CMPSCI 630 Concurrency **Fall 2014**

Lecture 7

Lecturer: Emery Berger *Scribe: Adam Nelson, Seema Guggari*

7.1 Review of Experiences with Processes and Monitors in Mesa

Half the class thought this week's first paper was great, and half the class hated it.

Why was it significant? People didn't really know what to do with abstractions like threads. The way that things evolved weren't necessarily the best way to structure them. Mesa tried to iron out the semantics of the synchronization primitives.

One of the biggest challenges in PL is interaction of features. For example: if you have objects, then synchronization, then exceptions... how do they interact? This is why designing good languages is hard, and what motivates the use of formal methods. When people design languages in academia, they design these languages using formal methods to prove behavior. Without this, undefined behavior is common. Behavior is often "defined by the compiler," which used to be OK, but can lead to bugs becoming part of the standard.

Mesa is a concurrent based programming language. It was introduced to overcome the shortcomings of monitors. There were several design constraints of monitors. For example, WAIT was not properly designed in a nested monitor call, there was no proper synchronization between monitors and also there was no mechanism to handle exceptions.

7.2 Evolution of Threads

When a process is created, it has its own address space, PC, registers and stack. Each of these processes are added to the Wait Queue, and the scheduler executes them one at a time.

Process switching is accomplished with a queue; the OS takes a process off the queue, runs it for a bit, then puts it back on the queue. But, in case of multiple processes, the scheduler would have many processes waiting on it and thus the CPUs performance is degraded. Hence a time scheduling approach was proposed, also known as preemptive scheduling.

In preemptive scheduling, the process continues executing until a time scheduler interrupts it since the process quantum expires, stores the contents of the PC, stack, registers and address space into a Process Control Block, then retrieves the next process from the Wait Queue and executes it for the same amount of time.

But, we run into one problem: when is a process "done"?

- Time slicing
- Blocking
- On exit

For example, if a process tries to read from a tape, this will take a long time, so the OS should put it back on the queue and let something else run. This happens if the call is *synchronous*, like `read()`.

`write()` is usually asynchronous, although `fsync()` is *supposed* to make it synchronous.

But what if a thread just runs, without blocking? In an infinite loop? Other processes could be waiting (keyboard requests, browser)... so we cut it off after a set amount of time (time slicing, a.k.a. preemptive scheduling). Processes are stopped after a period of time known as a *quantum*, or on a blocking call, whichever comes first.

A hardware timer sends an interrupt signal to the scheduler, which stores the process's state in the queue, and then restores the next process's state from the queue.

```
     quantum expiration
            |
process ----|----> save state ---> load state from next proc on queue ---> run it
```

In order to avoid context switching, some systems are designed to use non-preemptive scheduling, also known as *cooperative multitasking*. This is not used anymore. In cooperative multitasking,the threads tell the scheduler when they can pause (*yield*). Yielding does not mean a program is done, just that it can take a break. The process interrupts itself instead of waiting for quantum expiration, and it gives control to other process. Cooperative scheduling hogs the CPU completely for a single task.

Co-op is the optimist's view, preemptive is the pessimist's view.

```
for (i = 0; i < 1e9; i++) {
  // something time-consuming...
}
yield()
```

Figure 7.1: A program that could completely hang a co-op multitasking system.

Co-op multitasking is nice, but impractical. Consider the *threat model*: it's easy for a malicious program to never yield, and take up the whole CPU. A careless programmer might also do this. An infinite loop can take down the machine.

UNIX machines use preemptive scheduling with optional co-op scheduling. The "hint" `sched_yield()` is like a yield.

Co-op scheduling was originally used due to lack of hardware interrupts. Computers didn't have timers!

Preemptive scheduling and cooperative scheduling differ in a way that makes co-op scheduling look really nice. Cooperative scheduling is more general than preemptive scheduling

Let's think about it in terms of the possible schedules:

```
a = 1;
b = 2;
c = 3;
sched_yield();
```

How many ways are there of putting in breaks in this program?
In cooperative: only one, the `sched_yield`.
In preemptive: anywhere.

