CS111 Midterm Exam

Michael Warsah Liu

TOTAL POINTS

86 / 110

QUESTION 1

Principles 10 pts

1.1 Define Info Hiding 2 / 2

✓ **- 0 pts Correct**

1.2 Value of Info Hiding 3 / 3

✓ **- 0 pts Correct**

1.3 Define Modularity 2 / 2

✓ **- 0 pts a system is composed of distinct sub-components**

1.4 Good Modularity 3 / 3

✓ **- 0 pts Correct**

QUESTION 2

ABIs and APIs 10 pts

2.1 ABI acronym 2 / 2

✓ **- 0 pts Application Binary Interface**

2.2 ABI definition 0 / 3

✓ **- 3 pts wrong**

2.3 ABI vs API 2 / 2

✓ **- 0 pts without recompilation, one binary program works on all compliant platforms**

2.4 When API over ABI 3 / 3

✓ **- 0 pts They are using different instruction set architectures.**

QUESTION 3

Libraries 10 pts

3.1 Static library - advantages 0 / 3

✓ **- 3 pts no need to implement or explicitly include the module in the compilation or linkage edit.**

This is not different from explicitly included object modules

3.2 Which modules loaded 3 / 3

✓ **- 0 pts Correct**

3.3 Shared library - advantages 2 / 4

- 2 Point adjustment

(2) not an advantage of shared over static

QUESTION 4

Multi-Level Queues 10 pts

4.1 What problem they solve 2 / 2

✓ **- 0 pts Correct**

4.2 What drives queue changes 4 / 4

✓ **- 0 pts Correct**

4.3 Consequences of wrong queue 4 / 4

✓ **- 0 pts Correct**

QUESTION 5

Fixed Paritition Allocation 10 pts

5.1 problem with it 3 / 3

✓ **- 0 pts Internal Fragmentation**

5.2 effect of special sub-pools 0 / 3

✓ **- 3 pts incorrect**

5.3 problem with special sub-pools 0 / 2

✓ **- 2 pts incorrect**

5.4 preventing that problem 0 / 2

✓ **- 2 pts incorrect**

QUESTION 6

Paging MMU 10 pts

6.1 diagram MMU, translation 5 / 5

✓ **- 0 pts Correct**

6.2 info in page table entry 2 / 3

✓ **- 1 pts Only one information was correctly described.**

6.3 motivation for TLA buffers 2 / 2

✓ - 0 pts Correct

QUESTION 7

Synchronization Terminology 10 pts

7.1 indeterminate 0 / 2

✓ - 2 pts Incorrect

7.2 non-deterministic 1 / 2

✓ - 1 pts unclear

What is the cause of randomness?

7.3 race condition 1 / 2

✓ - 1 pts too close to critical section

7.4 critical secstion 2 / 2

✓ - 0 pts Correct

7.5 atomicity 2 / 2

✓ - 0 pts Correct

QUESTION 8

Correct locking criteria 10 pts

8.1 criteria and mechanisms that fails 4 / 4

✓ - 0 pts Correct

8.2 criteria and mechanisms that fails 3 / 3

✓ - 0 pts Correct

8.3 criteria and mechanisms that fails 2 / 3

- 1 Point adjustment

condition variables are not a locking mechanism

QUESTION 9

Asynchronous Completion mechanisms 10 pts

9.1 semaphores vs condition variables 3 / 3

✓ - 0 pts Correct

9.2 why they differ 2 / 3

✓ - 1 pts In a counting semaphore, the count can represent resource/completion availability, and a successful P grants the resource to the recipient. Mandatory queuing ensures no other process can take that resource.

9.3 when choose semaphores 2 / 2

✓ - 0 pts Correct

9.4 when choose condition variables 2 / 2

✓ - 0 pts Correct

QUESTION 10

Enforced locking 10 pts

10.1 advantage of enforced 4 / 4

✓ - 0 pts Correct

10.2 when choose advisory 4 / 4

✓ - 0 pts Correct

10.3 requirement for enforced 1 / 2

✓ - 1 pts clients cannot directly access the protected object without going through methods that enforce locking

QUESTION 11

Clock Algorithms 10 pts

11.1 what problem they solve 1 / 2

✓ - 1 pts 1. finding the absolutely LRU element in a very large list involves a very long and expensive search. 2. updating a time on every reference would greatly slow down the system.

