Lecture 3: Threads & Synchronization
CSE 120: Principles of Operating Systems

UC San Diego: Summer Session I, 2009
Frank Uyeda
Announcements

• Homework 1 is due now.
• PeerWise question & reviews due tomorrow.
• Project 1 milestone and deadline have been extended.
  – Milestone Friday 7/10
  – Deadline Monday 7/13
Review

• Processes
  – System resources
  – Instructions to run (program)
  – Created with fork()

• Process Control Block
  – Maintained by the kernel
  – Hold all information about the process

• Interprocess Communication
  – Shared memory
  – Message passing
Shared Memory vs. Message Passing

Process A

Process B

hello world (stdout)

hello world

shared_mem

Kernel

Physical Memory

Process A

Process B

Kernel

Physical Memory
On a UNIX system, a process makes a call to `fork()` and stores the return value in variable `pid`. What is the value of `pid` in the child process?

- The process id of the parent.
- A random non-zero process id.
- Zero.
- The process id of the child.
- The process id of the parent incremented by one.
• Which of the followings is NOT TRUE:
  – Many context switches don't cause a problem.
  – The roles of OS are to provide abstractions and manage resources.
  – A program can have multiple processes.
  – Not focused on one particular subject
  – Option 1 should be more clear
    • Context switches incur a performance slow down
    • It is that a “problem”?
  – Option 3 is not true in a uniprogramming system.
PeerWise

• Which of the followings is NOT TRUE:
  – Many context switches don't cause a problem.
  – The roles of OS are to provide abstractions and manage resources.
  – A program can have multiple processes.

• On a multiprogramming system, which on the following is NOT true:
  – A program can have multiple processes
  – A process can have multiple programs
  – A process always has a program
Goals for Today

• Threads
  – What do they provide that processes do not?
  – How do we implement them?

• Synchronization
  – What are Critical Sections?
  – How can we enforce synchronization with locks?
Web Server example revisited

• Web server example using Unix fork()

• Recall motivation for parallelism
  – Child processes can handle similar, sub-tasks
  – Sub-tasks are generally I/O intensive
  – I/O requests can overlap with computation
  – Increases overall system throughput

while (1) {
    int sock = accept();
    if (child_pid = fork() == 0) {
        Handle client request
        Close socket and exit
    } else {
        Close socket
    }
}
Web Server example revisited

- Web server example using Unix fork()

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork())== 0) {
        Handle client request
        Close socket and exit
    }
    else {
        Close socket
    }
}

while (1) {
    int sock = accept();
    if ((child_pid = fork())== 0) {
        Handle client request
        Close socket and exit
    }
    else {
        Close socket
    }
}
```

parent

child
Web Server example revisited

- Web server example using Unix fork()

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork())== 0) {
        Handle client request
        Close socket and exit
    } else {
        Close socket
    }
}
```
Inefficient Concurrency

• Space
  – Storing large PCBs
  – Redundant data storage (address spaces)

• Time
  – Time spent creating PCBs
  – Time spent context switching large PCBs

Key Idea: separate the concept of a process from its execution state
Threads

• Separate the concept of a process from its execution state
  – Process: address space, privileges, resources
  – Execution state: registers, PC, SP, FP

• Execution state is called a thread of control
  – Each thread belongs to a process
  – Each process has at least one thread
  – Threads are the units of scheduling (sort of)
    • CPU runs one thread at a time
    • Threads execute within the context of a process
Thread in a Process

Address Space

*0x00....... (Starting Address)*

*0xFFF..... (Ending Address)*

Stack

- SP1

Heap

Data Segment

Text Segment

- PC1
Threads in a Process

Address Space

0xFFF..... (Ending Address)

0x00....... (Starting Address)

Stack (Thread 1)

Stack (Thread 2)

Stack (Thread 3)

SP1

SP2

SP3

PC1

PC2

PC3

Heap

Data Segment

Text Segment

Thread 1

Thread 2

Thread 3
Thread Design Space

<table>
<thead>
<tr>
<th>Thread Design</th>
<th>MS-DOS</th>
<th>Classic Unix</th>
<th>Java VM</th>
<th>Solaris, Unix, Linux, XP, Vista, Mac OS X</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Thread Per Process</td>
<td>One Address Space</td>
<td>Many Address Spaces</td>
<td>Many Address Spaces</td>
<td>Many Address Spaces</td>
</tr>
<tr>
<td>Many Threads Per Process</td>
<td>One Address Space</td>
<td>Many Address Spaces</td>
<td>Many Address Spaces</td>
<td>Many Address Spaces</td>
</tr>
</tbody>
</table>

