Section 6: Deadlock and Address Translation

CS 162

March 6, 2020

Contents

1 Vocabulary

- **Deadlock** - A case of starvation due to a cycle of waiting. Computer programs sharing the same resource effectively prevent each other from accessing the resource, causing both programs to cease to make progress.
- **Banker's Algorithm** - A resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, before deciding whether allocation should be allowed to continue.
- **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 Deadlock

2.1 Introduction

What are the four requirements for Deadlock?

Mutual Exclusion, Hold & Wait, No Preemption, and Circular Wait.

What is starvation and what is deadlock? How are they different?

Starvation occurs when a thread waits indefinitely. An example of starvation is when a low-priority thread waiting for a resource that is constantly in use by high-priority threads.
Deadlock is the circular waiting of resources. Deadlock implies starvation but not vice-versa.

2.2 Banker's Algorithm

Suppose we have the following resources: A, B, C and threads T1, T2, T3 and T4. The total number of each resource as well as the current/max allocations for each thread are as follows:

Total		
A	B	C
7	8	9

	Current			Max		
T/R	A	B	C	A	B	C
T1	0	2	2	4	3	3
T2	2	2	1	3	6	9
T3	3	0	4	3	1	5
T4	1	3	1	3	3	4

Is the system in a safe state? If so, show a non-blocking sequence of thread executions.

Yes, the system is in a safe state.
To find a safe sequence of executions, we need to first calculate the available resources and the needed resources for each thread. To find the available resources, we sum up the currently held resources from each thread and subtract that from the total resources:

Available		
A	B	C
1	1	1

To find the needed resources for each thread, we subtract the resources they currently have from the maximum they need:

Needed			
	A	B	C
T1	4	1	1
T2	1	4	8
T3	0	1	1
T4	2	0	3

From these, we see that we must run T3 first, as that is the only thread for which all needed resources are currently available. After T3 runs, it returns its held resources to the resource pool, so the available resource pool is now as follows:

Available		
A	B	C
4	1	5

We can now run either T1 or T4, and following the same process, we can arrive at a possible execution sequence of either $T3 \rightarrow T1 \rightarrow T4 \rightarrow T2$ or $T3 \rightarrow T4 \rightarrow T1 \rightarrow T2$.

Repeat the previous question if the total number of C instances is 8 instead of 9.

Following the same procedure from the previous question, we see that there are 0 instances of C available at the start of this execution. However, every thread needs at least 1 instance of C to run, so we are unable to run any threads and thus the system is not in a safe state.

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

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 32;
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
N/A	000	NULL
Code Segment	001	10
Heap	010	11
Heap	011	00
N/A	100	NULL
N/A	101	NULL
N/A	110	NULL
Stack	111	01

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

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

Memory Segment	Virtual Page Number	Physical Page Number
N/A	000	NULL
Code Segment	001	10
Heap	010	PAGEOUT
Heap	011	PAGEOUT
Heap	100	PAGEOUT
Heap	101	00
Heap	110	11
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: 1 memory lookup. Hit in TLB, once for actual memory operation.
Worst-case scenario: 4 memory lookups. Miss 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
```