Recall: 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()

Recall: 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

Recall: Multiprocessing vs Multiprogramming

- Remember Definitions:
  - Multiprocessing ≡ Multiple CPUs
  - Multiprogramming ≡ Multiple Jobs or Processes
  - Multithreading ≡ 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
Recall: Process

(UNIX) Process

Sequential stream of instructions

A(int tmp) {
    if (tmp<2)
        B();
        printf(tmp);
    B();
    C();
    C() {
        A(2);
    }
    A(1);
}

Recall: Processes vs. Threads

1 thread at a time

Recall: Processes vs. Threads (Multi-Core)

8 threads at a time

Recall: Hyper-Threading

Switch overhead between hardware-threads: very-low (done in hardware)

Contention for ALUs/FPUs may hurt performance
**Kernel versus User-Mode Threads**

- We have been talking about kernel threads
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things

- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule

- Lighter weight option: User level Threads

**User-Mode Threads**

- Lighter weight option:
  - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
  - Cheap

- Downside of user threads:
  - When one thread blocks on I/O, all threads block
  - Kernel cannot adjust scheduling among all threads
  - Option: *Scheduler Activations*
    » Have kernel inform user level when thread blocks...

---

**Some Threading Models**

- Simple One-to-One Threading Model (PINTOS!)

- Many-to-One

- Many-to-Many

**Classification**

<table>
<thead>
<tr>
<th># threads Per AS</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
<td></td>
</tr>
<tr>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)</td>
<td>Mach, OS/2, Linux Windows 10 Win NT to XP, Solaris, HP-UX, OS X</td>
<td></td>
</tr>
</tbody>
</table>

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space
Recall: ATM Bank Server

- ATM server problem:
  - Service a set of requests
  - Do so without corrupting database
  - Don’t hand out too much money

Recall: 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)

Recall: 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               Thread 2
load r1, acct->balance load r1, acct->balance
add r1, amount1        add r1, amount2
store r1, acct->balance store r1, acct->balance
```

Administrivia

- I’m back!
  - Sorry, I’ve been sick for a while
  - Will try to resume office hours (M/Th 1:00) on Thursday
  - Thanks for the well-wishes on Piazza!
- Should have formed your groups and be working on Project 1!
  - Including the part which is to be done individually
- Should be attending section according to your assignments
- Don’t miss the brief, weekly quizzes
  - They help us to evaluate how people are doing in the class
- Midterm I: Thursday 2/27
  - All material up to that Tuesday is fair game
  - We will have a review session prior to the day (stay tuned!)
Recall: 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
- 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

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 it's 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):
  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 #1½

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

```plaintext
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:

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

Thread B
leave note B;
if (noNote A) {
  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 worst 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:

```plaintext
Thread A
leave note A;
while (note B) {
  do nothing;
  if (noNote A) {
    if (noMilk) {
      buy milk;
    }
  }
  remove note A;

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

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

Case 1

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

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

Case 1

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

```plaintext
leave note A;
while (note B) {
    do nothing;
}
if (noMilk) {
    buy milk;
}
remove note B;
```

Case 2

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

```plaintext
leave note A;
while (note B) {
    if (noNote A) {
        if (noMilk) {
            buy milk;
        }
        remove note B;
    }
}
if (noMilk) {
    buy milk;
}
remove note A;
```
Case 2

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

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

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

Wait for note B to be removed

Solution #3 discussion

• Our solution protects a single “Critical-Section” piece of code for each thread:
  
  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
  – lock.Acquire() — wait until lock is free, then grab
  – 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:
  
  milklock.Acquire();
  if (nomilk)
      buy milk;
  milklock.Release();

• Once again, section of code between Acquire() and Release() called a “Critical Section”

• Of course, you can make this even simpler: suppose you are out of ice cream instead of milk
  – Skip the test since you always need more ice cream ;-)

2/11/20 Kubiatowicz CS162 ©UCB Spring 2020
Where are we going with synchronization?

- 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

Little Example: Stack of Strings (SoS)

