CS 153
Design of Operating Systems

Winter 2016

Lecture 7: Synchronization

Administrivia

- Homework 1
  - Due today by the end of day
- Hopefully you have started on project 1 by now?
  - Kernel-level threads (preemptable scheduling)
  - Be prepared with questions for this weeks Lab
  - Read up on scheduling and synchronization in textbook and start early!
  - Need to read and understand a lot of code before starting to code
  - Students typically find this to be the hardest of the three projects

Aside: PintOS Threads

- Looked at kernel code “threads/thread.c”
- The code executes in kernel space (Ring 0)
- In project 1, test cases run in Ring 0 as well. They create kernel-level threads directly from the kernel.

Cooperation between Threads

- Threads cooperate in multithreaded programs
  - To share resources, access shared data structures
    - Threads accessing a memory cache in a Web server
  - To coordinate their execution
    - One thread executes relative to another

Threads: Sharing Data

```
int num_connections = 0;

web_server() {
   while (1) {
     int sock = accept();
     thread_fork(handle_request, sock);
   }
}

handle_request(int sock) {
     ++num_connections;
     Process request
     close(sock);
}
```

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
- Discussed in terms of threads, but also applies to processes

Shared Resources

We initially focus on coordinating access to shared resources

- Basic problem
  - If two concurrent threads are accessing a shared variable, and that variable is read/modified/written by those threads, then access to the variable must be controlled to avoid erroneous behavior

- Over the next couple of lectures, we will look at
  - Mechanisms to control access to shared resources
    - » Locks, mutexes, semaphores, monitors, condition variables, etc.
  - Patterns for coordinating accesses to shared resources
    - » Bounded buffer, producer-consumer, etc.

Classic 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:

```
int  balance[MAX_ACCOUNT_ID];    // shared across threads
```

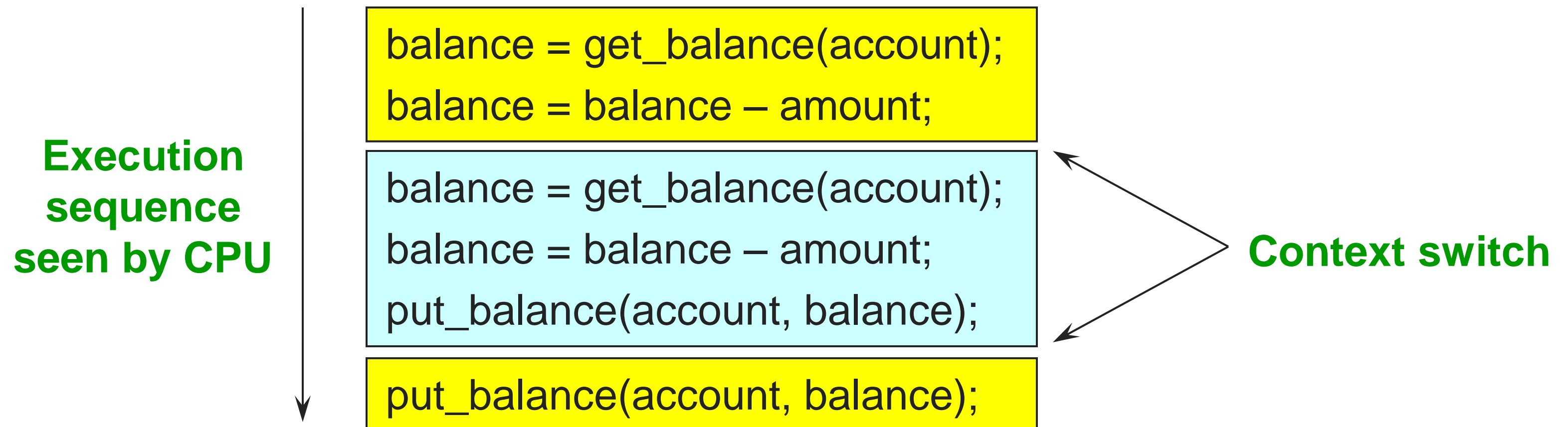
```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

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

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

Interleaved Schedules

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

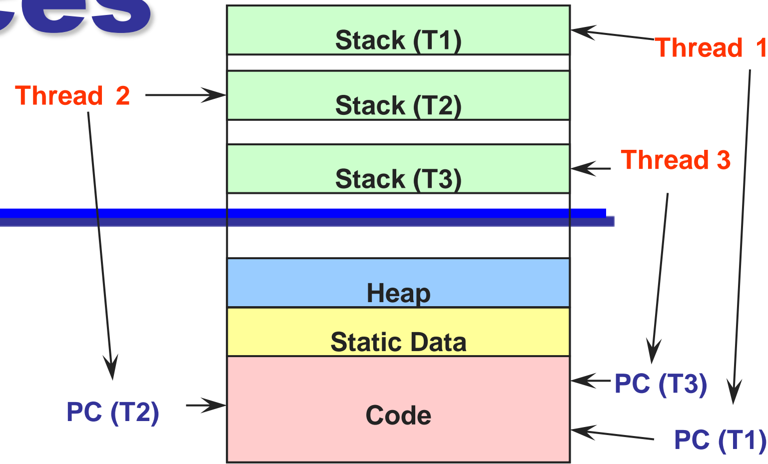


- What is the balance of the account now?

Shared Resources

- Problem: two threads accessed a shared resource
  - Known as a race condition (memorize 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);

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

About Requirements

There are three kinds of requirements that we'll use

- Safety property: nothing bad happens
  - Mutex
- Liveness property: something good happens
  - Progress, Bounded Waiting
- Performance requirement
  - Performance
- Properties hold for each run, while performance depends on all the runs
  - Rule of thumb: When designing a concurrent algorithm, worry about safety first (but don't forget liveness!).

Mechanisms For Building Critical Sections

- Atomic read/write
  - Can it be done?
- 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

Mutual Exclusion with Atomic Read/Writes: First Try