11.2 elements of clock algorithm 2 / 2

✓ - 0 pts Correct

11.3 refrigerator LRU algorithm 2 / 2

✓ - 0 pts Correct

11.4 approximation of LRU 2 / 2

✓ - 0 pts Correct

11.5 regrigerator working set algorithm 2 / 2

✓ - 0 pts Correct

CS111 Midterm 05/03/2018

Name Michael Liu

Student ID # 104 778 500

Seat Row 4 Seat Col 8 Exam # 75

This is a closed-book, no-notes exam.

All questions are of equal value. Most questions have multiple parts. You must answer every part of every question. Read each question CAREFULLY; Make sure that

you understand EXACTLY what question is being asked

what type of answer is expected

your answer clearly and directly responds to the asked question

Many students lose many points for answering questions other than the one I asked. Misunderstanding a question may be evidence that you have not mastered the underlying concepts.

If you are unsure about what a question is asking for, raise your hand and ask.

Spend more time thinking and less time writing. Short and clear answers get more credit than long, rambling or vague ones.

Write carefully. I do not grade for penmanship, spelling or grammar, but if I cannot read or understand your answer, I can't give you credit for it.

1: (a) Define "Information-Hiding" (in the context of s/w design):

encapsulation is the practice of hiding implementation details irrelevant to the users of a software module.

(b) Briefly explain why Information-Hiding is a good thing:

- Users don't need to know or understand the implementation: they only need to look at the interface of the module to use it.
- It's easy to change the implementation of a software module later if we follow the same interface.

(c) Define "Modularity" (without using the word "module"):

splitting a larger functionality into logically discrete parts that work together, and whose implementations are independent of each others'.

(d) Briefly describe a (covered in this course) characteristic of good modular decomposition:

- logically distinct responsibilities
- little overlap of function
- parts able to be interchanged freely

2: (a) What does the acronym "ABI" stand for?

Application Binary Interface.

(b) Define the term?

The machine code that can be executed on a particular processor.

(c) Why is ABI compatibility preferable to API compatibility?

Software developers don't need to recompile their software in order to distribute their product on ABI-compatible machines. While they do for API-compatibility.

(d) When might it be necessary or reasonable for two OSs that support the same APIs to not support the same ABIs?

When the OSes want to implement the same API methods for processors using different ISAs.

3: (a) Give one advantage of static (non-shared) libraries over user-supplied object modules.

All the necessary library code is packaged in one binary, so there's no risk of missing libraries at runtime.

(b) What determines WHICH object modules FROM WHICH libraries become incorporated into a load module?

the linker walks a predefined registry (either a well-known path or user-supplied) of libraries, and tries to match each missing method symbol in the sym-table with the first matching name of one of the methods in the traversed libraries (so search order matters), and it only loads the relevant code (not the whole library).

(c) Briefly list two advantages of shared libraries over ordinary libraries?

- memory savings. the same shared lib code can be mapped into the addr spaces of multiple programs (read-only).

- reliability. using a well-known and presumably well-tested shared library gives users (and the developer) some peace of mind regarding the correctness of the library code.

4: (a) What fundamental problem (or truth about processes) motivates the use of multi-level feedback queues for process scheduling?

there is a natural time-slice period for each process which is difficult to determine without a dynamic process.

(b) State TWO DISTINCT ways a process might find its way onto the right queue.

- it yields a lot, and goes above the high watermark for its current queue → this moves it to a higher (lower quantum) qu
- it regularly gets preempted, and goes under the low watermark for its current queue → this moves it to a lower (higher quantum) queue.

(c) What would be the negative consequences of a process being on the wrong queue (provide an answer for wrong in each direction)?

(c1) Too high up → process might be a workload-heavy data processing job that would really appreciate it if the darn scheduler would stop preempting it and invalidating its cache. Processing the job might be slower in this case. (e.g. UI rendering)

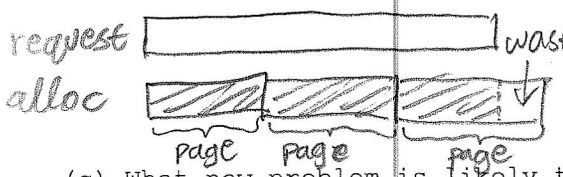
(c2) Too low down → process is probably an interactive job with a real human person who's just wondering why the response time is so darn slow. (High latency on interactive jobs)

5: (a) What is the primary problem associated with fixed-partition memory allocation (returning fixed sized regions that may be larger than the requested size)?

Internal fragmentation (unused space is wasted space).

(b) Briefly explain how special pools of fixed-size buffers affect this problem?

This is paging. With paging, rather than returning one big chunk, with an average wastage of half the allocated memory, we return smaller, fixed-size chunks such that wastage is on average half of the size of one (i.e. the last) page.



(c) What new problem is likely to arise when we create such pools?

