Section 7: Banker's Algorithm and Address Translation

October 16-18

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 The Central Galactic Floopy Corporation

It's the year 3162. Floopies are the widely recognized galactic currency. Floopies are represented in digital form only, at the Central Galactic Floopy Corporation (CGFC).

You receive some inside intel from the CGFC that they have a Galaxynet server running on some old OS called x86 Ubuntu 14.04 LTS. Anyone can send requests to it. Upon receiving a request, the server forks a POSIX thread to handle the request. In particular, you are told that sending a transfer request will create a thread that will run the following function immediately, for speedy service.

```
void transfer(account_t *donor, account_t *recipient, float amount) {
  assert (donor != recipient); // Thanks CS161
  if (donor->balance < amount) {
    printf("Insufficient funds.\n");
    return;
  }
  donor->balance -= amount;
  recipient->balance += amount;
}
```

Assume that there is some struct with a member `balance` that is `typedef`-ed as `account_t`. Describe how a malicious user might exploit some unintended behavior.

There are multiple race conditions here.

Suppose Alice and Bob have 5 floopies each. We send two quick requests: transfer(&alice, &bob, 5) and transfer(&bob, &alice, 5). The first call decrements Alice's balance to 0, adds 5 to Bob's balance, but before storing 10 in Bob's balance, the next call comes in and executes to completion, decrementing Bob's balance to 0 and making Alice's balance 5. Finally we return to the first call, which just has to store 10 into Bob's balance. In the end, Alice has 5, but Bob now has 10. We have effectively duplicated 5 floopies.

Graphically:

Thread 1	**Thread 2**
temp1 = Alice's balance (== 5)	
temp1 = temp1 - 5 (== 0)	
Alice's balance = temp1 (== 0)	
temp1 = Bob's balance (== 5)	
temp1 = temp1 + 5 (== 10)	
INTERRUPTED BY THREAD 2	
	temp2 = Bob's balance (== 5)
	temp2 = temp2 - 5 (== 0)
	Bob's balance = temp2 (== 0)
	temp2 = Alice's balance (== 0)
	temp2 = temp2 + 5 (== 5)
	Alice's balance = temp2 (== 5)
	THREAD 2 COMPLETE
RESUME THREAD 1	
Bob's balance = temp1 (== 10)	
THREAD 1 COMPLETE	

It is also possible to achieve a negative balance. Suppose at the beginning of the function, the donor has enough money to participate in the transfer, so we pass the conditional check for sufficient funds. Immediately after that, the donor's balance is reduced below the required amount by some

other running thread. Then the transfer will go through, resulting in a negative balance for the donor.

Sending two identical transfer(&alice, &bob, 2) may also cause unintended behavior, since the increment/decrement operations are not atomic (though it is arguably harder to exploit for profit).

Since you're a good person who wouldn't steal floopies from a galactic corporation, what changes would you suggest to the CGFC to defend against this exploit?

The entire function must be made atomic. One could do this by disabling interrupts for that period of time (if there is a single processor), or by acquiring a lock beforehand and releasing the lock afterwards. Alternatively, you could have a lock for each account. In order to prevent deadlocks, you will have to acquire locks in some predetermined order, such as lowerst account number first.

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 dirty, resident, and protection bits) 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. The page size is independent of what we calculated above, and 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 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. Check that the virtual address that caused the page fault is valid and check that there is no pro
2. Using the replacement algorithm, iterate through the disk table and get
the number of a frame that will be used for the incoming page
3. Swap the page currently in that frame to its slot on the disk
4. Swap the requested page from its slot on disk into the above frame
5. 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
```