CS162
Operating Systems and Systems Programming
Lecture 5

Concurrency and Mutual Exclusion

February 4th, 2020
Prof. John Kubiatowicz
http://cs162.eecs.Berkeley.edu

Acknowledgments: Lecture slides are from the Operating Systems course taught by John Kubiatowicz at Berkeley, with few minor updates/changes. When slides are obtained from other sources, a reference will be noted on the bottom of that slide, in which case a full list of references is provided on the last slide.
Recall: Fork, Wait, and (optional) Exec

```c
int cpid = fork();
if (cpid > 0) {
    // Parent Process
    int mypid = getpid();
    printf("[%d] parent of [%d]\n", mypid, cpid);
    int tcpid = wait(
        &status);
    printf("[%d] bye %d\n", mypid, tcpid);
} else if (cpid == 0) {
    // Child Process
    int mypid = getpid();
    printf("[%d] child\n", mypid);
    execl(filename, (char *)0);  // Opt: start new program
} else { // Error! }
```

- Return value from Fork: integer
  - When > 0: return value is pid of new child (Running in Parent)
  - When = 0: Running in new Child process
  - When < 0: Error! Must handle somehow
- Wait() system call: wait for next child to exit
  - Return value is PID of terminating child
  - Argument is pointer to integer variable to hold exit status
- Exec() family of calls: replace process with new executable
Recall: Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a “signal” from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```c
computePI() {
    while(TRUE) {
        ComputeNextDigit();
        yield();
    }
}
```
• How do we run a new thread?

```c
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); // Do any cleanup */
}
```

• How does dispatcher switch to a new thread?
  – Save anything next thread may trash: PC, regs, stack pointer
  – Maintain isolation for each thread
• Consider the following code blocks:

```java
proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
```

• Suppose we have 2 threads:
  – Threads S and T

Thread S's switch returns to Thread T's (and vice versa)
Goals for Today

• Finish discussion of Threads
• Concurrency and need for Synchronization Operations
• Basic Synchronization through Locks
• Initial Lock Implementations
What happens when a thread requests a block of data from the file system?
- User code invokes a system call
- Read operation is initiated
- Run new thread/switch

Thread communication similar
- Wait for Signal/Join
- Networking
External Events

• What happens if thread never does any I/O, never waits, and never yields control?
  – Could the ComputePI program grab all resources and never release the processor?
    » What if it didn’t print to console?
  – Must find way that dispatcher can regain control!

• Answer: utilize external events
  – 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

• If we make sure that external events occur frequently enough, can ensure dispatcher runs
• Interrupts invoked with interrupt lines from devices
• Interrupt controller chooses interrupt request to honor
  – Interrupt identity specified with ID line
  – Mask enables/disables interrupts
  – Priority encoder picks highest enabled interrupt
  – Software Interrupt Set/Cleared by Software
• CPU can disable all interrupts with internal flag
• Non-Maskable Interrupt line (NMI) can’t be disabled
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