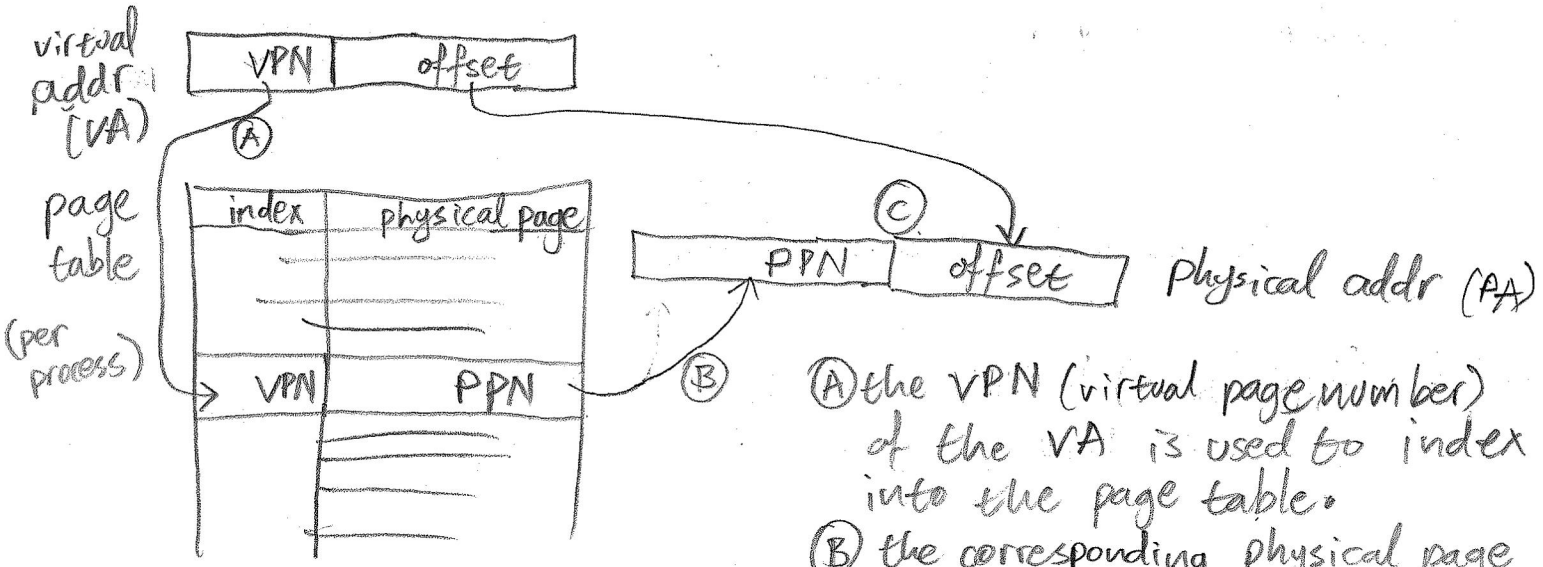
We need a way to keep track of which pages have been allocated to which processes.

(d) Briefly describe an approach for dealing with that problem?

We can have a page table that stores the addresses of the pages, along with relevant information like the owning process.

simple pic, simple expl.

6: (a) Draw a diagram of a paging MMU, and illustrating how it translates a virtual address into a physical address.



(A) the VPN (virtual page number) of the VA is used to index into the page table.
(B) the corresponding physical page number is extracted.
(C) the PPN is combined with the offset to get the PA.

(b) List (and briefly describe) two key pieces of information (other than the physical page frame number) that one might find in a page table entry

present bit: whether the page is in memory or is in swap space.

valid bit: whether the process is allowed to reference this page.

(c) Given the (relative simplicity) of paged virtual address translation, why has it been necessary to create Translation Look-Aside Buffers?

speed. memory access to ~~get the~~ translate a VA into a PA is extremely expensive, ~~VA~~ TLBs allow us to cache common/recent translations in a fast and small register, so we avoid having to access the memory twice for every useful memory access.

☆2.

7: Define (and distinguish the differences between) the following terms:

(a) "indeterminate"

program follows unknown behavior.

(b) "non-deterministic" ... distinguish from "indeterminate"

program follows known behavior that follows a random process.

(c) "race condition"

when correctness of program depends on timing.

(d) "critical section" ... distinguish from "race condition"

a section of code with enforced serial access.
race conditions may call for a critical section.

(e) "atomicity"

an "atomic" procedure either (1) all happens at once
(2) doesn't happen

i.e. no *partial execution.

