Concurrency & Synchronization

Nima Honarmand
Agenda

• Review basic concurrency concepts
  • Concurrency and parallelism
  • Data race and mutual exclusion
  • Locks

• New stuff
  • New concurrency issues: condition variables
  • How to implement locks and condition variables efficiently
    • Focusing on OS issues
  • A deeper understanding of concurrency bugs
Concurrency Review
Concurrency and Parallelism

- Two tasks (threads, functions, instructions, etc.) are concurrent if their executions overlap in time.

- Two tasks are parallel if they execute at the same time.
  - A special case of concurrency.
  - Parallel tasks have to execute on different processors.

Run 3 threads on 1 processor
Run 3 threads on 2 processors
Sources of Concurrency

Question: How could one task run before the current one completes?

1) Tasks running on different processors

2) Context switching between tasks on the same processor
   • Preemptive as well as cooperative

3) Interrupts
   • Kernel mode: hardware interrupts and in-kernel exceptions
   • User mode: signals
Why Concurrent Programming?

• In user-mode
  • To utilize multiple processors
    • Multi- and many-core processors are here to stay
  • To improve application responsiveness in the presence of blocking operations
    • E.g., processing a UI input in the background without freezing the application

• In kernel-mode
  • Because user-mode often requires kernel-mode concurrency
    • Each thread in a multi-threaded program has a kernel-mode component (remember the iceberg?)
  • Also, because interrupts/exceptions can create unforeseen parallelism
Challenges of Concurrency

• Crux: execution order (interleaving) of instructions of concurrent tasks in not generally under our control
  • Single CPU: We can’t control scheduler decisions
  • Multi-CPU: We can’t control when, and how fast, each processor executes its instructions

• We need to control instruction interleaving for at least two reasons
  1) Mutual exclusion
  2) Condition synchronization
Mutual Exclusion

- Some computer resources cannot be accessed by multiple threads at the same time
  - E.g., a printer can’t print two documents at once

- **Mutual exclusion** is the term to indicate that some resource can only be used by one thread at a time
  - Active thread excludes its peers

- In concurrent programs, shared data structures are often mutually exclusive
  - Two threads adding to a linked list at the same time can corrupt the list
Why Mutual Exclusion for Shared Data?

• To avoid **data races**

• Imagine two concurrent threads executing this code

• What is your expected outcome?

• What are the possible outcomes?

• Undesirable things happen when concurrent tasks access shared data simultaneously
  
  • At least one access should be a write for bad things to happen

C code:
```
balance += 1;
```

Assembly code:
```
mov 0x8049a1c, %eax
add $0x1, %eax
mov %eax, 0x8049a1c
```
Why Mutual Exclusion for Shared Data?

• As programmers, we are used to thinking sequentially
  • We break a functionality into a sequence of code lines or instructions
    → We almost always use more than one instruction/line of code to achieve our goal

• We are also used to think only about the results of the current piece of code
  • Difficult for us to think about the effect of a concurrent task monkeying around with the data we are using

• Is this human nature or just because of how we were taught programming?
  • The jury is out on this!
Why Mutual Exclusion for Shared Data?

• To recap
  1) We have to do multiple things to implement an operation
     • Either do all of it, or none of it
     • Partial execution will result in an inconsistent state
  2) We don’t want anyone to touch the data we are using when doing that
     • We need isolation from others

• So, we need *atomicity*
  • Do either all or none + in isolation
Why Mutual Exclusion for Shared Data?

• One way to (almost) achieve atomicity is...

• ...to make sure we have *exclusive* access to our shared data for the length of time our *critical* instructions run
  • Hence, the name “mutual exclusion”

• A *critical section* of code is any piece of code that touches shared data (or more generally, accesses a shared resources)
  • It’s an abstraction to help us think more clearly about structure of concurrent code

• **Locks** are our main mechanisms to achieve mutual exclusion
Example: Traverse a Linked List

• Suppose we want to find an element in a singly linked list, and move it to the head

• Visual intuition:
Example: Traverse a Linked List

• Suppose we want to find an element in a singly linked list, and move it to the head

• Visual intuition:
Example: Traverse a Linked List

```c
lprev = NULL;
for(lptr = lhead; lptr; lptr = lptr->next) {
    if(lptr->val == target){
        // Already head?, break
        if(lprev == NULL) break;
        // Move cell to head
        lprev->next = lptr->next;
        lptr->next = lhead;
        lhead = lptr;
        break;
    }
    lprev = lptr;
}
```

• Where is the critical section?
Example: Traverse a Linked List

- A critical section often needs to be larger than it first appears
  - The 3 key lines are not enough of a critical section

Thread 1

```c
// Move cell to head
lprev->next = lptr->next;
lptr->next = lhead;
lhead = lptr;
```

Thread 2

```c
// Move cell to head
lprev->next = lptr->next;
lptr->next = lhead;
lhead = lptr;
```
Example: Traverse a Linked List

Thread 1

```c
lprev = NULL;
for(lptr = lhead; lptr; lptr = lptr->next)
{
   if(lptr->val == target){
      // Already head?, break
      if(lprev == NULL) break;
      // Move cell to head
      lprev->next = lptr->next;
      lptr->next = lhead;
      lhead = lptr;
      break;
   }
   lprev = lptr; }
```

Thread 2

```c
lprev = NULL;
for(lptr = lhead; lptr; lptr = lptr->next) {
   if(lptr->val == target){
      // Already head?, break
      if(lprev == NULL) break;
      // Move cell to head
      lprev->next = lptr->next;
      lptr->next = lhead;
      lhead = lptr;
      break;
   }
   lprev = lptr; }
```

- Putting entire search in a critical section reduces concurrency, but it is safe

- Writing high-performance and correct concurrent programs is a (very) difficult task
Condition Synchronization

• Mutual exclusion is not all we need for concurrent programming

• Very often, synchronization consists of one task waiting for another to make a condition true
  • Ex1: master thread tells worker thread a request has arrived
    • Worker thread has to wait until this happen
  • Ex2: parent thread waits until a child thread terminates (pthread_join())

• Until condition becomes true, thread can sleep
  • Ties synchronization to scheduling

• We use condition variables for this purpose (next lecture)