```
add $r1,$r2,$r3
subi $r4,$r1,#4
slli $r4,$r4,#2
...  
PC saved
```

```
Enable all Ints
Kernel Mode
```

```
Restore priority
(set mask)
```

```
Reenable All Ints
Save registers
Dispatch to Handler
```

```
Transfer Network
Packet from hardware
to Kernel Buffers
```

```
Restore registers
Clear current Int
```

```
Disable All Ints
```

```
RTI
```

```
“Interrupt Handler”
```

```
lw $r2,0($r4)
lw $r3,4($r4)
add $r2,$r2,$r3
sw 8($r4),$r2
...  
```

```
External Interrupt
Pipeline Flush
...  
```

```
Add $r1,$r2,$r3
Subi $r4,$r1,#4
Slli $r4,$r4,#2
...  
```

```
Save registers
Dispatch to Handler
Reenable All Ints
```

```
Transfer Network
Packet from hardware
to Kernel Buffers
```

```
Restore registers
Clear current Int
```

```
Disable All Ints
```

```
RTI
```

```
“Interrupt Handler”
```

```
lw $r2,0($r4)
lw $r3,4($r4)
add $r2,$r2,$r3
sw 8($r4),$r2
...  
```

```
External Interrupt
Pipeline Flush
...  
```

```
Add $r1,$r2,$r3
Subi $r4,$r1,#4
Slli $r4,$r4,#2
...  
```

```
Save registers
Dispatch to Handler
Reenable All Ints
```

```
Transfer Network
Packet from hardware
to Kernel Buffers
```

```
Restore registers
Clear current Int
```

```
Disable All Ints
```

```
RTI
```

```
“Interrupt Handler”
```

```
lw $r2,0($r4)
lw $r3,4($r4)
add $r2,$r2,$r3
sw 8($r4),$r2
...  
```
Use of Timer Interrupt to Return Control

• Solution to our dispatcher problem
  – Use the timer interrupt to force scheduling decisions

• Timer Interrupt routine:

```c
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
Hardware context switch support in x86

• Syscall/Intr (U → K)
  - PL 3 → 0;
  - TSS ← EFLAGS, CS:EIP;
  - SS:SP ← k-thread stack (TSS PL 0);
  - push (old) SS:ESP onto (new) k-stack
  - push (old) eflags, cs:eip, <err>
  - CS:EIP ← <k target handler>

• Then
  - Handler then saves other regs, etc
  - Does all its works, possibly choosing other threads, changing PTBR (CR3)

  - kernel thread has set up user GPRs

• iret (K → U)
  - PL 0 → 3;
  - Eflags, CS:EIP ← popped off k-stack
  - SS:SP ← user thread stack (TSS PL 3);

Pintos: tss.c, intr-stubs.S
Pintos: Kernel Crossing on Syscall or Interrupt

User

stack

user
code

Kernel

code

Kernel

thread

stack

syscall / interrupt

PTBR

TCB

saves

processing

ready to resume

iret

Time
## Pintos: Context Switch – Scheduling

As the time progresses, the context switches between user and kernel mode. When a syscall or interrupt occurs, the kernel code is executed, and the context switch takes place. The kernel thread stack is restored, and the kernel thread is ready to resume processing.

### Kernel Code
- Start: Kernel code execution
- Context Switch: From user to kernel mode
- Syscall/Interrupt: Switch to kernel
- Execution: Kernel code execution
- Ready to Resume: Context switch back to user mode

### User Code
- Start: User code execution
- Context Switch: From kernel to user mode
- Execution: User code execution
- Ready to Resume: Context switch back to kernel mode

### TCB
- Thread Control Block: Stores thread information
- Saves: Saves user stack
- PTBR: Page Table Base Register
- TCB: Thread Control Block
- SS: Stack Segment
- CS: Code Segment
- EIP: Extended Instruction Pointer
- ESP: Extended Stack Pointer

### Pintos: switch.S
- Switch kernel threads
- Schedule
- Processing ready to resume

---

1/30/20
Kubiatowicz CS162 ©UCB Spring 2020
ThreadFork(): Create a New Thread

- **ThreadFork()** is a user-level procedure that creates a new thread and places it on ready queue

- **Arguments to ThreadFork()**
  - Pointer to application routine (**fcnPtr**)
  - Pointer to array of arguments (**fcnArgPtr**)
  - Size of stack to allocate

- **Implementation**
  - Sanity check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable)
How do we initialize TCB and Stack?

- Initialize Register fields of TCB
  - Stack pointer made to point at stack
  - PC return address ⇒ OS (asm) routine `ThreadRoot()`
  - Two arg registers (say rdi and rsi for x86) initialized to `fcnPtr` and `fcnArgPtr`, respectively

- Initialize stack data?
  - No. Important part of stack frame is in registers (ra)
  - Think of stack frame as just before body of `ThreadRoot()` really gets started

`ThreadRoot stub`

Initial Stack

Stack growth
How does Thread get started?

