# CSE 153
# Design of Operating Systems

## Fall 2018

Lecture 5: Threads/Synchronization

# Implementing threads

- Kernel Level Threads
  - All thread operations are implemented in the kernel
  - The OS schedules all of the threads in the system
  - Don’t have to separate from processes
- OS-managed threads are called kernel-level threads or lightweight processes
  - Windows: threads
  - Solaris: lightweight processes (LWP)
  - POSIX Threads (pthreads): PTHREAD_SCOPE_SYSTEM

# Kernel Thread (KLT) Limitations

- KLTs make concurrency cheaper than processes
  - Much less state to allocate and initialize
- However, there are a couple of issues
  - Issue 1: KLT overhead still high
    - Thread operations still require system calls
    - Ideally, want thread operations to be as fast as a procedure call
  - Issue 2: KLTs are general; unaware of application needs
- Alternative: User-level threads (ULT)

# Alternative: User-Level Threads

- Implement threads using user-level library

- ULTs are small and fast
  - A thread is simply represented by a PC, registers, stack, and small thread control block (TCB)
  - Creating a new thread, switching between threads, and synchronizing threads are done via procedure call
    - No kernel involvement
  - User-level thread operations 100x faster than kernel threads
  - pthreads: PTHREAD_SCOPE_PROCESS

# ULT Limitations

- But, user-level threads are not a perfect solution
  - As with everything else, they are a tradeoff
- ULTs are invisible to the OS

- As a result, the OS can make poor decisions
  - Scheduling a process with idle threads
  - Blocking a process whose thread initiated an I/O, even though the process has other threads that can execute
  - Unscheduling a process with a thread holding a lock

- Solving this requires communication between the kernel and the user-level thread manager

# Summary KLT vs. ULT

- Kernel-level threads
  - Integrated with OS (informed scheduling)
  - Slow to create, manipulate, synchronize
- User-level threads
  - Fast to create, manipulate, synchronize
  - Not integrated with OS (uninformed scheduling)
- Understanding the differences between kernel and user-level threads is important
  - For programming (correctness, performance)
  - For test-taking ☺

# Sample Thread Interface

- thread_fork(procedure_t)
  - Create a new thread of control
  - Also thread_create(), thread_setstate()
- thread_stop()
  - Stop the calling thread; also thread_block
- thread_start(thread_t)
  - Start the given thread
- thread_yield()
  - Voluntarily give up the processor
- thread_exit()
  - Terminate the calling thread; also thread_destroy

# Thread Scheduling

- The thread scheduler determines when a thread runs
- It uses queues to keep track of what threads are doing
  - Just like the OS and processes
  - But it is implemented at user-level in a library
- Run queue: Threads currently running (usually one)
- Ready queue: Threads ready to run
- Are there wait queues?
  - How would you implement thread_sleep(time)?

# Non-Preemptive Scheduling

- 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();
}
```

- What is the output of running these two threads?

# thread_yield()

- The semantics of thread_yield are that it gives up the CPU to another thread
  - In other words, it context switches to another thread
- So what does it mean for thread_yield to return?
- Execution trace of ping/pong
  - printf("ping\n");
  - thread_yield();
  - printf("pong\n");
  - thread_yield();
  - …

# Implementing thread_yield()

```
thread_yield() {
   thread_t old_thread = current_thread;
   current_thread = get_next_thread();
   append_to_queue(ready_queue, old_thread);
   context_switch(old_thread, current_thread);
   return;
}
```

As old thread

As new thread

- The magic step is invoking context_switch()
- Why do we need to call append_to_queue()?

# Thread Context Switch

- The context switch routine does all of the magic
  - Saves context of the currently running thread (old_thread)
    - Push all machine state onto its stack (*not* its TCB)
  - Restores context of the next thread
    - Pop all machine state from the next thread’s stack
  - The next thread becomes the current thread
  - Return to caller as new thread
- This is all done in assembly language
  - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls

# Preemptive Scheduling

- Non-preemptive threads have to voluntarily give up CPU
  - A long-running thread will take over the machine
  - Only voluntary calls to thread_yield(), thread_stop(), or thread_exit() causes a context switch

- Preemptive scheduling causes an involuntary context switch
  - Need to regain control of processor asynchronously
  - Use timer interrupt (How do you do this?)
  - Timer interrupt handler forces current thread to “call” thread_yield

# Threads Summary

- Processes are too heavyweight for multiprocessing
  - Time and space overhead
- Solution is to separate threads from processes
  - Kernel-level threads much better, but still significant overhead
  - User-level threads even better, but not well integrated with OS
- Scheduling of threads can be either preemptive or non-preemptive

- Now, how do we get our threads to correctly cooperate with each other?
  - Synchronization…

# Cooperation between Threads

- What is the purpose of threads?

- Threads cooperate in multithreaded programs

- Why?
  - 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 interleavings 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:

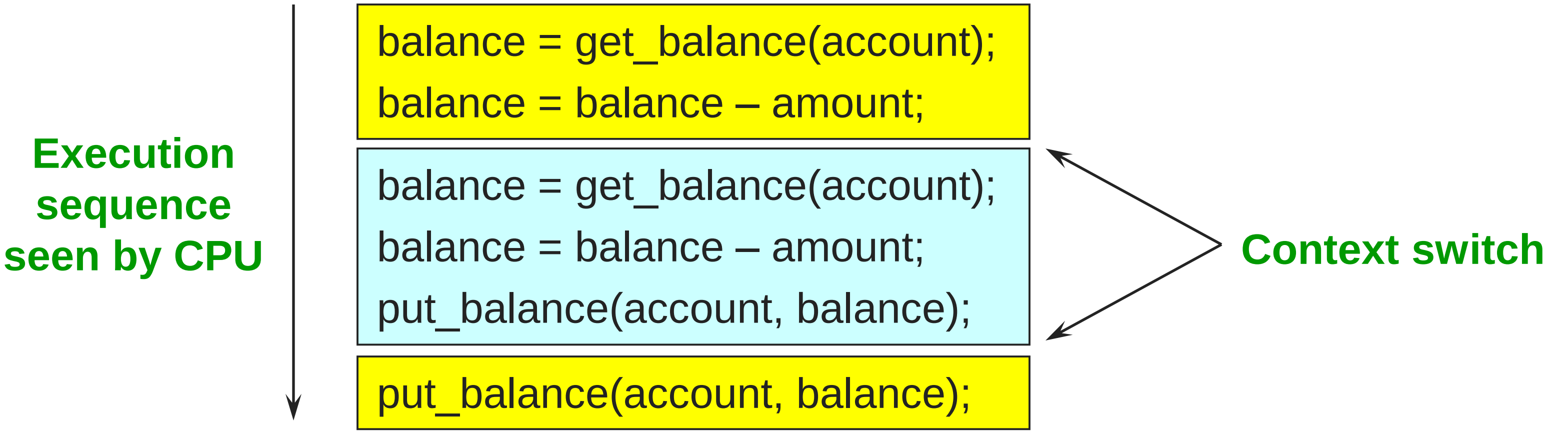
```
withdraw (account, amount) {
    balance = get_balance(account);
    balance = balance – amount;
    put_balance(account, balance);
    return balance;
}
```