Note: slide adapted from Alex Snoeren
Using `fork()` for concurrency

```c
while (1) {
    int sock = accept();
    if ((child_pid = fork())== 0) {
        Handle client request
        Close socket and exit
    }
    else {
        Close socket
    }
}
```

Before

After
Multi-threaded Web Server

Using `thread_fork()` for concurrency

```c
...  
  while (1) {
    int sock = accept();
    thread_fork(handle_req, sock);
  }
}

handle_req(int sock) {
  Handle client request
  close(sock)
}
```
Using Threads

• Threads are useful for parallel applications
  – Threads abstraction captures parallelism from multiprocessing while using only a single process
  – Requires less OS overhead across time and space
  – No need for IPC between threads in same process
Scheduling Threads

• Scheduler still schedules per-process
  – Threads run in context of current PCB
  – How are the threads scheduled?

• There are two basic approaches
  – Kernel decides
  – Process (user-level) decides

• Why does this matter?
Kernel and User Threads (2)

• Java Virtual Machine (JVM) example
  – Java threads are user-level threads
  – Classic Unix: one kernel thread per-process
    • All Java threads would be on single kernel thread
  – On XP, Modern Unix: can have multiple kernel threads per-process
    • Can multiplex Java threads on multiple kernel threads
Implementing Threads

• Implementing threads can be difficult
  – Interface
  – Scheduling
    • Preemptive vs non-preemptive scheduling
    • Context switch
  – Synchronization

• Focus on user-level threads
  – Kernel-level threads similar to process management in OS
  – What you will be partially implementing in Nachos
Thread Interface

• thread_fork(procedure_t)
  – Create a new thread of control
• thread_start(thread_t)
  – Start the given thread
• thread_stop()
  – Stop the calling thread; also thread_block()
• 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
  - Scheduling queues keep track of blocked threads and ready threads
- Run queue: Threads currently running (1)
- Ready queue: Threads ready to run
- Wait queues: Threads waiting for something
Non-Preemptive Scheduling

- Thread voluntarily gives up the CPU with `thread_yield()`

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

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

What is the output of running these two threads?
thread_yield()

- The thread invoking thread_yield() voluntarily gives up the processor
  - Calling thread changes from running to ready
  - Scheduler selects another thread from ready queue to run
  - There is a context switch from the calling thread to the next thread

- What does it mean for thread_yield() to return?
  - It could be in the context of another thread
Implementing thread_yield()

```c
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
Thread Context Switch

• Thread context switch is similar to process context switch
  – Saves context of currently running thread (old_thread)
  – Restores context of the next thread
  – Next thread becomes current_thread
  – Returns to caller as new thread

• Fortunately, Nachos has written this method for you
Preemptive Scheduling

• What’s the danger with non-preemptive threads?
  – A thread can “own” CPU
  – Only voluntary calls of `thread_yield()`, `stop()`, `exit()"
  releases the CPU

• **Preemptive scheduling causes involuntary context switch**
  – Uses timer interrupt to force `thread_yield()` call
  – Nachos uses this technique
Threads Summary

• Parallelism useful for many applications
  – Processes are too heavyweight
  – Separating parallel tasks as threads improves efficiency of multitasking

• Two primary types of threads
  – Kernel-level threads: can be efficient, but has overhead
  – User-level threads: much faster, but not supported well by OS

• Are there any drawbacks to threads?
  – Difficult to program correctly
    • No longer assume exclusive use of the resources.
  – It’s hard to get them to cooperate
Break & KQS Cards

• K: what do you want me to Keep doing?
• Q: what do you want me to Quit doing?
• S: what do you want me to Start doing?

• Is there a specific topic that we’ve gone over that is still unclear to you?
Support for Concurrency

• Concurrent access to a shared variable can be tricky
  – Shared variables can be read/modified by any thread
  – Thread execution can be interleaved arbitrarily by preemptive scheduler
  – Access to any shared state must therefore be synchronized to avoid unexpected and unwanted behavior

• We will look at:
  – Common synchronization mechanisms for accessing shared data
    • Locks, mutexes, semaphores, condition variables, monitors
  – Common problems and patterns for using synchronization mechanisms
    • Bounder buffer, Producer/Consumer,...
Synchronization

• 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 (ping-pong example)

• For correctness, we need to control this cooperation
  – Threads interleave executions arbitrarily and at different rates
  – Scheduling is not under program control

• Cooperation is controlled using synchronization
  – Restrict the possible interleavings

• We’ll discuss in terms of threads, also applies to processes
  – Remember Shared Memory IPC?
When Are Resources Shared?

• Local variables are not shared (private)
  – Refer to data on the stack
  – Each thread has its own stack
  – Should never pass/share a pointer to a local variable on the stack for thread T1 to another thread T2

• Global variables and static objects are shared
  – Stored in static data segment
  – Accessible by any thread

• Dynamic objects (heap) are shared
  – Allocated from heap with malloc/free or new/delete
Threading Assumptions

• We assume that the only atomic actions are reads and writes of words
  – Not always the case on some architectures
• We assume that a context switch can occur at any time.
• We assume that you can delay a thread as long as you like as long as it’s not delayed forever.
Classic Example

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

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

• Now suppose that you and your significant other share a bank account with a balance of $1000

• Then you each go to a separate ATM and simultaneously withdraw $100 from the account.
We’ll represent the situation by creating a thread for each person to do the withdrawals

These threads run in the same Bank process

