CS 162: Operating Systems and Systems Programming

Lecture 4: Threads & Synchronization

June 27, 2019
Instructor: Jack Kolb
https://cs162.eecs.berkeley.edu
Recall: Process Management

- **fork** – copy the current process
- **exec** – change the *program* being run by the current process
- **wait** – wait for a process to finish
- **kill** – send a *signal* (interrupt-like notification) to another process
- **sigaction** – set handlers for signals
Recall: Process Management

```
main () {
    ...
}
```

```
fork

pid = fork();
if (pid == 0)
    exec(...);
else
    wait(pid);

wait

pid = fork();
if (pid == 0)
    exec(...);
else
    wait(pid);
```
Recall: Multiplexing Processes

- Snapshot of each process in its PCB
  - Only one active at a time (for now…)
- Give out CPU to different processes
  - Scheduling
- Policy Decision
- Give out non-CPU resources
  - Memory/IO
  - Another policy decision
Recall: Context Switch Overhead.
Recall: Lifecycle of a Process

- As a process executes, it changes state:
  - **new**: being created
  - **ready**: waiting to run
  - **running**: instructions executing on the CPU
  - **waiting**: waiting for some event to occur (e.g., keypress)
  - **terminated**: finished execution
Recall: Adding Threads

• Version of program with Threads (loose syntax):

```c
main() {
    thread_fork(ReadLargeFile, "pi.txt");
    thread_fork(RenderUserInterface, "classlist.txt");
}
```

• `thread_fork`: Start independent thread running given procedure

• What is the behavior here?
  • Now, you would actually see the class list
  • This *should* behave as if there are two separate CPUs
Recall: Switching Threads

• Consider the following code blocks:
  ```
  func A() {
    B();
  }
  func B() {
    while(TRUE) {
      yield();
    }
  }
  ```

• Two threads, S and T, each run A

Thread S's switch returns to Thread T's (and vice versa)
Aren't we still "context switching" between threads?

• Yes, but **much cheaper** than switching processes
  • No need to change address space

• Some numbers from Linux:
  • Frequency of context switch: 10-100ms
  • Switching between processes: 3-4 μsec.
  • Switching between threads: 100 ns
Processes vs. Threads

- **Switch overhead**:  
  - Same process: low  
  - Different proc.: high

- **Protection**:  
  - Same proc: low  
  - Different proc: high

- **Sharing overhead**:  
  - Same proc: low  
  - Different proc: high
Processes vs. Threads

- **Switch overhead:**
  - Same process: low
  - Different proc.: high

- **Protection**
  - Same proc: low
  - Different proc: high

- **Sharing overhead**
  - Same proc: low
  - Different proc: high
Example: Multithreaded Server

serverLoop() {
    connection = AcceptNewConnection();
    thread_fork(ServiceWebPage, connection);
}

• One thread per connection
• Problem: How fast is creating threads?
  • Better than fork(), but still overhead
• Problem: What if we get a lot of requests?
  • Might run out of memory (thread stacks)
  • Schedulers usually have trouble with too many threads
Web Server: Thread Pools

• **Bounded** pool of worker threads
  • Allocated in **advance**: no thread creation overhead
  • **Queue** of pending requests

![Diagram showing the flow of a request from a client, through a master thread, and into a thread pool with a queue.](image-url)
Multiprocessing vs Multiprogramming

- Multiprocessing: Multiple cores
- Multiprogramming: Multiple Jobs/Processes
- Multithreading: Multiple threads/processes

What does it mean to run two threads concurrently?
- Scheduler is free to run threads in any order and interleaving
Yield is covered, what about I/O?

- User code invokes syscall
- IO operation initiated (more later)
- Run new thread, switch
- Really, same thing as before
  - Just put the thread on a different queue
Preempting a Thread

• What happens if thread never does any I/O, never waits, and never yields control?
  • Must find way that dispatcher can regain control!

• **Interrupts**: signals from hardware or software that stop the running code and jump to kernel
  • Timer: like an alarm clock that goes off every some milliseconds

• Interrupt is a hardware-invoked mode switch
  • Handled immediately, no scheduling required
Example: Network Interrupt

- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately

Hardware instructions:

```assembly
... add $r1, $r2, $r3
subi $r4, $r1, #4
slli $r4, $r4, #2
... lw $r2, 0($r4)
lw $r3, 4($r4)
add $r2, $r2, $r3
sw 8($r4), $r2
...
```

Actions after interrupt:

1. **Raise priority** (set mask)
2. **Reenable All Ints**
3. **Save registers**
4. **Dispatch to Handler**
5. **Transfer Network Packet from hardware to Kernel Buffers**
6. **Restore registers**
7. **Clear current Int**
8. **Disable All Ints**
9. **Restore priority** (clear Mask)
10. **RTI**
Switching Threads from Interrupts

- Prevent thread from running forever with **timer interrupt**

```c
timerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```

- Same thing from IO interrupts
  - Example: immediately start process waiting for keypress
How does a thread get started?

- Can't call `switch()` without starting a thread
- How do we make a new thread?

