Section 8.5: Extra Section Potpourri

CS 162

October 23, 2020

Contents

Note: These questions are usually included on section worksheets in previous semesters. In an effort to make section worksheets more manageable, we have omitted these questions from previous worksheets because these would be questions that TAs would likely have not had time for in a normal section. These questions are still in scope for the midterm, but are generally not as emphasized as opposed to other topics.

1 Advanced Scheduling

In what sense is CFS completely fair?

CFS attempt to give all processes equal access to the CPU. Ideally, all threads get concurrent access to their share of the CPU.

This fairness can be reweighted with the notion of niceness.

How do we easily implement Lottery scheduling?

Given that each thread is allocated their own set number of tickets totalling N tickets across all threads, we can select at random a number between 1 through N. This number would fall into a bin defined by the number of tickets per thread and that thread would then get to run.

Is Stride scheduling prone to starvation?

No, Stride scheduling tries to achieve proportional shares to the CPU so a thread will eventually get a chance at running. Concretely, if all other threads' pass value are strictly increasing, then even the lowest priority thread will get a chance at running.

In Stride scheduling, if a job is more urgent, should it be assigned a larger stride or a smaller stride?

Smaller stride. Lower stride jobs run more often as their pass value increases at a slower rate.

2 Page Fault Handling for Pages Only On Disk

The page table maps VPN to PPN, but what if the page is not in main memory and only on disk? Think about structures/bits you might need to add to the page table/OS to account for this. Write pseudocode for a page fault handler to handle this.

```
Have a disk map structure that contains a disk address, and process id
for each ppn. Have each process be associated with a page table. Each of
these two tables describes the entire virtual memory address space, and
physical memory address space, respectively. The page table identifies
which ppn is associated with which vpn, and contains bits such as used,
modified, and presence to describe whether or not it is in physical
memory or only on disk. The disk map the corresponding disk address for
each ppn. The entire address space is on the disk, but only a subset of
it is resident in main memory.

page fault:
index: vpn, value: ppn

frame table:
index: ppn, value: process id, disk address

page table entry:
p|u|m|f

p = presence flag
u = used flag
m = modified flag
f = page frame (ppn)

disk table entry:
pid | disk address | bits/metadata for replacement algorithm

Page Fault Handler Pseudocode:

1. Using the replacement algorithm, iterate through the disk table and get
the number of a frame that will be used for the incoming page
2. Swap the page currently in that frame to its slot on the disk
3. Swap the requested page from its slot on disk into the above frame
4. Update the page table entry so that vpn -> ppn and the presence flag is
set to true (since it's now in main memory)

return
```

3 Inverted Page Tables

3.1 Inverted Page Table

Why IPTs? Consider the following case:
- 64-bit virtual address space
- 4 KB page size
- 512 MB physical memory

How much space (memory) needed for a single level page table? Hint: how many entries are there? 1 per virtual page. What is the size of a page table entry? access control bits + physical page #.

```
One entry per virtual page

 - 2^64 addressable bytes / 2^12 bytes per page = 2^52 page table entries

 Page table entry size

 - 512 MB physical memory = 2^29 bytes
- 2^29 bytes of memory/2^12 bytes per page = 2^17 physical pages
- 17 bits needed for physical page number
{ Page table entry = ~4 bytes
- 17 bit physical page number = ~3 bytes
- Access control bits = ~1 byte

Page table size = page table entry size * # total entries

{ 2^52 page table entries * 2^2 bytes = 2^54 bytes (16 petabytes)

i.e. A WHOLE LOT OF MEMORY
```

How about multi level page tables? Do they serve us any better here?