```
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:



**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

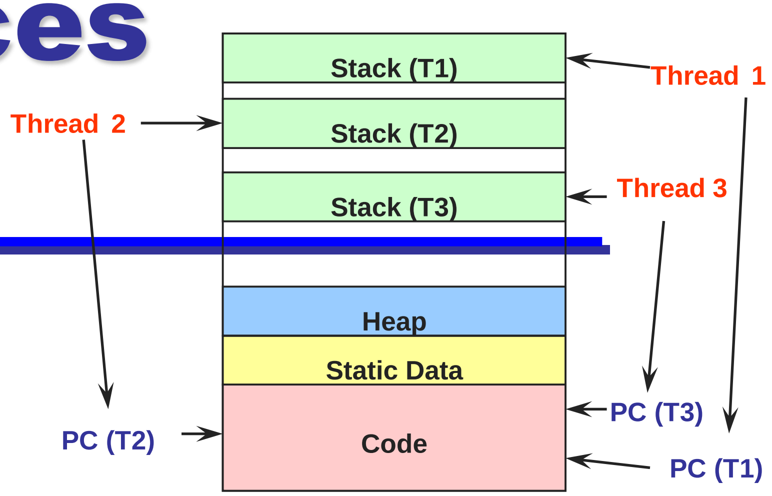
- Problem: two threads accessed a shared resource
  - Known as a race condition (remember this buzzword!)

- 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?

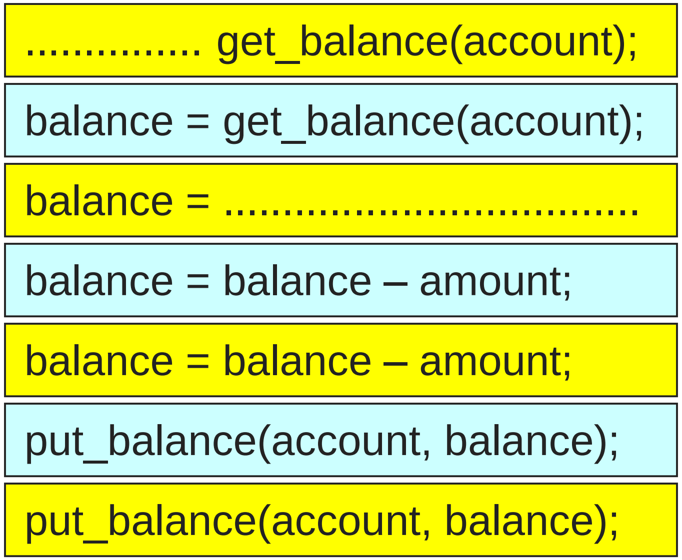


- Local variables?
  - Not shared: refer to data on the stack
  - Each thread has its own stack
  - Never pass/share/store a pointer to a local variable on the stack for thread T1 to another thread T2
- Global variables and static objects?
  - Shared: in static data segment, accessible by all threads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

# 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
}
```

$$(\text{blue at } 4) \wedge try1 \wedge (turn == 1 \vee \neg try2 \vee (try2 \wedge (\text{yellow at 6 or at 7}))$$

$$\wedge\ (\text{yellow at } 8) \wedge try2 \wedge (turn == 2 \vee \neg try1 \vee (try1 \wedge (\text{blue at 2 or at 3}))$$

$$\ldots \Rightarrow (turn == 1\ \wedge turn == 2)$$

# 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