Section 8: Performance and Address Translation

CS 162

July 20, 2020

Contents

1 Vocabulary **2**

2 Queuing Theory **3**

3 Paging and Address Translation **4**

3.1 Conceptual Questions 4

3.2 Page Allocation 5

3.3 Address Translation 7

3.4 Page Fault Handling for Pages Only On Disk 8

1 Vocabulary

- **Queuing Theory** Here are some useful symbols: (both the symbols used in lecture and in the book are listed)
  - μ is the average service rate (jobs per second)
  - T_{ser} or S is the average service time, so $T_{ser} = \frac{1}{\mu}$
  - λ is the average arrival rate (jobs per second)
  - U or u or ρ is the utilization (fraction from 0 to 1), so $U = \frac{\lambda}{\mu} = \lambda S$
  - T_q or W is the average queuing time (aka waiting time) which is how much time a task needs to wait before getting serviced (it does not include the time needed to actually perform the task)
  - L_q or Q is the average length of the queue, and it's equal to λT_q (this is Little's law)
- **Virtual Memory** - Virtual Memory is a memory management technique in which every process operates in its own address space, under the assumption that it has the entire address space to itself. A virtual address requires translation into a physical address to actually access the system's memory.
- **Memory Management Unit** - The memory management unit (MMU) is responsible for translating a process' virtual addresses into the corresponding physical address for accessing physical memory. It does all the calculation associating with mapping virtual address to physical addresses, and then populates the address translation structures.
- **Address Translation Structures** - There are two kinds you learned about in lecture: segmentation and page tables. Segments are linearly addressed chunks of memory that typically contain logically-related information, such as program code, data, stack of a single process. They are of the form (s,i) where memory addresses must be within an offset of i from base segment s. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are used by the accessing process, while physical addresses are used by the hardware or more specifically to the RAM.
- **Translation Lookaside Buffer (TLB)** - A translation lookaside buffer (TLB) is a cache that memory management hardware uses to improve virtual address translation speed. It stores virtual address to physical address mappings, so that the MMU can store recently used address mappings instead of having to retrieve them mutliple times through page table accesses.

2 Queuing Theory

Explain intuitively why response time is nonlinear with utilization. Draw a plot of utilization (x axis) vs response time (y axis) and label the endpoints on the x axis.

Even with high utilization (99%), some of the time (1%), the server is idle, which is a waste. All this wasted time adds up, and in the steady state, the queue becomes very long.
Graph should be linear-ish close to $u = 0$ and grow asymptotically toward ∞ at $u = 1$.

If 50 jobs arrive at a system every second and the average response time for any particular job is 100ms, how many jobs are in the system (either queued or being serviced) on average at a particular moment? Which law describes this relationship?

$50\ jobs/s \times 0.1s = 5\ jobs$ (5 jobs at any time). This is Little's law - arrival rate $\times$ average response time = average length of the queue.

Is it better to have N queues, each of which is serviced at the rate of 1 job per second, or 1 queue that is serviced at the rate of N jobs per second? Give reasons to justify your answer.

One queue that can process N jobs per second is faster (i.e the second option). Better response time ($\frac{1}{N}$ sec vs 1 sec) and better utilization (no load-balancing problems), which gives you lower queuing delays on average.

What is the average queueing time for a work queue with 1 server, average arrival rate of λ, average service time S, and squared coefficient of variation of service time $\mathbf{C}$?

(Recall from definitions that we sometimes swap between symbols - S is another name for T_{ser}.)
$T_q = T_{ser}(\frac{u}{1-u})(\frac{C+1}{2})$ where $u = \lambda T_{ser}$

What does it mean if $\mathbf{C} = 0$? What does it mean if $\mathbf{C} = 1$?

If $\mathbf{C} = 0$, then your service rate is regular and deterministic, which means that tasks are completed at a constant rate.
If $\mathbf{C} = 1$, then your service rate can be modeled as a Poisson distribution, and the interval between jobs being serviced can be modeled as a exponential distribution.

3 Paging and Address Translation

3.1 Conceptual Questions

If the physical memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

Increases by 1 bit. Assuming the page size remains the same, there are now twice as many physical pages, so the physical page number needs to expand by 1 bit.

If the physical memory size (in bytes) is doubled, how does the number of entries in the page table change?

No change. The number of entries in the page table is determined by the size of the virtual address and the size of a page – it's not affected by the size of physical memory.

If the virtual memory size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

No change. The number of bits in a page table entry is determined by the number of control bits (usually 2: dirty and resident) and the number of physical pages – the size of each entry is not affected by the size of virtual memory.

If the virtual memory size (in bytes) is doubled, how does the number of entries in the page map change?

The number of entries doubles. Assuming the page size remains the same, there are now twice as many virtual pages and so there needs to be twice as many entries in the page map.

If the page size (in bytes) is doubled, how does the number of bits in each entry of the page table change?

Each entry is one bit smaller. Doubling the page size while maintaining the size of physical memory means there are half as many physical pages as before. So the size of the physical page number field decreases by one bit.