What is the number of levels needed to ensure that any page table requires only a single page (4 KB)?

```
{ Assume page table entry is 4 bytes
{ 4 KB page / 4 bytes per page table entry =
1024 entries
{ 10 bits of address space needed
{ ceiling(52/10) = 6 levels needed

7 memory accesses to do something? SLOW!!!
```

3.2 Linear Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

```
{ add up all bits = 80 bits = 10 bytes
- 10 bytes * # of physical pages = 10 * 2^17 = 2^3 * 2^17 = 1 MB

Iterate through all entries.
For each entry in the inverted page table,
compare process ID and virtual page
number in entry to the requested process
ID and virtual page number
Extremely slow. must iterate through 2^17 entries of the hash table
worst-case scenario.
```

3.3 Hashed Inverted Page Table

What is the size of of the hashtable? What is the runtime of finding a particular entry?

Assume the following:
- 16 bits for process ID
- 52 bit virtual page number (same as calculated above)
- 12 bits of access information

```
{ add up all bits = 80 bits = 10 bytes
- 10 bytes * # of physical pages = 10 * 2^17 = 2^3 * 2^17 = 1 MB

Linear inverted page tables require too many
memory accesses.
- Keep another level before actual inverted page
table (hash anchor table)
{ Contains a mapping of process ID and virtual page
number to page table entries
- Use separate chaining for collisions
- Lookup in hash anchor table for page table entry
{ Compare process ID and virtual page number
- if match, then found
- if not match, check the next pointer for another page table
entry and check again

So, with a good hashing scheme and a hashmap proportional to
the size of physical memory, O(1) time. Very efficient!
```

4 Second Chance List Algorithm

Suppose you have four pages of physical memory: 00, 01, 10, 11 and five pages of virtual memory: 000, 001, 010, 011, 101. Run the second chance list algorithm, with two physical pages delegated to the active list and two physical pages delegated to the second chance list.

The access pattern (assume we write to these pages) for virtual pages is as follows:

Page	000	001	010	011	000	101

The page table looks like this at the start of the algorithm.

Virtual Page	Physical Page	Extra
000		PAGEOUT
001		PAGEOUT
010		PAGEOUT
011		PAGEOUT
101		PAGEOUT

The 'extra' bits should read 'RW', 'INVALID', 'FIFO: 0', 'LRU: 0' where the bit for FIFO / LRU is 0 for more recent and 1 for less recent.

4.1 After Writes to '000', '001'

What does the table look like after these first two writes?

Virtual Page	Physical Page	Extra
000	00	RW, FIFO: 1
001	01	RW, FIFO: 0
010		PAGEOUT
011		PAGEOUT
101		PAGEOUT

4.2 After Writes to '010', '011'

What does the table look like after the next two writes?

Virtual Page	Physical Page	Extra
000	00	INVALID, LRU: 1
001	01	INVALID, LRU: 0
010	10	RW, FIFO: 1
011	11	RW, FIFO: 0
101		PAGEOUT

4.3 After Write to '000'

What happens when you write to virtual page 000? What does the table look like after this write?

Virtual page 000 is translated to physical page 00. However, this page is marked as invalid, so the page fault handler will be invoked. The handler will use the FIFO replacement algorithm on the active list to find a physical page to evict, and insert that page into the second-chance list. Then physical page 00 is inserted into the active list.

Virtual Page	Physical Page	Extra
000	00	RW, FIFO: 0
001	01	INVALID, LRU: 1
010	10	INVALID, LRU: 0
011	11	RW, FIFO: 1
101		PAGEOUT

4.4 After Write to '101'

What happens when you write to virtual page 101? What does the table look like after this write?

Virtual page 101 does not have a physical page assigned, so this will be a page fault. The page fault handler will use the LRU replacement algorithm on the second chance list to find a physical page to evict, then use this physical page for 101.

Next, it will use the FIFO replacement algorithm on the active list to find a physical page to evict, and insert that page into the second-chance list.

In this case, virtual page 001 is paged out, and virtual page 101 now maps to physical page 01. Finally, physical page 01 is inserted into the newly empty slot in the active list.

Virtual Page	Physical Page	Extra
000	00	RW, FIFO: 1
001		PAGEOUT
010	10	INVALID, LRU: 1
011	11	INVALID, LRU: 0
101	01	RW, FIFO: 0