CSE 153
Design of Operating Systems

Winter 23

Lecture 7/8: Synchronization (1)

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

What about processes?

- Does this apply to processes too?
  - Yes!
- Processes are a little easier because they don't share by default
- But share the OS structures and machine resources so we need to synchronize them too
  - Basically, the OS is a multi-threaded program

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
  - Exactly what problems occur
  - How to build 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.

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:

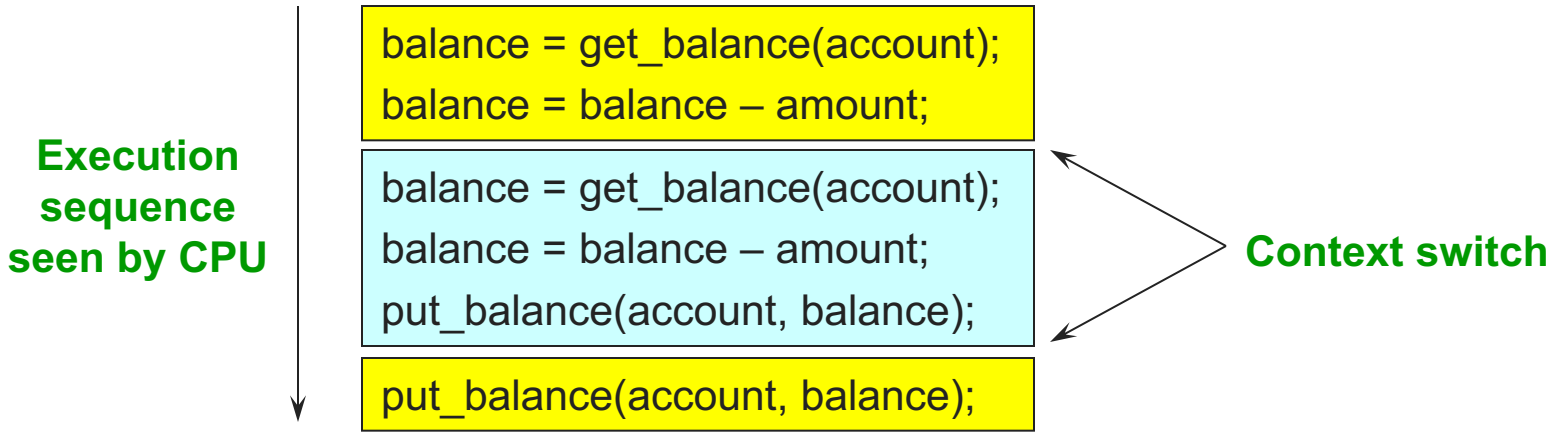
```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
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:

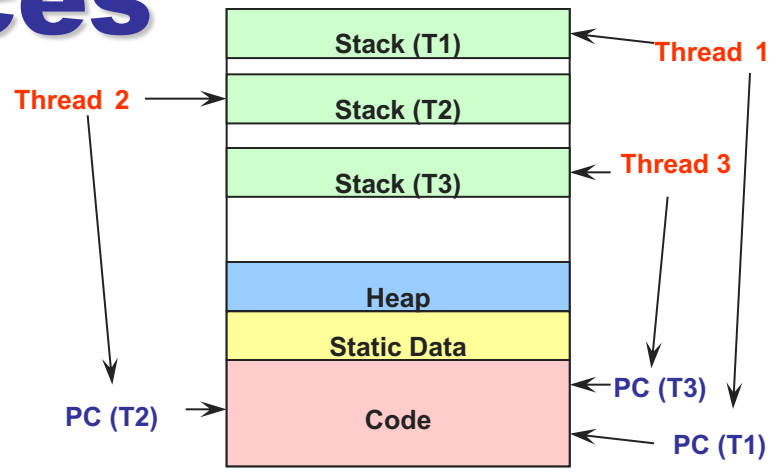


- What is the balance of the account now?

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?

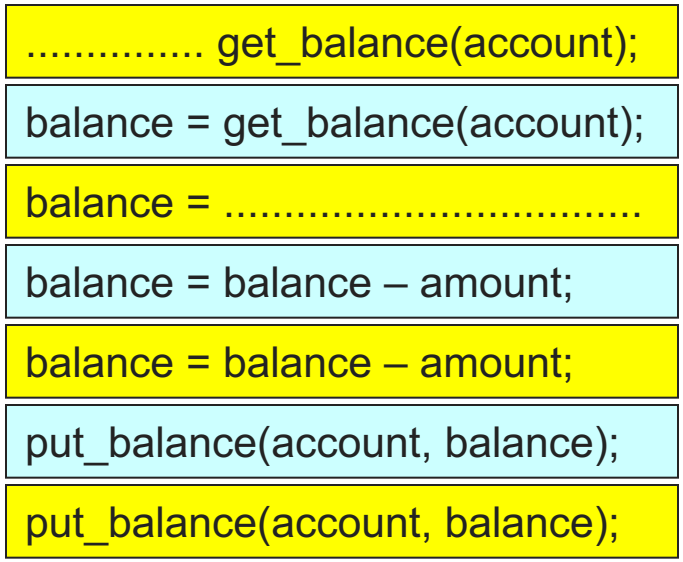


- Local variables?
  - Not shared: refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2
- Global variables and static objects?
  - Shared: in static data segment, accessible by all threads
- 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



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

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

- 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

Second 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?

Third try – two variables

Bool flag[2]

```
while (flag[1] != 0);
flag[0] = 1;
critical section
flag[0]=0;
outside of critical section
```

```
while (flag[0] != 0);
flag[1] = 1;
critical section
flag[1]=0;
outside of critical section
```

We added two variables to try to break the race for the same variable

Is there anything wrong with this solution?

Fourth try – set before you check

Bool flag[2]

```
flag[0] = 1;
while (flag[1] != 0);
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0);
critical section
flag[1]=0;
outside of critical section
```

Is there anything wrong with this solution?

Fifth try – double check and back off

Bool flag[2]

```
flag[0] = 1;
while (flag[1] != 0) {
        flag[0] = 0;
        wait a short time;
        flag[0] = 1;
}
critical section
flag[0]=0;
outside of critical section
```

```
flag[1] = 1;
while (flag[0] != 0) {
        flag[1] = 0;
        wait a short time;
        flag[1] = 1;
}
critical section
flag[1]=0;
outside of critical section
```

Six try – 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
```

Another solution: 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)

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; more in a second