Now, let's say we have threads. The difference between processes and threads: threads can share an address space. (Threads are "lightweight processes." Switching is faster when the address space is the same.)

Cooperative Scheduling:

```
t1              t2
-------------   --------------
a = 1;          a = 2;
b = 2;          b = 4;
c = 3;          c = 6;
sched_yield();  sched_yield();
print(a,b,c);
```

Output:
If thread 2 executes first: `1 2 3`
If thread 1 executes first: `2 4 6`

Preemptive Scheduling:

```
t1              t2
-------------   --------------
a = 1;          a = 2;
b = 2;          b = 4;
c = 3;          c = 6;
print(a,b,c);
```

Output:
Uncertain. There can be many interleavings and hence mutiple combinataions. As the mumber of threads increase, there are more possiblities.

Figure 7.2: Execution order in cooperative scheduling vs. preemptive scheduling.

What a process stores when unscheduled: PC, registers, stack, file descriptors, and the mapping from virtual to physical addresses (page table). This is why threads are lighter-weight than processes: the page table remains the same, file descriptors remain the same, and the hardware cache (TLB) of the page table isn't invalidated. Keeping the cache is the most important part, performance-wise.

7.3 Translation Lookaside Buffer and Cache Associativity

The TLB is a cache which stores the page table entries that map virtual addresses to physical addresses. It is a *fully associative* cache, meaning that it follows a strict LRU policy: when out of space, the least-recently-used item in the cache is the first one to be evicted. This is great, but it's expensive: it requires a huge hardware queue of the cache entries.

Hardware L1, L2, etc. caches are not as nice. They do something that's kind of a hack. Suppose I go look up a page: instead of doing LRU, all the cache does is hash the page entry to some bucket. If there's nothing

there, it's stored it there; if there's something, it's evicted. This is basically a hash table.

Figure 7.3: A 1-way set associative cache, with hash function $(pg \div 4096)\%8$, and two entries. Trying to add an entry like 4096^2 will evict 4096.

This flat hash table is called *direct mapping*. It's a really simple strategy, but there's no LRU-ness at all. Things could keep getting evicted, back-and-forth. This is ugly, and terrible, and fast.

LRU is the best, direct mapping is the worst, and there's a spectrum in between. For example, we can extend the direct mapping by giving two slots to each bucket, and maintaining them in LRU order. (2-way LRU) This is called 2-way set-associative. Each bucket is called a set. The more sets, the more expensive it is.

8-way set-associative caches are considered the "sweet spot." This has been determined (somewhat) empirically.

TLBs are almost uniformly fully-associative, because speed is *just that important*. A TLB lookup is on the critical path of every single memory access, because userland programs only know about virtual memory. Frequent TLB misses can cause performance to plummet.

When a process starts, or after a context switch, all of the caches are *cold*, and virtual-to-physical address mappings can't be easily found in the TLB. Hence initially things are slow, then it gets faster. This is known as *warm-up*.

For every single access, the program checks the page table for its physical address map. If requested entry is not found in the page table, it is a TLB miss and the OS walks through every address in RAM to compute the physical address. This involves more cycles when compared to accessing address from TLB. If we frequently encounter TLB misses, it leads to TLB *thrashing*, which degrades performance.

7.4 Scheduling and Priority

Round-robin scheduling: everyone gets a turn in a fixed order, no prioritization. A FIFO queue is round-robin. Round-robin scheduling is nice, everyone understands it... and no one uses it.

For security reasons, the TLB must be wiped when processes are switched; we don't want Process 1 to have any way to access Process 2's memory. There's a hardware command that does this: it's called "TLB shootdown." So, every time you switch processes, it's not just storing the PC and stack; it's throwing away megabytes of cache. Because of this, for example, in Linux, the processor timeslice is 5ms—doesn't sound like much, but, in computer terms, it's REALLY long. Now, if round-robin scheduling were used, we might wait a full 5-10 seconds for each program to come back around... and that would be terrible. No one does this.

With threads, when we switch to another thread, we don't shoot down the TLB. This is why they're

lightweight.