8: The text gave three criteria in terms of which lock mechanisms should be evaluated. In class this list was expanded to four criteria. List and briefly describe three of those criteria AND provide an example of a real locking mechanism that does poorly on each criteria ... briefly explaining why it does poorly.

(a) correctness: whether the mechanism actually solves the concurrency issue we have.
interrupt disables don't work correctly in a multi-processor environment because a process from another core can concurrently access critical section

(b) performance: how much we pay in terms of latency and/or throughput in order to implement the mechanism.
spin-locks can possibly have good performance in very limited scenarios but in most cases, they waste CPU cycles and even delays the completion of the condition they're waiting for.

(c) fairness: is every thread guaranteed to have a turn?
is the waiting time bounded?
condition variables can actually do quite poorly here, if the wake() method (/signal()) wakes up a random waiting thread. Since it's random, no thread is guaranteed a turn.

9: The text discussed both semaphores and condition variables as mechanisms that could be used to implement asynchronous event notification and waiting.
(a) Describe an important difference in what the waiter can assume after resuming after wait on a counting semaphore and on a condition variable.

semaphore: Hoare semantics. desired state is guaranteed on wake.

cond vars: Mesa semantics. can't assume that state is as desired. need to wait in a while loop that checks state.

(b) Briefly explain why semaphores and condition variables are different in this respect.

semaphores are like FIFO queues, while cond vars let any arbitrary thread run conditionally on the state. This gives the programmer flexibility: does he want threads to run in FIFO order, thus maintaining some fairness, or does he not care?

(c) Briefly describe a situation where this difference would make semaphores a better choice.

When we want to ensure fairness, i.e. make sure that every thread gets a turn, maybe in batch systems where every process is a long-running one and compute-time fairness is important.

(d) Briefly describe a situation where this difference would make condition variables a better choice.

when we care more about performance, we can allow arbitrary threads to do work conditional on the state using cond vars. This extra flexibility allows further optimization. Perhaps use cond vars in producer-consumer systems, where we don't care which thread fills or empties the buffer, as long as it's done quickly.

10: (a) What is the primary advantage of "enforced" (vs advisory) locking?

It's enforced, meaning that we don't have to trust other processes to not access our super important data, i.e. the system guarantees this.

(b) Describe a problem characteristic that would make "advisory" preferable?

Say there's a parent thread which spawns a few child threads, and the parent periodically gets data from its children. But sometimes, it would like this data faster, so it attempts to read directly from some files used by the children to store work. Advisory locking lets the parent decide whether it wants to read this data even as the child is using the file.

(c) What is required to make it possible to "enforce" locking?

OS support is needed. There should be a system call that lets the OS know when a process wants to enforce a lock on a file, and the OS should reject access requests for this file from other processes.

XC: (a) What otherwise difficult/expensive problems (be very specific) do "clock algorithms" address?

Problems where getting a 'good enough' approximation of the oldest elemen means massive ~~$$~~ cost (compute-time) savings. Mainly when list of items is large and we need to choose often.

(b) What are the key elements of a "clock algorithm"?

- A list of all elements (circular)
- An indicator field on every element in list (binary/count)
- A pointer to 'age' elements one at a time
- A process that 'renews' elements when used

(c) Briefly describe an LRU Clock Algorithm for deciding what old thing to remove from your refrigerator to make room for a new thing. I specifically want to understand how you implement your progressive scan, and what your "recently used" test is.

- keep a circular list of all items in fridge, with a 'Y'/'N' indicator field beside each item, and a pointer to the first item.
- whenever I use an item (and put it back in), flip indicator to 'Y'
- when I need to throw a thing away, repeat:
  - if currently pointed to item is 'N', select this to throw.
  - else, flip to 'N', advance pointer.

(d) Explain why your progressive clock-scan yields a reasonable approximation of Least Recently Used.

Frequently-used items tend to also not be LRU, and tend to have a 'Y' field, so they tend to not be thrown out. Conversely, not using an item for a long time means that the field will eventually decay to a 'N' and the item will be thrown out.

(e) Not unlike Global LRU, this seems a clumsy mechanism, in that it imposes a more-or-less constant replace-by age on all items, even though some expire in days while others are good for years. Describe changes to your scan algorithm and "recently used" test to implement "Working Set Clock" replacement. I specifically want to understand your progressive scan, what your "recently used" test is, and how you set the key comparison parameters.

- change indicator ('Y'/'N') to a counter, initialized to the # days till item expiry.
- every time I use an item, increment counter.
- every time we want to choose an item to throw, repeat:
  - if counter is 0, choose this.
  - else, decrement counter and advance pointer.