```c
SetupNewThread(tNew) {
    ...
    TCB[tNew].regs.sp = newStack;
    TCB[tNew].regs.retpc = &ThreadRoot;
}
```
How does a thread get started?

- So when does the new thread really start executing?

run_new_thread selects this thread's TCB, "returns" into beginning of ThreadRoot
Bootstrapping Threads: ThreadRoot

ThreadRoot() {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    call fcnPtr(fcnArgPtr);
    ThreadFinish();
}

- Stack will grow and shrink with execution of thread
- ThreadRoot() never returns
  - ThreadFinish() destroys thread, invokes scheduler
So Far: Kernel-Supported Threads

- Each thread has a **thread control block**
  - CPU registers, including PC, pointer to stack
  - Scheduling info: priority, etc.
  - Pointer to **Process control block**
- OS scheduler uses TCBs, not PCBs

![Diagram of Thread Control Block and Process Control Block](image)
Kernel-Supported Threads

- Process $P_0$ is marked as not supported.
- Thread T0 is executing.
- Thread T1 is executing.
- Operating system processes interrupts or system calls.
- TCB0 and TCB1 are save and reload state operations.
- Idle states are denoted for TCB0 and TCB1.
Kernel-Supported Threads

• Threads run and block (e.g., on I/O) independently
• One process may have multiple threads waiting on different things
• Two mode switches for every context switch (expensive)
• Create threads with syscalls

• Alternative: multiplex several streams of execution (at user level) on top of a single kernel thread
User-Mode Threads

• User program contains its own scheduler
• Several user threads per kernel thd.
• User threads may be scheduled non-preemptively
  • Only switch on yield
• Context switches cheaper
  • Copy registers and jump (switch in userspace)
User-Mode Threads: Problems

• One user-level thread blocks on I/O: they all do
  • Kernel cannot adjust scheduling among threads it doesn’t know about

• Multiple Cores?

• Can't completely avoid blocking (syscalls, page fault)

• Solution: *Scheduler Activations*
  • Have kernel inform user-level scheduler when a thread blocks
Some Threading Models

Simple One-to-One Threading Model

Almost all current implementations

Many-to-One

Many-to-Many
Logistics

• Group formation enabled on autograder
  • Sign up by Friday 11:59 PM
• HW0 due on Friday, 11:59 PM
• C Review & Pintos Intro: 11am-1pm Friday
  • Wozniak Lounge (438 Soda Hall)
• Project 1 Released Friday
  • Design Reviews with your TA next week
Thread Abstraction

- Illusion: Infinite number of processors
Thread Abstraction

- Illusion: Infinite number of processors
- Reality: Threads execute with variable “speed”
  - Programs must be designed to work with any schedule
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$x = x + 1;$</td>
<td>$x = x + 1;$</td>
</tr>
<tr>
<td>$y = y + x;$</td>
<td>$y = y + x;$</td>
</tr>
<tr>
<td>$z = x + 5y;$</td>
<td>$z = x + 5y;$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Possible Executions

a) One execution

b) Another execution

c) Another execution
Correctness with Concurrent Threads

- Non-determinism:
  - Scheduler can run threads in **any order**
  - Scheduler can switch threads **at any time**
  - This can make testing very difficult

- *Independent Threads*
  - No state shared with other threads
  - Deterministic, reproducible conditions

- *Cooperating Threads*
  - Shared state between multiple threads

- **Goal: Correctness by Design**
Remember: Multiprogramming

- Scheduler can run threads in any order
- And with multiple cores:
  - Even more interleaving
  - **Could truly be running at the same time**
Race Conditions

• What are the possible values of x below?
  • Initially x = y = 0;

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 2;</td>
</tr>
</tbody>
</table>

• Must be 1. Thread B cannot interfere.
Race Conditions

• What are the possible values of \( x \) below?
• Initially \( x = y = 0; \)

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = y + 1; )</td>
<td>( y = 2; )</td>
</tr>
<tr>
<td></td>
<td>( y = y \times 2; )</td>
</tr>
</tbody>
</table>

• 1 or 3 or 5 (non-deterministic)
• Race Condition: Thread A races against Thread B
Race Conditions

• What are the possible values of $x$ below?
• Initially $x = y = 0$;

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1;$</td>
<td>$y = 2;$</td>
</tr>
</tbody>
</table>

• Why not 3?
  • Write a bit at a time
Atomic Operations

• Definition: **An operation that runs to completion or not at all**
  • Need some to allow threads to work together

• Example: Loading or storing words
  • **Result of 3 is not possible on most machines**

• Some instructions are not atomic
  • Ex: double-precision floating point store
Real-Life Analogy: Too Much Milk

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away</td>
</tr>
</tbody>
</table>
Too Much Milk: Correctness

1. At most one person buys milk

2. At least one person buys milk if needed
Solution Attempt #1

- Leave a note
  - Place on fridge before buying
  - Remove after buying
  - Don’t go to store if there’s already a note

- Leaving/checking a note is atomic (word load/store)