Instead of round-robin scheduling, we use several different wait queues, in order to implment a priority system. Priority is a class system for threads: if there's *anything* at the highest level, schedule it before *anything* at the lower levels.

Priorities are user-defined, or else the scheduler assigns them. Some threads even inherit these priorities from their parent thread or process. However, if, say, a thread waiting on I/O has highest priority, it wont utilize the CPU fully. Hence the scheduler needs to prioritize the threads well. If a certain thread is using its entire quantum well, the scheduler assigns it higher priority. If CPU utilization is low, priority assigned would be lower. The scheduler process has the highest priority, and user process have the lowest priority.

7.5 Shared Memory vs. Distributed Memory

One reason to use threads is if you actually need them to share memory: for example, two threads working on the same task. This is usually done with fork-join. Forking processes would require *message passing*, which is slow. The whole issue of communicating between processes is called IPC (*inter-process communication*). This is usually done like communication between two computers (*slowly*), although there are tricky, sneaky ways to do it faster, which we won't talk about here.

Threads have *shared memory*, while processes have *distributed memory*. Shared memory has a lot of superficial attractiveness, especially because a lot of algorithms can be made parallel without much change to the code; however, it is also dangerous. Of course, distributed memory with message passing can encounter problems, too, such as *deadlock* (two processes mutually waiting on each other) and *livelock* (two processes switching in an infinite loop without doing any work).

Even on one processor, concurrency is still important, because of *latency hiding*: while waiting for I/O, the computer can do something else. So, a browser, even on a single-CPU machine, has a thread pool, and the threads all try to load from the network simultaneously.

Concurrency and parallelism are not the same thing!

- Concurrency: Things that happen "at the same time" (even if they don't). Latency hiding.
- Parallelism: Things that truly run at the same time, on multiple processors, to increase performance.

7.6 Locks

Locks help make shared memory usable, but they're kind of a blunt instrument, and they don't play well with priority. For example:

```
t1 (high priority)    t2 (low priority)
------------------    ------------------
                      lock(a)
                      .
lock(a)               .
.                     .
.                     .
```

Even if `t1` has higher priority, it is blocked, since `t2` owns the lock first. This is known as *priority inversion*, where lower-priority threads block the higher-priority threads.

Edsger Dijkstra came up with the first provably-correct synchronization algorithm: semaphores. (If you know any Romance languages, *semaphore* basically means a traffic light.) Dijkstra's version is known as a *counting semaphore*; a degenerate version with only two values is called a *binary semaphore*.

Semaphores are special integer variables, which support only two operations (named in Dutch): *P* and *V*.

P = *Prolagen* = "try to decrease" (Dijkstra made this word up)
V = *Verlieren* = "increase"

As an example, let's say a semaphore is initialized with value 1:

```
seminit(1); // 1
P();        // 0
V();        // 1
```

If someone else tries to issue a P and the semaphore's value is already 0, they block, and wake up again at some point to check if it's not zero anymore.

Why use counting semaphores? Limiting number of users who can access something, for performance? We usually use binary semaphores, which are basically locks. However, the problem with locks is that they don't compose:

```
lock(a);
.
.
lock(a); // deadlock!
```

This can be solved with recursive locks: if you already "hold" a lock, locking it again causes it to "nest" and require another unlock.

Java's `synchronized` keyword adds a recursive lock to a method. Once you do this, you're almost at monitors: Java has `wait()` and `notify()`, which gives you condition variables... but, since `synchronized` is not the default, it's not exactly monitors. This was done for performance reasons... which now no longer matter, because Java is smart enough to skip locks that aren't used.

```
Class A
{
       foo()
       {
              lock (x);
              // ...
              unlock(x);
       }

        bar()
       {
               lock(x);
               // ...
               foo();
               // ...
               unlock(x);
       }
}
```

Figure 7.4: An example of a situation where recursive locks prevent a deadlock. Both `foo()` and `bar()` are protected by the lock `x`, but, in order to call `foo()` from `bar()` safely, `x` must be a recursive lock.