Section 7: Scheduling, Deadlock, Caches

July 15, 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.
- **Priority Inversion** - If a higher priority thread is blocking on a resource (a lock, as far as you're concerned but it could be the Disk or other I/O device in practice) that a lower priority thread holds exclusive access to, the priorities are said to be inverted. The higher priority thread cannot continue until the lower priority thread releases the resource. This can be amended by implementing priority donation.
- **Priority Donation** - If a thread attempts to acquire a resource (lock) that is currently being held, it donates its effective priority to the holder of that resource. This must be done recursively until a thread holding no locks is found, even if the current thread has a lower priority than the current resource holder. (Think about what would happen if you didn't do this and a third thread with higher priority than either of the two current ones donates to the original donor.) Each thread's effective priority becomes the max of all donated priorities and its original priority.

- **Cache** - A repository for copies that can be accessed more quickly than the original. Caches are good when the frequent case is frequent *enough* and the infrequent case is not too expensive. Caching ensures locality in two ways: temporal (time), keeping recently accessed data items 'saved', and spatial (space), since we often bring in contiguous blocks of data to the cache upon a cache miss.
- **AMAT** - Average Memory Access Time: a key measure for cache performance. The formula is (Hit Rate x Hit Time) + (Miss Rate x Miss Time) where Hit Rate + Miss Rate = 1.
- **Compulsory Miss** - The miss that occurs on the first reference to a block. This also happens with a 'cold' cache or a process migration. There's essentially nothing that you can do about this type of miss, but over the course of time, compulsory misses become insignificant compared to all the other memory accesses that occur.
- **Capacity Miss** - The miss that occurs when the cache can't contain all the blocks that the program accesses. One solution for capacity misses is increasing the cache size.
- **Conflict Miss** - The miss that occurs when multiple memory locations are mapped to the same cache location (i.e a collision occurs). In order to prevent conflict misses, you should either increase the cache size or increase the associativity of the cache. These technically do not exist in virtual memory, since we use fully-associative caches.
- **Coherence Miss** - Coherence misses are caused by external processors or I/O devices that update what's in memory (i.e invalidates the previously cached data).
- **Tag** - Bits used to identify the block - should match the block's address. If no candidates match, cache reports a cache miss.
- **Index** - Bits used to find where to look up candidates in the cache. We must make sure the tag matches before reporting a cache hit.
- **Offset** - Bits used for data selection (the byte offset in the block). These are generally the lowest-order bits.
- **Direct Mapped Cache** - For a 2^N byte cache, the uppermost (32 - N) bits are the cache tag; the lowest M bits are the byte select (offset) bits where the block size is 2^M. In a direct mapped cache, there is only one entry in the cache that could possibly have a matching block.
- **N-way Set Associative Cache** - N directly mapped caches operate in parallel. The index is used to select a set from the cache, then N tags are checked against the input tag in parallel. Essentially, within each set there are N candidate cache blocks to be checked. The number of sets is X / N where X is the number of blocks held in the cache.
- **Fully Associative Cache** - N-way set associative cache, where N is the number of blocks held in the cache. Any entry can hold any block. An index is no longer needed. We compare cache tags from all cache entries against the input tag in parallel.

2 Scheduling

2.1 All Threads Must Die

You have three threads with the associated priorities shown below. They each run the functions with their respective names. Assume upon execution all threads are initially unblocked and begin at the top of their code blocks. The operating system runs with a preemptive priority scheduler. You may assume that set_priority commands are atomic.

Tyrion : 4
Ned: 5
Gandalf: 11

Note: The following uses references to Pintos locks and data structures.

```
struct list braceYourself;    // pintos list. Assume it's already initialized and populated.
struct lock midTerm;          // pintos lock. Already initialized.
struct lock isComing;

void tyrion(){
    thread_set_priority(12);
    lock_acquire(&midTerm);
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
    lock_acquire(&midTerm);
    lock_acquire(&isComing);
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
    lock_acquire(&isComing);
    thread_set_priority(3);
    while (thread_get_priority() < 11) {
        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
        timer_sleep(20);
    }
    lock_release(&isComing);
    thread_exit();
}
```

What is the output of this program when there is no priority donation? Trace the program execution and number the lines in the order in which they are executed.

```
void tyrion(){
5    thread_set_priority(12);
6    lock_acquire(&midTerm); //blocks
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
3    lock_acquire(&midTerm);
4    lock_acquire(&isComing);  //blocks
    list_remove(list_head(braceYourself));
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}

void gandalf(){
1    lock_acquire(&isComing);
2    thread_set_priority(3);
7    while (thread_get_priority() < 11) {
8        printf("YOU .. SHALL NOT .. PAAASS!!!!!!); //repeat till infinity
9        timer_sleep(20);
    }
    lock_release(&isComing);
    thread_exit();
}

Gandalf, as you might expect, endlessly prints "YOU SHALL NOT PASS!!" every 20 clock ticks or so.
```