```c
struct str_lst_elem {
    char *str;
    struct str_lst_elem *next;
};

struct str_lst {
    struct str_lst_elem *head;
};

void str_lst_init(struct str_lst *lst) {
    lst->head = NULL;
    pthread_mutex_init(&lst->lock, NULL);
}

void str_lst_push(struct str_lst *lst, char *str) {
    struct str_lst_elem *new_elem = malloc(sizeof(struct str_lst_elem));
    new_elem->str = str;
    new_elem->next = lst->head;
    lst->head = new_elem;
}

char *str_lst_pop(struct str_lst *lst) {
    char *topval;
    pthread_mutex_lock (&lst->lock);
    struct str_lst_elem *top = lst->head;
    if (!top) {
        topval = NULL;
    } else {
        topval = top->str;
        lst->head = top->next;
    }
    pthread_mutex_unlock (&lst->lock);
    return topval;
}
```

Thread Safe: Stack of Strings

```c
struct str_lst_elem {
    char *str;
    struct str_lst_elem *next;
};

struct str_lst {
    struct str_lst_elem *head;
    pthread_mutex_t lock;
};

void str_lst_init(struct str_lst *lst) {
    lst->head = NULL;
    pthread_mutex_init(&lst->lock, NULL);
    pthread_mutex_lock(&lst->lock, NULL);
}
```
Thread safe: SoS (cont)

```c
void str_lst_push(struct str_lst *lst, char *str) {
    struct str_lst_elem *new_elem = malloc(sizeof(struct str_lst_elem));
    new_elem->str = str;
    pthread_mutex_lock (&lst->lock);
    new_elem->next = lst->head;
    lst->head = new_elem;
    pthread_mutex_unlock (&lst->lock);
};
```

```c
char* str_lst_pop(struct str_lst *lst) {
    char *topval;
    pthread_mutex_lock (&lst->lock);
    struct str_lst_elem *top = lst->head;
    if (!top) {
        topval = NULL;
    } else {
        topval = top->str;
        lst->head = top->next;
    }
    pthread_mutex_unlock (&lst->lock);
    return topval;
};
```

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

Naïve use of Interrupt Enable/Disable

- 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 (although virtual memory tricky)
    » Preventing external events by disabling interrupts
- Consequently, naïve implementation of locks:
  ```c
  int value = FREE;
  ```
  ```c
  Acquire() {
      disable interrupts;
      if (value == BUSY) {
          put thread on wait queue;
          Go to sleep();
          // Enable interrupts?
      } else {
          value = BUSY;
      }
      enable interrupts;
  }
  ```
  ```c
  Release() {
      disable interrupts;
      if (anyone on wait queue) {
          take thread off wait queue
          Place on ready queue;
      } else {
          value = FREE;
      }
      enable interrupts;
  }
  ```
- Problems with this approach:
  - Can’t let user do this! Consider following:
    ```c
    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
New Lock Implementation: Discussion

- Why do we need to disable interrupts at all?
  - Avoid interruption between checking and setting lock value
  - Otherwise two threads could think that they both have lock

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

- Note: unlike previous solution, the critical section (inside Acquire()) is very short
  - User of lock can take as long as they like in their own critical section: doesn't impact global machine behavior
  - Critical interrupts taken in time!

Interrupt Re-enable in Going to Sleep

- What about re-enabling ints when going to sleep?
  ```c
  Acquire() {
    disable interrupts;
    if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
    } else {
      value = BUSY;
    }
    enable interrupts;
  }
  ```

- Before Putting thread on the wait queue?
  - Release can check the queue and not wake up thread
Interrupt Re-enable in Going to Sleep

• What about re-enabling ints when going to sleep?
  
  ```c
  Acquire() {
    disable interrupts;
    if (value == BUSY) {
      put thread on wait queue;
      Go to sleep();
    } else {
      value = BUSY;
      enable interrupts;
    }
  }
  