- Need to construct a new kernel thread that is ready to run when switch goes to it.
- Note that switch doesn’t know any difference between new or preexisting thread!
How does a thread get started?

- How do we make a new thread?
  - Setup TCB/kernel thread to point at new user stack and ThreadRoot code
  - Put pointers to start function and args in registers
  - This depends heavily on the calling convention (i.e. RISC-V vs x86)

- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

```c
SetupNewThread(tNew) {
    ...
    TCB[tNew].regs.sp = newStackPtr;
    TCB[tNew].regs.retpc = &ThreadRoot;
    TCB[tNew].regs.r0 = fcnPtr
    TCB[tNew].regs.r1 = fcnArgPtr
}
```

![Diagram](image.png)
What does `ThreadRoot()` look like?

- **ThreadRoot()** is the root for the thread routine:

  ```c
  ThreadRoot(fcnPTR, fcnArgPtr) {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    Call fcnPtr(fcnArgPtr);
    ThreadFinish();
  }
  ```

- **Startup Housekeeping**
  - Includes things like recording start time of thread
  - Other statistics

- **Stack will grow and shrink with execution of thread**

- **Final return from thread returns into ThreadRoot() which calls ThreadFinish()**
  - `ThreadFinish()` wake up sleeping threads
Administrivia

- anything?
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]
Kernel-Supported User Threads

THREAD T0

- executing
- interrupt or system call

- save state into PCB0
  - ...
  - ...
  - reload state from PCB0

THREAD T1

- idle
- executing

- save state into PCB1
  - ...
  - ...
  - reload state from PCB0

TCB0

TCB1

process P₀

operating system

process P₁
User-level Multithreading: \textit{pthreads}

- \texttt{int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start\_routine)(void*), void *arg);}  
  - thread is created executing \textit{start\_routine} with \textit{arg} as its sole argument. (return is implicit call to \textit{pthread\_exit})

- \texttt{void pthread\_exit(void *value\_ptr);}  
  - terminates and makes \textit{value\_ptr} available to any successful join

- \texttt{int pthread\_join(pthread_t thread, void **value\_ptr);}  
  - suspends execution of the calling thread until the target \textit{thread} terminates.
  - On return with a non-NULL \textit{value\_ptr} the value passed to \texttt{pthread\_exit()} by the terminating thread is made available in the location referenced by \textit{value\_ptr}.

\texttt{man pthread}

\url{https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html}
Little Example

How to tell if something is done?
Really done?
OK to reclaim its resources?

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

int common = 162;

void *threadfun(void *threadid)
{
    long tid = (long)threadid;
    printf("Thread %ld stack: %lx common: %lx (%d)\n", tid, 
           (unsigned long) &tid, (unsigned long) &common, common++);
    pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
    long t;
    int nthreads = 2;
    if (argc > 1) {
        nthreads = atoi(argv[1]);
    }