```
int turn = 1;
```

```
while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
```

```
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called alternation

It satisfies mutex:

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
- (turn == 1) ≡ (turn != 2)

Is there anything wrong with this solution?

Mutual Exclusion with Atomic Read/Writes: First Try

```
int turn = 1;
```

```
while (true) {
    while (turn != 1) ;
    critical section
    turn = 2;
    outside of critical section
}
```

```
while (true) {
    while (turn != 2) ;
    critical section
    turn = 1;
    outside of critical section
}
```

This is called alternation

It satisfies mutex:

- If blue is in the critical section, then turn == 1 and if yellow is in the critical section then turn == 2
- (turn == 1) ≡ (turn != 2)

It violates progress: blue thread could go into an infinite loop outside of the critical section, which will prevent the yellow one from entering

Mutex with Atomic R/W: Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
    try1 = true;
    turn = 2;
    while (try2 && turn != 1) ;
    critical section
    try1 = false;
    outside of critical section
}
```

```
while (true) {
    try2 = true;
    turn = 1;
    while (try1 && turn != 2) ;
    critical section
    try2 = false;
    outside of critical section
}
```

- This satisfies all the requirements
- Here's why...

Mutex with Atomic R/W: Peterson's Algorithm

```
int turn = 1;
bool try1 = false, try2 = false;
```

```
while (true) {
    {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
1   try1 = true;
    { try1 ∧ (turn == 1 ∨ turn == 2) }
2   turn = 2;
    { try1 ∧ (turn == 1 ∨ turn == 2) }
3   while (try2 && turn != 1) ;
   {  try1 ∧ (turn == 1 ∨ ¬ try2 ∨
        (try2 ∧ (yellow at 6 or at 7)) }
    critical section
4   try1 = false;
    {¬ try1 ∧ (turn == 1 ∨ turn == 2) }
    outside of critical section
}
```

```
while (true) {
    {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
5   try2 = true;
    { try2 ∧ (turn == 1 ∨ turn == 2) }
6   turn = 1;
    { try2 ∧ (turn == 1 ∨ turn == 2) }
7   while (try1 && turn != 2) ;
    {  try2 ∧ (turn == 2 ∨ ¬ try1 ∨
        (try1 ∧ (blue at 2 or at 3)) }
    critical section
8   try2 = false;
    {¬ try2 ∧ (turn == 1 ∨ turn == 2) }
    outside of critical section
}
```

(blue at 4) ∧ try1 ∧ (turn == 1 ∨ ¬ try2 ∨ (try2 ∧ (yellow at 6 or at 7))
∧ (yellow at 8) ∧ try2 ∧ (turn == 2 ∨ ¬ try1 ∨ (try1 ∧ (blue at 2 or at 3))
... ⇒ (turn == 1 ∧ turn == 2)

Peterson's Algorithm

- Hard to reason
- Simpler ways to implement Mutex exists (see later)

Locks

- A lock is an object in memory providing two operations
  - acquire(): before entering the critical section
  - release(): after leaving a critical section
- Threads pair calls to acquire() and release()
  - Between acquire()/release(), the thread holds the lock
  - acquire() does not return until any previous holder releases
  - What can happen if the calls are not paired?

Using Locks

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

Critical Section

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

```
acquire(lock);
```

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

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

- Why is the "return" outside the critical section? Is this ok?
- What happens when a third thread calls acquire?

Next time...

- Semaphores and monitors
  - Read Chapter 5.2 – 5.7

Semaphores

- Semaphores are data structures that also provide mutual exclusion to critical sections
  - Block waiters, interrupts enabled within CS
  - Described by Dijkstra in THE system in 1968
- Semaphores count the number of threads using a critical section (more later)
- Semaphores support two operations:
  - wait(semaphore): decrement, block until semaphore is open
    - Also P() after the Dutch word for test
  - signal(semaphore): increment, allow another thread to enter
    - Also V() after the Dutch word for increment

Blocking in Semaphores

- Associated with each semaphore is a queue of waiting processes
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread
    - In other words, signal() has "history"
    - This history uses a counter

Using Semaphores

- Use is similar to locks, but semantics are different

```
struct account {
    double balance;
    semaphore S;
}
withdraw (account, amount) {
    wait(account->S);
    tmp = account->balance;
    tmp = tmp - amount;
    account->balance = tmp;
    signal(account->S);
    return tmp;
}
```

```
wait(account->S);
tmp = account->balance;
tmp = tmp – amount;
```

```
wait(account->S);
```

```
wait(account->S);
```

```
account->balance = tmp;
signal(account->S);
```

```
…
signal(account->S);
```

```
…
signal(account->S);
```

Threads block

It is undefined which thread runs after a signal