure 1.

(a) Are four-way stops prone to starvation?

No. At a four-way stop, vehicles must yield to the vehicle that has been at the stop the longest, thus we cannot have a situation where one vehicle has to wait forever while other vehicles keep being allowed to go.

(b) Are traffic circles prone to deadlock?

No. There can be no circular wait, since each car is waiting for space on the traffic circle, rather than waiting for another (possibly waiting) car.

(c) Are traffic circles prone to starvation?

Yes. Suppose there is a continuous stream of traffic from east to west (and no cars travelling south to north). A car wishing to drive north to south will not find a opening in the traffic and will starve.

(d) Give an additional traffic rule that could be used to prevent deadlock at four-way stops.

If four vehicles arrive at the four-way stop at the same time, the most northerly car gets to go first. (The stop sign could be labeled with an "N" so that drivers knew which lane had priority).

W9. Some paging hardware does not support the "referenced bit" needed by various page-replacement algorithms. Explain how the protection bits in the page table can be used to emulate a referenced bit.

We can use a technique of having two page tables: the hardware page table and a master page table. The master page table is maintained by the operating system and includes a referenced bit. For pages whose referenced bit is set, the entries in the hardware page table mirror the entries in the master page table, but pages which have their referenced bit clear in the master page table are marked as invalid in the hardware page table. When a protection fault occurs, the operating system checks its master page table. If the access should have been allowed according to the master page table, the protection information from the master page table is copied into the hardware page table, and the referenced bit is set in the master page table.

W10. When calculating the hash function for an inverted page table, we include the process ID as well as the page number. Why?

An inverted page table is a global table for all processes, and usually logical address spaces for processes overlap—for example, logical address 100 may exist in many processes. Thus each entry in the table must include the process ID. The hash function must also use the process ID to prevent numerous collisions from occurring.

W11. This question is about the similarities and differences between FIFO page replacement with page buffering (FPB) and the clock page-replacement algorithm. Both methods usually provide a fair approximation of LRU page replacement.

In the questions below, assume a uniprocessor machine where the page fault routine will not be interrupted to run another process. State any additional assumptions you make.

(a) Explain how these algorithms have strong similarities.

Both algorithms perform the following operations on each page:

- Mark the page as being potentially inactive.
- Allow the page to wait for a while to see if its status changes. (If it is accessed, the mark is removed.)
- Later, if the page is still inactive, the page is reused for something else.

In the clock algorithm, potentially inactive pages are marked by having their referenced bit cleared. In FPB, potentially inactive pages are marked by being placed into the free queue.

In the clock algorithm, the next page to be replaced is the first unmarked page following the hand of the clock. In FPB, the next page to be replaced will be the page at the head of the free queue.

While the clock algorithm allows the possibility that all or none of the pages could be marked as inactive, the FPB algorithm strives to keep some fixed percentage of frames marked inactive.

Finally, the FPB algorithm uses a FIFO strategy when choosing pages to mark page as inactive—the page that gets marked is always the one that has been marked active the longest. In contrast, the clock algorithm simply cycles through the frames in order.

(b) Consider the situation where every frame is in active use, all frames are clean, and there is a heavy demand for frames (i.e., the system is thrashing).

i. Does the clock algorithm degenerate to another (non-LRU) policy (such as FIFO or RAND) under these circumstances? If so, state what policy it becomes and why.

It degenerates to RAND or FIFO depending on how badly the machine is thrashing. If the thrashing is severe, every page access refers to a page that is not in memory and the clock algorithm will degenerate to FIFO. If the thrashing is not so severe and some page accesses touch pages that are in memory, it will degenerate to RAND. The problem in the latter case is that the hand of the clock is going around much too fast to provide any reasonable approximation of LRU replacement.

ii. Does the FPB algorithm degenerate to another (non-LRU) policy (such as FIFO or RAND) under these circumstances? If so, state what policy it becomes and why.

It degenerates to RAND or FIFO depending on how badly the machine is thrashing. If the thrashing is severe, every page access refers to a page that is not in memory and the FPB algorithm will degenerate to FIFO. If the thrashing is not so severe and some page accesses touch pages that are in the free queue, it will degenerate to RAND. The problem in the latter case is that pages are not staying in the free queue long enough for the FPB algorithm to provide reasonable approximation of LRU replacement.

(c) Consider the situation where every frame is is accessed only occasionally, averaging once every ten seconds, and page replacement is very rare, averaging once a day.

i. Does the Clock algorithm degenerate to another (non-LRU) policy (such as FIFO or RAND) under these circumstances? If so, state what policy it becomes and why.

The clock algorithm degenerates to FIFO because by the time page replacement occurs, every in-memory page will be marked as having been accessed.

ii. Does the FPB algorithm degenerate to another (non-LRU) policy (such as FIFO or RAND) under these circumstances? If so, state what policy it becomes and why.

No, the FPB algorithm does not degenerate. While the clock algorithm fails in this case, because it cannot find any pages still marked as inactive after a day of activity, FPB does not suffer from this problem because it always keeps a certain

percentage of pages on the free queue. Thus it provides a good approximation of LRU in this case.

(d) Which algorithm provides a better approximation of the LRU policy? Why?

From the examples considered in the question, FPB seems to do better. FPB benefits from keeping a fixed portion of frames marked as free, and from the way it follows an oldest-first (FIFO) strategy for choosing the next frame to add to the free queue.

However, it is worth remembering that FPB is not *always* better (programs that suddenly demand a lot of frames at once may exhaust the free queue and cause FPB to degenerate to FIFO, whereas the clock algorithm would be able to find enough free frames).

W12. Devise a question suitable for use in an examination on operating systems. State the question and give a model answer. Your question should:

- Require about five to seven minutes for an average student to read, consider, and answer.
- Require the *application* of operating systems knowledge—not be a simple matter of factual recall.
- Have a straightforward answer that can be assessed quickly.
- Be original (not found in your textbook or very similar to a question suggested by your friends).

Your score for this question will be based both on your question and your answer. You may not receive full credit if your question is too easy or too hard.

(If your question is a good one, I may adapt it for use on either a sample final exam given out to students, or on the actual final. You may have an advantage over your fellow students if you have to answer your own question on the final exam, so you have a special incentive to produce a good question.)

I'll provide several answers to this question on your final exam! Most students gave excellent answers to this question.