    pthread_t *threads = malloc(nthreads*sizeof(pthread_t));
    printf("Main stack: %lx common: %lx (%d)\n", 
           (unsigned long) &t, (unsigned long) &common, common);
    for(t=0; t<nthreads; t++)
    {
        int rc = pthread_create(&threads[t], NULL, threadfun, (void *)t);
        if (rc)
        {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    for(t=0; t<nthreads; t++)
    {
        pthread_join(threads[t], NULL);
    }
    pthread_exit(NULL); /* last thing in the main thread */
```
Main thread creates (forks) collection of sub-threads passing them args to work on, joins with them, collecting results.
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>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>
</tbody>
</table>
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x = x + 1;)</td>
<td>(x = x + 1;)</td>
<td>(x = x + 1)</td>
</tr>
<tr>
<td>(y = y + x;)</td>
<td>(y = y + x;)</td>
<td>(.............)</td>
</tr>
<tr>
<td>(z = x + 5y;)</td>
<td>(z = x + 5y;)</td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(.............)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(y = y + x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(z = x + 5y)</td>
</tr>
</tbody>
</table>
## Programmer vs. Processor View

<table>
<thead>
<tr>
<th>Programmer’s View</th>
<th>Possible Execution #1</th>
<th>Possible Execution #2</th>
<th>Possible Execution #3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x = x + 1;</td>
<td>x = x + 1;</td>
<td>x = x + 1</td>
<td>x = x + 1</td>
</tr>
<tr>
<td>y = y + x;</td>
<td>y = y + x;</td>
<td>...............</td>
<td>y = y + x</td>
</tr>
<tr>
<td>z = x + 5y;</td>
<td>z = x + 5y;</td>
<td>thread is suspended</td>
<td>thread is suspended</td>
</tr>
<tr>
<td></td>
<td></td>
<td>other thread(s) run</td>
<td>other thread(s) run</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td>y = y + x</td>
<td></td>
<td>thread is resumed</td>
<td>thread is resumed</td>
</tr>
<tr>
<td>z = x + 5y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Possible Executions

a) One execution

b) Another execution

c) Another execution
• Each Thread has a *Thread Control Block (TCB)*
  – Execution State: CPU registers, program counter (PC), pointer to stack (SP)
  – Scheduling info: state, priority, CPU time
  – Various Pointers (for implementing scheduling queues)
  – Pointer to enclosing process (PCB) – user threads
  – … (add stuff as you find a need)

• OS Keeps track of TCBs in “kernel memory”
  – In Array, or Linked List, or …
  – I/O state (file descriptors, network connections, etc)
Multithreaded Processes

- Process Control Block (PCBs) points to multiple Thread Control Blocks (TCBs):

- Switching threads within a block is a simple thread switch
- Switching threads across blocks requires changes to memory and I/O address tables
Multiprocessing vs Multiprogramming

• Remember Definitions:
  – Multiprocessing $\equiv$ Multiple CPUs
  – Multiprogramming $\equiv$ Multiple Jobs or Processes
  – Multithreading $\equiv$ Multiple threads per Process

• What does it mean to run two threads “concurrently”?
  – Scheduler is free to run threads in any order and interleaving: FIFO, Random, …
  – Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks
Correctness for systems with concurrent threads

- If dispatcher can schedule threads in any way, programs must work under all circumstances
  - Can you test for this?
  - How can you know if your program works?

- **Independent Threads:**
  - No state shared with other threads
  - Deterministic $\Rightarrow$ Input state determines results
  - Reproducible $\Rightarrow$ Can recreate Starting Conditions, I/O
  - Scheduling order doesn’t matter (if `switch()` works!!!)

- **Cooperating Threads:**
  - Shared State between multiple threads
  - Non-deterministic
  - Non-reproducible

- Non-deterministic and Non-reproducible means that bugs can be intermittent
  - Sometimes called “Heisenbugs”
Heisenberg
Interactions Complicate Debugging

• Is any program truly independent?
  – Every process shares the file system, OS resources, network, etc
  – Extreme example: buggy device driver causes thread A to crash “independent thread” B

• You probably don’t realize how much you depend on reproducibility:
  – Example: Evil C compiler
    » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  – Example: Debugging statements can overrun stack

• Non-deterministic errors are really difficult to find
  – Example: Memory layout of kernel+user programs
    » depends on scheduling, which depends on timer/other things
    » Original UNIX had a bunch of non-deterministic errors
  – Example: Something which does interesting I/O
    » User typing of letters used to help generate secure keys
Why allow cooperating threads?

- People cooperate; computers help/enhance people’s lives, so computers must cooperate
  - By analogy, the non-reproducibility/non-determinism of people is a notable problem for “carefully laid plans”
- Advantage 1: Share resources
  - One computer, many users
  - One bank balance, many ATMs
    » What if ATMs were only updated at night?
  - Embedded systems (robot control: coordinate arm & hand)
- Advantage 2: Speedup
  - Overlap I/O and computation
    » Many different file systems do read-ahead
  - Multiprocessors – chop up program into parallel pieces
- Advantage 3: Modularity
  - More important than you might think
  - Chop large problem up into simpler pieces
    » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    » Makes system easier to extend
High-level Example: Web Server

- Server must handle many requests
- Non-cooperating version:
  ```
  serverLoop() {
      con = AcceptCon();
      ProcessFork(ServiceWebPage(), con);
  }
  ```
- What are some disadvantages of this technique?
Threaded Web Server

- Now, use a single process
- Multithreaded (cooperating) version:
  
  ```
  serverLoop() {
    connection = AcceptCon();
    ThreadFork(ServiceWebPage(),connection);
  }
  ```

- Looks almost the same, but has many advantages:
  - Can share file caches kept in memory, results of CGI scripts, other things
  - Threads are much cheaper to create than processes, so this has a lower per-request overhead
- Question: would a user-level (say one-to-many) thread package make sense here?
  - When one request blocks on disk, all block…
- What about Denial of Service attacks or digg / Slash-dot effects?
Thread Pools

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming

```java
master() {
    allocThreads(worker,queue);
    while (TRUE) {
        con=AcceptCon();
        Enqueue(queue,con);
        wakeUp(queue);
    }
}

worker(queue) {
    while (TRUE) {
        con=Dequeue(queue);
        if (con==null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```
• ATM server problem:
  – Service a set of requests
  – Do so without corrupting database
  – Don’t hand out too much money
ATM bank server example

• Suppose we wanted to implement a server process to handle requests from an ATM network:

```c
BankServer() {
    while (TRUE) {
        ReceiveRequest(&op, &acctId, &amount);
        ProcessRequest(op, acctId, amount);
    }
}
```

```c
ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ...
}
```

```c
Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
```

• How could we speed this up?
  – More than one request being processed at once
  – Event driven (overlap computation and I/O)
  – Multiple threads (multi-proc, or overlap comp and I/O)
Event Driven Version of ATM server

• Suppose we only had one CPU
  – Still like to overlap I/O with computation
  – Without threads, we would have to rewrite in event-driven style

• Example

```java
BankServer() {
    while (TRUE) {
        event = WaitForNextEvent();
        if (event == ATMRequest)
            StartOnRequest();
        else if (event == AcctAvail)
            ContinueRequest();
        else if (event == AcctStored)
            FinishRequest();
    }
}
```

– What if we missed a blocking I/O step?
– What if we have to split code into hundreds of pieces which could be blocking?
– This technique is used for graphical programming
Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments
  - One thread per request
- Requests proceeds to completion, blocking as required:

  ```c
  Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
  }
  ```

- Unfortunately, shared state can get corrupted:

  Thread 1
  ```c
  load r1, acct->balance
  add r1, amount1
  store r1, acct->balance
  ```

  Thread 2
  ```c
  load r1, acct->balance
  add r1, amount2
  store r1, acct->balance
  ```
Problem is at the Lowest Level

• Most of the time, threads are working on separate data, so scheduling doesn’t matter:

  Thread A          Thread B
  x = 1;            y = 2;

• However, what about (Initially, y = 12):

  Thread A          Thread B
  x = 1;            y = 2;
  x = y+1;          y = y*2;

  – What are the possible values of x?

• Or, what are the possible values of x below?

  Thread A          Thread B
  x = 1;            x = 2;

  – X could be 1 or 2 (non-deterministic!)

  – Could even be 3 for serial processors:

    » Thread A writes 0001, B writes 0010 → scheduling order ABABABABBA yields 3!
Atomic Operations

• To understand a concurrent program, we need to know what the underlying indivisible operations are!