```java
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 the potential sequence of executions for these threads
The problem is that the execution of the two threads can be interleaved:

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

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

What is the balance of the account?

This is known as a race condition:
- The outcome of a computation is dependent on the timing or interleaving of instructions.
Mutual Exclusion

• One way to ensure consistent behavior is to only let one thread “win the race”
  – This technique is called mutual exclusion

• Code that uses mutual exclusion to synchronize its execution is called a critical section
  – Only one thread at a time can execute in the critical section
  – All other threads are forced to wait to enter
  – When a thread leaves a critical section, another can enter

```java
withdraw(account, amount) {
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    return balance;
}
```
Critical Section Requirements

1. Mutual exclusion
   - If one thread is in the critical section, then no other is

2. Progress
   - 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. No assumptions on performance
   - Requirements must be met with any number of CPUs with arbitrary relative speeds
Locks

• One way to implement critical sections is to “lock the door” on the way in, and unlock it again on the way out.

• 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
Using Locks

withdraw(account, amount) {
    acquire(lock);
    balance = get_balance(account);
    balance = balance - amount;
    put_balance(account, balance);
    release(lock);
    printf("balance %f", balance)
    return balance;
}

acquire(lock);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
release(lock);
printf("balance %f", balance)

acquire(lock);
balance = get_balance(account);
balance = balance - amount;
put_balance(account, balance);
release(lock);
printf("balance %f", balance)

• What happens when orange tries to acquire lock?
• Why is the “print” outside the critical section? Is this ok?
• What happens when a third thread calls acquire?
• What can happen if the calls are not paired?
Implementing Locks: Spin Locks

• Let’s try a spin lock

    ```c
    struct lock {
        int held = 0;
    }

    void acquire(lock) {
        while (lock->held);  // Busy-wait (spin-wait)
        lock->held = 1;
    }

    void release(lock) {
        lock->held = 0;
    }
    ```

• This is called a spin lock because a thread spins waiting (busy-waiting) for the lock to be released

• Does this work?
Problem (1)

Program Code:

```c
void acquire(lock);
...
void release(lock);
```

Sequence of Execution:

```c
struct lock { int turn = 0; }

void acquire(lock) {
    while (lock->held);
}

void acquire(lock) {
    while (lock->held);
    lock->held = 1;
}

lock->held = 1;
```

FAIL! Two processes both in the critical section!
Taking Turns

• How did we solve this problem in Kindergarten?
  – Let’s assume only two threads, and take turns
  – Assume **turn** initialized to one of threads

• Does this work?
  – Why not?

```
struct lock {
    int turn = 0;
}

void acquire(lock) {
    while (lock->turn != this_thread);
}

void release(lock) {
    lock->turn = other_thread;
}
```
FAIL! Progress condition violated.
Blue thread has not requested the lock, yet the orange thread still has to wait.
Declaring Intent

• Problem was we didn’t know if other thread was ready
  – Let’s be polite and wait until the other thread isn’t interested

struct lock {
  int interested[2] = [FALSE,FALSE];
}
void acquire(lock) {
  lock->interested[this_thread] = TRUE;
  while (lock->interested[other_thread]);
}
void release(lock) {
  lock->interested[this_thread] = FALSE;
}
Problem (3)

Program Code:

```c
void acquire(lock);
...
void release(lock);
```

Sequence of Execution:

```c
struct lock {
    int interested[2] = [FALSE,FALSE];
}

void acquire(lock) {
    lock->interested[this_thread] = TRUE;
}

void acquire(lock) {
    lock->interested[this_thread] = TRUE;
}

while (lock->interested[other_thread]);

while (lock->interested[other_thread]);
```

