Section 8: Clock Algorithm, Second Chance List Algorithm, and Intro to I/O

CS 162

March 20, 2020

Contents

1 Vocabulary

- **Demand Paging** The process where the operating system only stores pages that are "in demand" in the main memory and stores the rest in persistent storage (disk). Accesses to pages not currently in memory page fault and the page fault handler will retrieve the request page from disk (paged in). When main memory is full, then as new pages are paged in old pages must be paged out through a process called eviction. Many cache eviction algorithms like least recently used can be applied to demand paging, the main memory is acting as the cache for pages which all start on disk.
- **Working Set** The subset of the address space that a process uses as it executes. Generally we can say that as the cache hit rate increases, more of the working set is being added to the cache.
- **Resident Set Size** The portion of memory occupied by a process that is held in main memory (RAM). The rest has been paged out onto disk through demand paging.
- **Thrashing** Phenomenon that occurs when a computer's virtual memory subsystem is constantly paging (exchanging data in memory for data on disk). This can lead to significant application slowdown.
- **Clock Algorithm**: An approximation of LRU. Main idea: replace *an* old page, not the *oldest* page. On a page fault, check the page currently pointed to by the 'clock hand. Checks a use bit which indicates whether a page has been used recently; clears it if it is set and advances the clock hand. Otherwise, if the use bit is 0, selects this candidate for replacement.

  Other bits used for Clock: "modified"/"dirty" indicates whether page must be written back to disk upon pageout; "valid" indicates whether the program is allowed to reference this page; "read-only"/"writable" indicates whether the program is allowed to modify this page.
- **Nth Chance Algorithm**: An approximation of LRU. A version of Clock Algorithm where each page gets N chances before being selected for replacement. The clock hand must sweep by N times without the page being used before the page is replaced. For a large N, this is a very good approximation of LRU.
- **Second-Chance List Algorithm**: An approximation of LRU. Divides pages into two - an active list and a second-chance list. The active list uses a replacement policy of FIFO, while the second-chance list uses a replacement policy of LRU. Not required reading, but if you're interested in the details, this algorithm is covered in detail in this paper: https://users.soe.ucsc.edu/~sbrandt/221/Papers/Memory/levy-computer82.pdf. The version presented in lecture and for the purposes of this course includes some significant simplifications.
- **Inverted Page Table** - The inverted page table scheme uses a page table that contains an entry for each phiscial frame, not for each logical page. This ensures that the table occupies a fixed fraction of memory. The size is proportional to physical memory, not the virtual address space. The inverted page table is a global structure – there is only one in the entire system. It stores reverse mappings for all processes. Each entry in the inverted table contains has a tag containing the task id and the virtual address for each page. These mappings are usually stored in associative memory (remember fully associative caches from 61C?). Associatively addressed memory compares input search data (tag) against a table of stored data, and returns the address of matching data. They can also use actual hash maps.
- **I/O** In the context of operating systems, input/output (I/O) consists of the processes by which the operating system receives and transmits data to connected devices.

- **Controller** The operating system performs the actual I/O operations by communicating with a device controller, which contains addressable memory and registers for communicating the the CPU, and an interface for communicating with the underlying hardware. Communication may be done via programmed I/O, transferring data through registers, or Direct Memory Access, which allows the controller to write directly to memory.
- **Interrupt** One method of notifying the operating system of a pending I/O operation is to send a interrupt, causing an interrupt handler for that event to be run. This requires a lot of overhead, but is suitable for handling sporadic, infrequent events.
- **Polling** Another method of notifying the operating system of a pending I/O operating is simply to have the operating system check regularly if there are any input events. This requires less overhead, and is suitable for regular events, such as mouse input.
- **Response Time** Response time measures the time between a requested I/O operating and its completion, and is an important metric for determining the performance of an I/O device.
- **Throughput** Another important metric is throughput, which measures the rate at which operations are performed over time.
- **Asynchronous I/O** For I/O operations, we can have the requesting process sleep until the operation is complete, or have the call return immediately and have the process continue execution and later notify the process when the operation is complete.
- **Memory-Mapped I/O** Memory-mapped I/O (not to be confused with memory-mapped file I/O) uses the same address bus to address both memory and I/O devices – the memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is accessed by the CPU, it may refer to a portion of physical RAM, but it can also refer to memory of the I/O device. Thus, the CPU instructions used to access the memory can also be used for accessing devices.

2 Demand Paging

2.1 Demand Paging

An up-and-coming big data startup has just hired you do help design their new memory system for a byte-addressable system. Suppose the virtual and physical memory address space is 32 bits with a 4KB page size.

Suppose you know that there will only be 4 processes running at the same time, each with a Resident Set Size (RSS) of 512MB and a working set size of 256KB. What is the minimum amount of TLB entries that your system would need to support to be able to map/cache the working set size for one process? What happens if you have more entries? What about less?