• Atomic Operation: an operation that always runs to completion or not at all
  – It is *indivisible*: it cannot be stopped in the middle and state cannot be modified by someone else in the middle
  – Fundamental building block – if no atomic operations, then have no way for threads to work together

• On most machines, memory references and assignments (i.e. loads and stores) of words are atomic
  – Consequently – weird example that produces “3” on previous slide can’t happen

• Many instructions are not atomic
  – Double-precision floating point store often not atomic
  – VAX and IBM 360 had an instruction to copy a whole array
Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

  Thread A
  
i = 0;
  while (i < 10)
      i = i + 1;
  printf("A wins!");

  Thread B
  
i = 0;
  while (i > -10)
      i = i - 1;
  printf("B wins!");

- Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic

- Who wins? Could be either

- Is it guaranteed that someone wins? Why or why not?

- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?
Hand Simulation Multiprocessor Example

- Inner loop looks like this:

  Thread A                           Thread B
  r1=0  load r1, M[i]                r1=0  load r1, M[i]
  r1=1  add  r1, r1, 1               r1=-1  sub r1, r1, 1
  M[i]=1  store r1, M[i]             M[i]=-1  store r1, M[i]

- Hand Simulation:
  - And we’re off. A gets off to an early start
  - B says “hmph, better go fast” and tries really hard
  - A goes ahead and writes “1”
  - B goes and writes “-1”
  - A says “HUH??! I could have sworn I put a 1 there”

- Could this happen on a uniprocessor? With Hyperthreads?
  - Yes! Unlikely, but if you are depending on it not happening, it will and your system will break…
Correctness Requirements

• Threaded programs must work for all interleavings of thread instruction sequences
  – Cooperating threads inherently non-deterministic and non-reproducible
  – Really hard to debug unless carefully designed!

• Example: Therac-25
  – Machine for radiation therapy
    » Software control of electron accelerator and electron beam/Xray production
    » Software control of dosage
  – Software errors caused the death of several patients
    » A series of race conditions on shared variables and poor software design
    » “They determined that data entry speed during editing was the key factor in producing the error condition: If the prescription data was edited at a fast pace, the overdose occurred.”
Motivating Example: “Too Much Milk”

• Great thing about OS’s – analogy between problems in OS and problems in real life
  – Help you understand real life problems better
  – But, computers are much stupider than people

• Example: People need to coordinate:

<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></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>Look in Fridge. Out of 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>
Definitions

• **Synchronization**: using atomic operations to ensure cooperation between threads
  – For now, only loads and stores are atomic
  – We are going to show that its hard to build anything useful with only reads and writes

• **Mutual Exclusion**: ensuring that only one thread does a particular thing at a time
  – One thread *excludes* the other while doing its task

• **Critical Section**: piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code
  – Critical section is the result of mutual exclusion
  – Critical section and mutual exclusion are two ways of describing the same thing
More Definitions

• **Lock**: prevents someone from doing something
  – Lock before entering critical section and before accessing shared data
  – Unlock when leaving, after accessing shared data
  – Wait if locked
    » Important idea: all synchronization involves waiting

• For example: fix the milk problem by putting a key on the refrigerator
  – Lock it and take key if you are going to go buy milk
  – Fixes too much: roommate angry if only wants OJ

  – Of Course – We don’t know how to make a lock yet
Too Much Milk: Correctness Properties

- Need to be careful about correctness of concurrent programs, since non-deterministic
  - Impulse is to start coding first, then when it doesn't work, pull hair out
  - Instead, think first, then code
  - Always write down behavior first

- What are the correctness properties for the “Too much milk” problem???
  - Never more than one person buys
  - Someone buys if needed

- Restrict ourselves to use only atomic load and store operations as building blocks
Too Much Milk: Solution #1

• Use a note to avoid buying too much milk:
  – Leave a note before buying (kind of “lock”)
  – Remove note after buying (kind of “unlock”)
  – Don’t buy if note (wait)

• Suppose a computer tries this (remember, only memory read/write are atomic):

```java
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
        remove note;
    }
}
```
Too Much Milk: Solution #1

• Use a note to avoid buying too much milk:
  – Leave a note before buying (kind of “lock”)
  – Remove note after buying (kind of “unlock”)
  – Don’t buy if note (wait)

• Suppose a computer tries this (remember, only memory read/write are atomic):