FAIL! Neither thread can make any progress.
**Peterson’s Algorithm**

- Take turns only if somebody else is interested; otherwise just go

```c
struct lock {
    int turn = 0;
    int interested[2] = [FALSE, FALSE];
}

void acquire(lock) {
    lock->interested[this_thread] = TRUE;
    turn = other_thread;
    while (lock->interested[other_thread] && turn == other_thread);
}

void release(lock) {
    lock->interested[this_thread] = FALSE;
}
```
Test-and-Set

• The semantics of test-and-set are:
  1. Record the old value \textit{and}
  2. Set the value to indicate available \textit{and}
  3. Return the old value

• Hardware executes it atomically!
  – Remember: atomic means “all or none”

```c
bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

• When executing test-and-set on “flag”
  – What is value of flag afterwards if it was initially False? True?
  – What is the return result if flag was initially False? True?
Using Test-and-Set

• Here is a simple lock implementation with test-and-set:

```c
struct lock {
    int held = 0;
}

void acquire(lock) {
    while (test-and-set(&lock->held));
}

void release(lock) {
    lock->held = 0;
}

bool test_and_set(bool *flag) {
    bool old = *flag;
    *flag = True;
    return old;
}
```

• When will the while return?
• What about multiprocessors?
Problems with Spin Locks

• The problem with spinlocks is that they are wasteful
  – If a thread is spinning on a lock, then the thread holding the lock cannot make progress

• How did the lock holder give up the CPU in the first place?
  – Lock holder calls yield or sleep
  – Involuntary context switch

• Only want to use spin locks as primitives to build higher-level synchronization constructs
Disabling Interrupts

• Another implementation of acquire/release is to disable interrupts:

```c
struct lock {
    int held = 0;
};
void acquire(lock) {
    disable interrupts;
}
void release(lock) {
    enable interrupts;
}
```

• Note that there is no state associated with the lock
• Can two threads disable interrupts simultaneously?
On Disabling Interrupts

• Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., time)
  – Effectively gives thread exclusive access to the CPU
• In a “real” system, this is only available to the kernel
  – Why?
• Disabling interrupts is insufficient on a multiprocessor
  – Back to atomic instructions
• Like spin locks, only want to disable interrupts to implement higher-level synchronization primitives
  – Don’t want interrupts disabled between acquire() and release()
Locks in Nachos

public Lock() {
}
private KThread lockHolder = null;
Private ThreadQueue waitQueue;

public void acquire() {
    boolean intStatus = Machine.interrupt().disable();
    KThread thread = KThread.currentThread();
    if (lockHolder != null) {
        waitQueue.waitForAccess(thread);
        KThread.sleep();
    } else {
        waitQueue.acquire(thread);
        lockHolder = thread;
    }
    Machine.interrupt().restore(intStatus);
}

public void release() {
    boolean intStatus = Machine.interrupt().disable();
    lockHolder= waitQueue.nextThread();
    if (lockHolder != null) {
        lockHolder.ready();
    }
    Machine.interrupt().restore(intStatus);
}

• In nachos/threads/Lock.java
• Note that some kernel operations require interrupts to be disabled
  – Can you identify the critical section?
  – How can the thread sleep with interrupts disabled?
Summarize Where We Are

- **Goal**: Use mutual exclusion to protect critical sections of that access shared resources
- **Method**: Use locks (spin locks or disable interrupts)
- **Problem**: Critical sections can be long

**Spin locks:**
- Threads waiting to acquire lock spin in test-and-set loop
- Wastes CPU cycles
- Longer the CS, the longer the spin
- Greater the chance for lock holder to be interrupted

```plaintext
acquire(lock)  
...  
**Critical Section**  
...  
release(lock)
```

**Disabling Interrupts:**
- Should not disable interrupts for long periods of time
- Can miss or delay important events (e.g., timer, I/O)
Higher-Level Synchronization

- Spin Locks and disabling interrupts are useful only for very short and simple critical sections
  - Wasteful otherwise
  - These primitives are “primitive” – don’t do anything besides mutual exclusion
- Need higher-level synchronization primitives that:
  - Block waiters
  - Leave interrupts enabled within the critical section
- All synchronization requires atomicity
- So we’ll use our “atomic” locks as primitives to implement them
Lock Summary

• Correctly implementing locks is hard!
• We need mutual exclusion when working with threads to ensure correctness.
  – Only one thread can enter at a time
• Critical Section requirements
  – Mutual exclusion
  – Progress
  – Bounded Wait
• We can use locks to guard critical sections.
  – acquire() & release()
Next Time

• Read Chapters 6.5-6.10
• PeerWise question & reviews due tomorrow.
• Check Web site for course announcements
  – http://www.cs.ucsd.edu/classes/su09/cse120