A process has a working set size of 256KB which means that the working set fits in 64 pages. This means our TLB should have 64 entries. If you have more entries, then performance will increase since the process often has changing working sets, and it should be able to store more in the TLB. If it has less, then it can't easily translate the addresses in the working set and performance will suffer.

Suppose you run some benchmarks on the system and you see that the system is utilizing over 99% of its paging disk IO capacity, but only 10% of its CPU. What is a combination of the of disk space and memory size that can cause this to occur? Assume you have TLB entries equal to the answer from the previous part.

The CPU can't run very often without having to wait for the disk, so it's very likely that the system is thrashing. There isn't enough memory for the benchmark to run without the system page faulting and having to page in new pages. Since there will be 4 processes that have a RSS of 512MB each, swapping will occur as long as the physical memory size is under 2GB. This happens regardless of the number of TLB entries and disk size. If the physical memory size is lower than the aggregate working set sizes, thrashing is likely to occur.

Out of increasing the size of the TLB, adding more disk space, and adding more memory, which one would lead to the largest performance increase and why?

We should add more memory so that we won't need to page in new pages as often.

3 Clock Algorithm

3.1 Clock Page Table Entry

Suppose that we have a 32-bit virtual address split as follows:

10 Bits Table ID	10 Bits Page ID	12 Bits Offset

Assume that the physical address is 32-bit as well. Show the format of a page table entry (PTE) complete with bits required to support the clock algorithm.

20 Bits PPN	8 Bits Other	1 Bit Dirty	1 Bit Use	1 Bit Writable	1 Bit Valid

3.2 Clock Algorithm Step-through

For this problem, assume that physical memory can hold at most four pages. What pages remain in memory at the end of the following sequence of page table operations and what are the use bits set to for each of these pages?

Page	A	B	C	A	C	D	B	D	A	E	F

E: 1, F: 1, C: 0, D: 0
Recall that the clock hand only advances on page faults. No page replacement occurs until $t = 10$, when all pages are full. At $t = 10$, all pages have the use bit set. The clock hand does a full sweep, setting all use bits to 0, and selects page 1 (currently holding A) to be paged out. The clock hand advances and now points to page 2 (currently holding B). At $t = 11$, we check page 2's use bit, and since it is not set, select page 2 to be paged out. F is brought in to page 2. The clock hand advances and now points to page 3. We reach the end of the input and end.
Note: The table shows the clock hand position before page faults occur.

Page	A	B	C	A	C	D	B	D	A	E	F
1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	E: 1	E: 1
2		B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 0	F: 1
3			C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 0	C: 0
4						D: 1	D: 1	D: 1	D: 1	D: 0	D: 0
Clock	1	2	3	4	4	4	1	1	1	1	2

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

5 Inverted Page Tables

5.1 Inverted Page Tables

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!!!
```

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.
```

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!
```

6 Input/Output

6.1 Warmup

1. (True/False) If a particular IO device implements a blocking interface, then you will need multiple threads to have concurrent operations which use that device.

   True. Only with non-blocking IO can you have concurrency without multiple threads.

2. (True/False) For I/O devices which receive new data very frequently, it is more efficient to interrupt the CPU than to have the CPU poll the device.

   False. It is more efficient to poll, since the CPU will get overwhelmed with interrupts.

3. (True/False) With SSDs, writing data is straightforward and fast, whereas reading data is complex and slow.

   False, it is the opposite. SSD's have complex and slower writes because their memory can't be easily mutated.

4. (True/False) User applications have to deal with the notion of file blocks, whereas operating systems deal with the finer grained notion of disk sectors.

   False, blocks are also an OS concept and are not exposed to users.

6.2 I/O Devices

What is a block device? What is a character device? Why might one interface be more appropriate than the other?

Both of these are types of interfaces to I/O devices. A block device accesses large chunks of data (called blocks) at a time. A character device accesses individual bytes at a time. A block device interface might be more appropriate for a hard drive, while a character device might be more appropriate for a keyboard or printer.

Why might you choose to use DMA instead of memory mapped I/O? Give a specific example where one is more appropriate than the other.

DMA is more appropriate when you need to transfer large amounts of data to/from main memory without occupying the CPU, especially when the operation could potentially take a long time to finish (accessing a disk sector, for example). The DMA controller will send an interrupt to the CPU when the DMA operation completes, so the CPU does not need to waste cycles polling the device. While memory mapped I/O can be used to transfer device data into main memory, it must involve the CPU. Memory mapped I/O is useful for accessing devices directly from the CPU (writing to the frame buffer or programming the interrupt vector, for example).

Explain what is meant by "top half" and "bottom half" in the context of device drivers.

The top half of a device driver is used by the kernel to start I/O operations. The bottom half of a device driver services interrupts produced by the device. You should know that Linux has different definitions for "top half" and "bottom half", which are essentially the reverse of these definitions (top half in Linux is the interrupt service routine, whereas the bottom half is the kernel-level

bookkeeping).