  Thread A
  if (noMilk) {
      if (noNote) {
          leave Note;
          buy Milk;
          remove Note;
      }
  }

  Thread B
  if (noMilk) {
      if (noNote) {
          leave Note;
          buy Milk;
          remove Note;
      }
  }
Too Much Milk: Solution #1

• Use a note to avoid buying too much milk:
  – Leave a note before buying (kind of “lock”)
  – Remove note after buying (kind of “unlock”)
  – Don’t buy if note (wait)

• Suppose a computer tries this (remember, only memory read/write are atomic):

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

• Result?
  – Still too much milk but only occasionally!
  – Thread can get context switched after checking milk and note but before buying milk!

• Solution makes problem worse since fails intermittently
  – Makes it really hard to debug…
  – Must work despite what the dispatcher does!
Too Much Milk: Solution #1 ½

• Clearly the Note is not quite blocking enough
  – Let’s try to fix this by placing note first
• Another try at previous solution:

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

• What happens here?
  – Well, with human, probably nothing bad
  – With computer: no one ever buys milk
Too Much Milk Solution #2

• How about labeled notes?
  – Now we can leave note before checking

• Algorithm looks like this:

  Thread A
  
  leave note A;
  if (noNote B) {
    if (noMilk) {
      buy Milk;
    }
  }

  remove note A;

  Thread B
  
  leave note B;
  if (noNoteA) {
    if (noMilk) {
      buy Milk;
    }
  }

  remove note B;

• Does this work?

• Possible for neither thread to buy milk
  – Context switches at exactly the wrong times can lead each to think that the other is going to buy

• Really insidious:
  – Extremely unlikely this would happen, but will at worse possible time
  – Probably something like this in UNIX
Too Much Milk Solution #2: problem!

• I’m not getting milk, You’re getting milk
• This kind of lockup is called “starvation!”
Too Much Milk Solution #3

• Here is a possible two-note solution:

```java
Thread A
leave note A;
while (note B) {\X
do nothing;
    }
if (noMilk) {
    buy milk;
} \Y
remove note A;

Thread B
leave note B;
if (noNote A) {\Y
    if (noMilk) {
        buy milk;
    }
} \X
remove note B;
```

• Does this work? Yes. Both can guarantee that:
  – It is safe to buy, or
  – Other will buy, ok to quit

• At X:
  – If no note B, safe for A to buy,
  – Otherwise wait to find out what will happen

• At Y:
  – If no note A, safe for B to buy
  – Otherwise, A is either buying or waiting for B to quit
Case 1

- "leave note A" happens before "if (noNote A)"

```java
leave note A;
while (note B) {
    do nothing;
}

if (noMilk) {
    buy milk;
}
remove note A;

leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
}
remove note B;

if (noMilk) {
    buy milk;
}
remove note A;
```

"leave note A" happens before "if (noNote A)"
Case 1

- "leave note A" happens before "if (noNote A)"

```c
leave note A;
while (note B) {
    do nothing;
};
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
}
remove note B;
if (noMilk) {
    buy milk;
}
remove note A;
```
Case I

- “leave note A” happens before “if (noNote A)”

```
leave note A;
while (note B) {
  do nothing;
};

if (noMilk) {
  buy milk;
}
remove note A;
```

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```

Wait for note B to be removed happened before "leave note A" happens before "if (noNote A)"
Case 2

• "if (noNote A)" happens before "leave note A"

```c
leave note A;
while (note B) { \X
    do nothing;
};

if (noMilk) {
    buy milk;
}
remove note A;

leave note B;
if (noNote A) { \X
    if (noMilk) {
        buy milk;
    }
} \X
remove note B;
```

"if (noNote A)" happens before "leave note A"
Case 2

• “if (noNote A)” happens before “leave note A”

```plaintext
leave note A;
while (note B) {
    do nothing;
};

if (noMilk) {
    buy milk;
}
remove note A;
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
} remove note B;
```

