CS 202
Advanced Operating Systems

Spring 21

Finishing Scheduling, starting Concurrency and Synchronization

Administrivia

- How is Lab going?

- Office hours
  - after class for quick items, or email me.
  - If there is a lot of interest, I can set up weekly time

- Academic honesty
  - Please follow rules
  - Please do not mispresent someone else's work as yours
  - Work hard, have fun, don't worry too much about grades

Parallel/distributed scheduling

- Parallel processing started early
  - Many hands make light work
  - I did my PhD in this area – frustrating to work in it
    - Competition with Moore's law
    - Programming is hard
  - Scheduling when the machine is shared
    - E.g., Gang scheduling

- COW/NOW projects (~early 1990s)
  - Opportunistically use resources when they are available
  - Scheduling is important subsystem
  - Heterogeneous schedulers such as Condor, Hence, …

Parallel/distributed processing

- Late 1990s:
  - Grid computing
  - Clusters
  - Public resource computing
  - Other: example, peer to peer networks focused on content sharing
- 2000s:
  - Cloud computing
  - Data centers
- Scheduling nowadays: lets listen to the Hawk talk

INTRODUCTION TO CONCURRENCY AND SYNCHRONIZATION

Concurrency and synchronization

- Threads share the same address space and resources
- Threads cooperate on concurrent activities
- We are under the mercy of the scheduler; generally, the scheduler is unaware of the application
- What can go wrong?
  - Race conditions
  - Incorrect ordering of activities
- So, we need tools to synchronize
  - They should enable us to control concurrency effectively
  - We need to perform well
  - We need to handle some resulting issues: deadlocks, lock contention, convoying, scheduler interactions

Threads: Cooperation

- Threads voluntarily give up the CPU with thread_yield

Ping Thread

```
while (1) {
    printf("ping\n");
    thread_yield();
}
```

Pong Thread

```
while (1) {
    printf("pong\n");
    thread_yield();
}
```

Synchronization

- For correctness, we need to control this cooperation
  - Threads interleave executions arbitrarily and at different rates
  - Scheduling is not under program control
- We control cooperation using synchronization
  - Synchronization enables us to restrict the possible inter-leavings of thread executions
- Problem occurs around shared resources
  - Variables, etc…

A First Example

- Suppose we have to implement a function to handle withdrawals from a bank account:

```
withdraw (account, amount) {
        balance = get_balance(account);
        balance = balance – amount;
        put_balance(account, balance);
        return balance;
}
```

- Now suppose that you and your father share a bank account with a balance of $1000
- Then you each go to separate ATM machines and simultaneously withdraw $100 from the account

Example Continued

- We'll represent the situation by creating a separate thread for each person to do the withdrawals
- These threads run on the same bank machine:

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

- What's the problem with this implementation?
  - Think about potential schedules of these two threads

Interleaved Schedules

- The problem is that the execution of the two threads can be interleaved:

Execution sequence seen by CPU

```
balance = get_balance(account);
balance = balance – amount;
```

```
balance = get_balance(account);
balance = balance – amount;
put_balance(account, balance);
```

```
put_balance(account, balance);
```

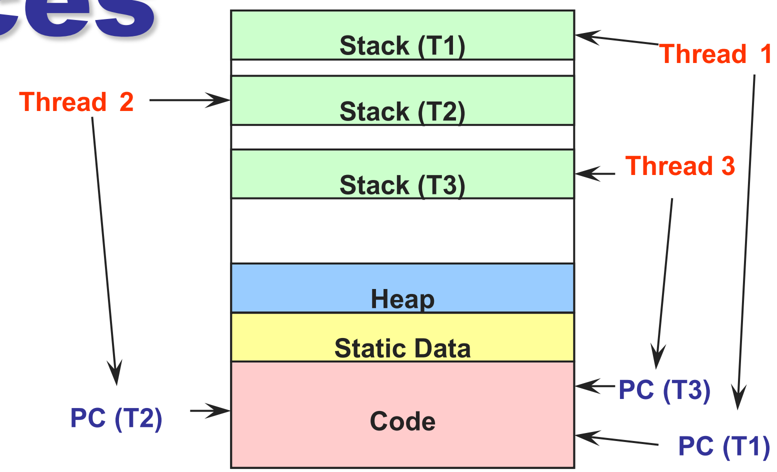
Context switch

- What is the balance of the account now?