If the page size (in bytes) is doubled, how does the number of entries in the page table change?

There are half as many entries. Doubling the page size while maintaining the size of virtual memory means there are half as many virtual pages as before. So the number of page table entries is also cut in half.

The following table shows the first 8 entries in the page table. Recall that the valid bit is 1 if the page is resident in physical memory and 0 if the page is on disk or hasn't been allocated.

Valid Bit	Physical Page
0	7
1	9
0	3
1	2
1	5
0	5
0	4
1	1

If there are 1024 bytes per page, what is the physical address corresponding to the hexadecimal virtual address 0xF74?

The virtual page number is 3 with a page offset of 0x374. Looking up page table entry for virtual page 3, we see that the page is resident in memory (valid bit = 1) and lives in physical page 2. So the corresponding physical address is `(2<<10)+0x374 = 0xB74`

3.2 Page Allocation

Suppose that you have a system with 8-bit virtual memory addresses, 8 pages of virtual memory, and 4 pages of physical memory.

How large is each page? Assume memory is byte addressed.

32 bytes

Suppose that a program has the following memory allocation and page table.

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	11
N/A	011	NULL
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

What will the page table look like if the program runs the following function? Page out the least recently used page of memory if a page needs to be allocated when physical memory is full. Assume that the stack will never exceed one page of memory.

```
#define PAGE_SIZE 1024;

void helper(void) {
    char *args[5];
    int i;
    for (i = 0; i < 5; i++) {
        // Assume malloc allocates an entire page every time
        args[i] = (char*) malloc(PAGE_SIZE);
    }
```

```
    printf("%s", args[0]);
}
```

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	00
Code Segment	001	10
Heap	010	11
N/A	011	NULL
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	00
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	11
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	PAGEOUT
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	11
Heap	100	00
N/A	101	NULL
N/A	110	NULL
Stack	111	01

Memory Segment	Virtual Page Number	Physical Page Number
Heap	000	PAGEOUT
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	PAGEOUT
Heap	100	00
Heap	101	11
N/A	110	NULL
Stack	111	01

What happens when the system runs out of physical memory? What if the program tries to access an address that isn't in physical memory? Describe what happens in the user program, the operating system, and the hardware in these situations.

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in physical memory. The computer hardware traps to the kernel and current state information is saved. The system will then find out which virtual page was needed. If the virtual address is valid, the system checks for a free page. If there are no free pages in memory, a page replacement policy is applied to remove a page. The page is brought in from disk, the

faulting instruction is backed up to the state it had when it began state information is restored, and execution is resumed.

3.3 Address Translation

Consider a machine with a physical memory of 8 GB, a page size of 8 KB, and a page table entry size of 4 bytes. How many levels of page tables would be required to map a 46-bit virtual address space if every page table fits into a single page?

Since each PTE is 4 bytes and each page contains 8KB, then a one-page page table would point to 2048 or 2^{11} pages, addressing a total of $2^{11} * 2^{13} = 2^{24}$ bytes.
Depth 1 = 2^{24} bytes Depth 2 = 2^{35} bytes Depth 3 = 2^{46} bytes So in total, 3 levels of page tables are required.

List the fields of a Page Table Entry (PTE) in your scheme.

Each PTE will have a pointer to the proper page, PPN, plus several bits – read, write, execute, and valid. This information can all fit into 4 bytes, since if physical memory is 2^{33} bytes, then 20 bits will be needed to point to the proper page, leaving ample space (12 bits) for the information bits.

Without a cache or TLB, how many memory operations are required to read or write a single 32-bit word?

Without extra hardware, performing a memory operation takes 4 actual memory operations: 3 page table lookups in addition to the actual memory operation.

With a TLB, how many memory operations can this be reduced to? Best-case scenario? Worst-case scenario?

Best-case scenario: 2 memory lookups. once in TLB, once for actual memory operation. Worst-case scenario: 5 memory lookups. once in TLB + 3 page table lookups in addition to the actual memory operation.

The pagemap is moved to main memory and accessed via a TLB. Each main memory access takes 50 ns and each TLB access takes 10 ns. Each virtual memory access involves:
- mapping VPN to PPN using TLB (10 ns)
- if TLB miss: mapping VPN to PPN using page map in main memory (50 ns)
- accessing main memory at appropriate physical address (50 ns)

Assuming no page faults (i.e. all virtual memory is resident) what TLB hit rate is required for an average virtual memory access time of 61ns?

```
(10+50)*x+(1-x)*(50+10+50) = 61

solve for x gives x = .98 = 98% hit rate
```

Assuming a TLB hit rate of .50, how does the average virtual memory access time of this scenario compare to no TLB?

```
With a TLB with a hit rate of 0.5:
x = 0.5
avg_time = (10+50)*x+(1-x)*(50+10+50)
avg_time = 85
```

```
Without a TLB:
time = 50 + 50
time = 100
```

3.4 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
```