“if (noNote A)” happens before “leave note A”
Case 2

- "if (noNote A)" happens before "leave note A"

```
leave note A;
while (note B) {
  do nothing;
}

if (noMilk) {
  buy milk;
}
remove note A;
```

```
leave note B;
if (noNote A) {
  if (noMilk) {
    buy milk;
  }
}
remove note B;
```
Solution #3 discussion

• Our solution protects a single “Critical-Section” piece of code for each thread:

```java
if (noMilk) {
    buy milk;
}
```

• Solution #3 works, but it’s really unsatisfactory
  – Really complex – even for this simple an example
    » Hard to convince yourself that this really works
  – A’s code is different from B’s – what if lots of threads?
    » Code would have to be slightly different for each thread
  – While A is waiting, it is consuming CPU time
    » This is called “busy-waiting”

• There’s a better way
  – Have hardware provide higher-level primitives than atomic load & store
  – Build even higher-level programming abstractions on this hardware support
Too Much Milk: Solution #4

• Suppose we have some sort of implementation of a lock
  \texttt{lock.Acquire()} – wait until lock is free, then grab
  \texttt{lock.Release()} – Unlock, waking up anyone waiting
  – These must be atomic operations – if two threads are waiting for the lock and both see it’s free, only one succeeds to grab the lock

• Then, our milk problem is easy:
  
  \begin{verbatim}
  milklock.Acquire();
  if (nomilk)
    buy milk;
  milklock.Release();
  \end{verbatim}

• Once again, section of code between \texttt{Acquire()} and \texttt{Release()} called a “Critical Section”
How to Implement Locks?

• **Lock**: prevents someone from doing something
  – Lock before entering critical section and before accessing shared data
  – Unlock when leaving, after accessing shared data
  – Wait if locked
    » Important idea: all synchronization involves waiting
    » Should sleep if waiting for a long time

• **Atomic Load/Store**: get solution like Milk #3
  – Pretty complex and error prone

• **Hardware Lock instruction**
  – Is this a good idea?
  – What about putting a task to sleep?
    » What is the interface between the hardware and scheduler?
  – Complexity?
    » Done in the Intel 432
    » Each feature makes HW more complex and slow
How can we build multi-instruction atomic operations?
- Recall: dispatcher gets control in two ways.
  » Internal: Thread does something to relinquish the CPU
  » External: Interrupts cause dispatcher to take CPU
- On a uniprocessor, can avoid context-switching by:
  » Avoiding internal events
  » Preventing external events by disabling interrupts

Consequently, naïve implementation of locks:
```
LockAcquire { disable Ints; }
LockRelease { enable Ints; }
```

Problems with this approach:
- Can’t let user do this! Consider following:
  ```
  LockAcquire();
  While(TRUE) {;}
  ```
- Real-Time system—no guarantees on timing!
  » Critical Sections might be arbitrarily long
- What happens with I/O or other important events?
  » “Reactor about to meltdown. Help?”
Better Implementation of Locks by Disabling Interrupts

- Key idea: maintain a lock variable and impose mutual exclusion only during operations on that variable

```c
int value = FREE;

Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        Go to sleep();
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone on wait queue) {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
    }
    enable interrupts;
}
```
Where are we going with synchronization?

<table>
<thead>
<tr>
<th>Programs</th>
<th>Shared Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level API</td>
<td>Locks  Semaphores  Monitors  Send/Receive</td>
</tr>
<tr>
<td>Hardware</td>
<td>Load/Store  Disable Ints  Test&amp;Set  Compare&amp;Swap</td>
</tr>
</tbody>
</table>

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level
Summary

• Concurrent threads are a very useful abstraction
  – Allow transparent overlapping of computation and I/O
  – Allow use of parallel processing when available

• Concurrent threads introduce problems when accessing shared data
  – Programs must be insensitive to arbitrary interleavings
  – Without careful design, shared variables can become completely inconsistent

• Important concept: Atomic Operations
  – An operation that runs to completion or not at all
  – These are the primitives on which to construct various synchronization primitives