Shared Resources

- Problem: two threads accessed a shared resource
  - Known as a race condition (remember this buzzword!)
- Need mechanisms to control this access
  - So we can reason about how the program will operate
- Our example was updating a shared bank account
- Also necessary for synchronizing access to any shared data structure
  - Buffers, queues, lists, hash tables, etc.

When Are Resources Shared?



- Local variables?
  - Not shared: refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2

- Global variables and static objects?
  - Shared: in static data segment, accessible by all threads

- Dynamic objects and other heap objects?
  - Shared: Allocated from heap with malloc/free or new/delete

How Interleaved Can It Get?

How contorted can the interleavings be?

- We'll assume that the only atomic operations are reads and writes of individual memory locations
  - Some architectures don't even give you that!
- We'll assume that a context switch can occur at any time
- We'll assume that you can delay a thread as long as you like as long as it's not delayed forever

```
............... get_balance(account);
balance = get_balance(account);
balance = .................................
balance = balance – amount;
balance = balance – amount;
put_balance(account, balance);
put_balance(account, balance);
```

What do we do about it?

- Does this problem matter in practice?
- Are there other concurrency problems?
- And, if so, how do we solve it?
  - Really difficult because behavior can be different every time
- How do we handle concurrency in real life?

Mutual Exclusion

- Mutual exclusion to synchronize access to shared resources
  - This allows us to have larger atomic blocks
  - What does atomic mean?
- Code that uses mutual called a critical section
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves a critical section, another can enter
  - Example: sharing an ATM with others
- What requirements would you place on a critical section?

Critical Section Requirements

Critical sections have the following requirements:

1) Mutual exclusion (mutex)
   - If one thread is in the critical section, then no other is
2) Progress
   - A thread in the critical section will eventually leave the critical section
   - If some thread T is not in the critical section, then T cannot prevent some other thread S from entering the critical section
3) Bounded waiting (no starvation)
   - If some thread T is waiting on the critical section, then T will eventually enter the critical section
4) Performance
   - The overhead of entering and exiting the critical section is small with respect to the work being done within it

Mechanisms For Building Critical Sections

- Locks
  - Primitive, minimal semantics, used to build others
- Semaphores
  - Basic, easy to get the hang of, but hard to program with
- Monitors
  - High-level, requires language support, operations implicit
- Architecture help
  - Atomic read/write
    - Can it be done?

How do we implement a lock? First try

```
pthread_trylock(mutex) {
    if (mutex==0) {
     mutex= 1;
     return 1;
    } else return 0;
}
```

```
Thread 0, 1, …

…//time to access critical region
while(!pthread_trylock(mutex); // wait
<critical region>
pthread_unlock(mutex)
```

- Does this work? Assume reads/writes are atomic
- The lock itself is a critical region!
  - Chicken and egg
- Computer scientist struggled with how to create software locks

Dekker's Algorithm

```
Bool flag[2]l
Int turn = 1;
```

```
flag[0] = 1;
while (flag[1] != 0) {
        if(turn == 2) {
        flag[0] = 0;
         while (turn == 2);
        flag[0] = 1;
        } //if
}//while
critical section
flag[0]=0;
turn=2;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0) {
        if(turn == 1) {
        flag[1] = 0;
         while (turn == 1);
        flag[1] = 1;
        } //if
}//while
critical section
flag[1]=0;
turn=1;
outside of critical section
```

Some observations

- This stuff (software locks) is hard
  - Hard to get right
  - Hard to prove right
- It also is inefficient
  - A spin lock – waiting by checking the condition repeatedly
- Even better, software locks don't really work
  - Compiler and hardware reorder memory references from different threads
    - Something called memory consistency model
    - Well beyond the scope of this class ☺
- So, we need to find a different way
  - Hardware help