What is the output and order of line execution if priority donation was implemented? Draw a diagram of the three threads and two locks that shows how you would use data structures and struct members (variables and pointers, etc) to implement priority donation for this example.

```
void tyrion(){
8    thread_set_priority(12);
9    lock_acquire(&midTerm); //blocks
    lock_release(&midTerm);
    thread_exit();

}

void ned(){
3    lock_acquire(&midTerm);
4    lock_acquire(&isComing);  //blocks
12   list_remove(list_head(braceYourself));  //KERNEL PANIC
    lock_release(&midTerm);
    lock_release(&isComing);
    thread_exit();
}
```

```
void gandalf(){
1    lock_acquire(&isComing);
2    thread_set_priority(3);
5    while (thread_get_priority() < 11) {   //priority is 5 first, but 12 at some later loop
6        printf("YOU .. SHALL NOT .. PAAASS!!!!!!);
7        timer_sleep(20);
    }
10    lock_release(&isComing);
11    thread_exit();
}

It turns out that Gandalf generally does mean well. Donations will make
Gandalf allow you to pass.
At some point Gandalf will sleep on a timer and leave Tyrion alone in the
ready queue.
Tyrion will run even though he has a lower priority (Gandalf has a 5
donated to him)
Tyrion then sets his priority to 12 and chain-donates to Gandalf. Gandalf
breaks his loop.
Ned unblocks after Gandalf exits.
However, allowing Ned to remove the head of a list will trigger an ASSERT
failure in lib/kernel/list.c.

Gandalf will print YOU SHALL NOT PASS at least once.
Then Ned will get beheaded and cause a kernel panic that crashes Pintos.
```

2.2 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.

3 Deadlock

3.1 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.

3.2 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.

4 Caching

4.1 Direct Mapped vs. Fully Associative Cache

An big data startup has just hired you to 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.

First, you create 1) a direct mapped cache and 2) a fully associative cache of the same size that uses an LRU replacement policy. You run a few tests and realize that the fully associative cache performs much worse than the direct mapped cache does. What's a possible access pattern that could cause this to happen?

Let's say each cache held X amount of blocks. An access pattern would be to repeatedly iterate over $X + 1$ consecutive blocks, which would cause everything in the fully associated cache to miss every time.

4.2 Two-way Set Associative Cache

Instead, your boss tells you to build a 8KB 2-way set associative cache with 64 byte cache blocks.

How would you split a given virtual address into its tag, index, and offset numbers?

The number of offset bits is determined by the size of the cache blocks. Thus, the offset will take 6 bits, since $2^6 = 64$.

Recall that for a set associative cache, each 'set' holds N 'candidiate' blocks. Thus, to find the index we must find how many sets there are. We divide by N first to get total bytes per bank, then find how many blocks fit in each bank to get the number of blocks. Since it's two way set associative, the cache is split into two 4KB banks. Each bank can store 64 blocks, since total bytes per bank / block size = $2^{12}/2^6 = 2^6$, so there will be 6 index bits. This matches what we expect, which is that the whole cache can hold 128 blocks.

The remaining bits will be used as the tag (32-6-6 = 20).

It will look like this:

20 Bits	6 Bits	6 Bits
Tag	Index	Offset

4.3 Average Read Time

You finish building the cache, and you want to show your boss that there was a significant improvement in average read time.

Suppose your system uses a two level page table to translate virtual addresses, and your system uses the cache for the translation tables and data. Each memory access takes 50ns, the cache lookup time is 5ns, and your cache hit rate is 90%. What is the average time to read a location from memory?

Recall that page tables are held in memory as well. Since the page table has two levels, there are three reads for each access: read from first-level page table, read from second-level page table, and finally read from the physical page. Because the system also uses the cache for the translation tables, accessing a page table costs the same as going to memory. Thus, to get our final answer we should calculate the average access time, then multiply that by 3 to get the average read time.

The average access time is: $0.9 * 5 + 0.1 * (5 + 50) = 10$ns. The miss time includes the cache lookup time, as well as the time for a memory access. Since there are three accesses, we multiply this by 3 to get an average read time of 30ns.

4.4 Average Read Time with TLB

In addition to the cache, you add a TLB to aid you in memory accesses, with an access time of 10ns. Assuming the TLB hit rate is 95%, what is the average read time for a memory operation? You should use the answer from the previous question for your calculations.

If the TLB hits, we only need to read one page - the physical page mapped to in the TLB (we don't consider TLB accesses as physical memory accesses), with an access time of 10ns for a single read. Otherwise, we need to read the page table again; as in the previous part, the average read time for three accesses is 30ns. Thus, the average read time is $0.95 * (10 + 10) + 0.05 * (10 + 30) = 21$ns