```java
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove Note;
    }
}
```
Attempt #1 in Action

Alice
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove Note;
  }
}

Bob
if (noMilk) {
  if (noNote) {
    leave Note;
    buy milk;
    remove note;
  }
}
Solution Attempt #2

```java
leave Note;
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
    }
}
remove Note;
```

But there’s always a note – you just left one!

At least you don’t buy milk twice…
Solution Attempt #3

• Leave a named note – each person ignores their own

Alice
leave note Alice
if (noMilk) {
  if (noNote Bob) {
    buy milk
  }
}
remove note Alice;

Bob
leave note Bob
if (noMilk) {
  if (noNote Alice) {
    buy milk
  }
}
remove note Bob;
Alice

leave note Alice

if (noMilk) {

    if (noNote Bob) {
        buy milk
    }
}

remove note Alice

Bob

leave note Bob

if (noMilk) {

    if (noNote Alice) {
        buy milk
    }
}

remove note Bob
Solution Attempt #4

Alice
leave note Alice
while (note Bob) {
    do nothing
}
if (noMilk) {
    buy milk
}
remove note Alice;

Bob
leave note Bob
if (noNote Alice) {
    if (noMilk) {
        buy milk
    }
}
remove note Bob;

• This is a correct solution
Issues with Solution 4

• Complexity
  • Proving that it works is hard
  • How do you add another thread?

• Busy-waiting
  • Alice consumes CPU time to wait
Break
Relevant Definitions

- **Mutual Exclusion**: Ensuring only one thread does a particular thing at a time (one thread excludes the others)

- **Critical Section**: Code exactly one thread can execute at once
  - Result of mutual exclusion
Relevant Definitions

• **Lock:** An object only one thread can hold at a time
  • **Provides** mutual exclusion

• Offers two **atomic** operations:
  • `Lock.Acquire()` – wait until lock is free; then grab
  • `Lock.Release()` – Unlock, wake up waiters
Using Locks

MilkLock.Acquire()
if (noMilk) {
    buy milk
}
MilkLock.Release()

But how do we implement this?
Implementing Locks: Single Core

• Idea: A context switch can only happen (assuming threads don’t yield) if there’s an interrupt

• “Solution”: Disable interrupts while holding lock
Naïve Interrupt Enable/Disable

Acquire() {
    disable interrupts;
}

Release() {
    enable interrupts;
}

• Problem: User can stall the entire system
  Lock.Acquire()
  While (1) {}

• Problem: What if we want to do I/O?
  Lock.Acquire()
  Read from disk
  /* OS waits for (disabled) interrupt)! */
Implementing Locks: Single Core

- Idea: Disable interrupts for **mutual exclusion** on accesses to value indicating lock status

```c
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread() // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone waiting) {
        take a thread off queue;
    } else {
        Value = FREE;
    }
    enable interrupts;
}
```
Reenabling Interrupts When Waiting

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread();
    } else {
        value = BUSY;
    }
    enable interrupts;
}

• Before on the queue?
  • Release might not wake up this thread!

• After putting the thread on the queue?
  • Gets woken up, but immediately switches away
Reenabling Interrupts When Waiting

```c
Acquire() {
    disable_interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread()
    } else {
        value = BUSY;
    }
    enable_interrupts;
}
```

- Best solution: after the current thread suspends
- How?
  - `run_new_thread()` should do it!
  - Part of returning from `switch()`
Summary

• Kernel vs. User-Mode Threads
  • Kernel threads: no fate-sharing with IO, but requires syscalls and mode transitions
  • User-mode threads: Lightweight, rely on yield, invisible to kernel scheduler

• Synchronization
  • Concurrency useful for overlapping computation and IO
  • But makes it harder to write correct code:
    • Arbitrary interleavings
    • Could access shared resources while in bad state
    • Solution: careful design
  • Building block: atomic operations