  Enable Position
  ```

• Before Putting thread on the wait queue?
  – Release can check the queue and not wake up thread

• After putting the thread on the wait queue

How to Re-enable After Sleep()?

• In scheduler, since interrupts are disabled when you call sleep:
  – Responsibility of the next thread to re-enable ints
  – When the sleeping thread wakes up, returns to acquire and re-enables interrupts

    ```
    Thread A
    .
    disable ints
    sleep
    .
    context switch
    sleep
    return enable ints
    .
    
    Thread B
    .
    disable ints
    sleep
    .
    context switch
    sleep
    return enable ints
    .
    ```
Atomic Read-Modify-Write Instructions

- Problems with previous solution:
  - Can’t give lock implementation to users
  - Doesn’t work well on multiprocessor
    » Disabling interrupts on all processors requires messages and would be very time consuming

- Alternative: atomic instruction sequences
  - These instructions read a value and write a new value atomically
  - Hardware is responsible for implementing this correctly
    » on both uniprocessors (not too hard)
    » and multiprocessors (requires help from cache coherence protocol)
  - Unlike disabling interrupts, can be used on both uniprocessors and multiprocessors

Examples of Read-Modify-Write

```
• test&set (&address) { /* most architectures */
  result = M[address];        // return result from "address" and
  M[address] = 1;             // set value at "address" to 1
  return result;
}
• swap (address, register) { /* x86 */
  temp = M[address];          // swap register's value to
  M[address] = register;      // value at "address"
  register = temp;
}
• compare&swap (&address, reg1, reg2) { /* 68000 */
  if (reg1 == M[address]) {   // If memory still == reg1,
    M[address] = reg2;      // then put reg2 => memory
    return success;
  } else {                    // Otherwise do not change memory
    return failure;
  }
}
• load-linked&store-conditional(&address) { /* R4000, alpha */
  loop:
  ll r1, M[address];movi r2, 1;// Can do arbitrary computation
  sc r2, M[address];beqz r2, loop;
}
```

Using of Compare&Swap for queues

```
• compare&swap (&address, reg1, reg2) { /* 68000 */
  if (reg1 == M[address]) {
    M[address] = reg2;
    return success;
  } else {
    return failure;
  }
}
```

Here is an atomic add to linked-list function:
```
addToQueue(&object) {
  do { // repeat until no conflict
    ld r1, M[root] // Get ptr to current head
    st r1, M[object] // Save link in new object
  } until (compare&swap(&root,r1,object));
}
```

Implementing Locks with test&set

• Another flawed, but simple solution:

  ```
  int value = 0;  // Free
  Acquire() {
    while (test&set(value)); // while busy
  }
  Release() {
    value = 0;
  }
  ```

• Simple explanation:
  - If lock is free, test&set reads 0 and sets value=1, so lock is now busy. It returns 0 so while exits.
  - If lock is busy, test&set reads 1 and sets value=1 (no change) It returns 1, so while loop continues.
  - When we set value = 0, someone else can get lock.

• Busy-Waiting: thread consumes cycles while waiting
  - For multiprocessors: every test&set() is a write, which makes value ping-pong around in cache (using lots of network BW)
Problem: Busy-Waiting for Lock

• Positives for this solution
  – Machine can receive interrupts
  – User code can use this lock
  – Works on a multiprocessor

• Negatives
  – This is very inefficient as thread will consume cycles waiting
    – Waiting thread may take cycles away from thread holding lock
      (no one wins!)
  – Priority Inversion: If busy-waiting thread has higher priority than thread holding lock ⇒ no progress!
  
  Priority Inversion problem with original Martian rover

  • For semaphores and monitors, waiting thread may wait for an arbitrary long time!
    – Thus even if busy-waiting was OK for locks, definitely not ok for other primitives
    – Homework/exam solutions should avoid busy-waiting!

Multiprocessor Spin Locks: test&test&set

• A better solution for multiprocessors:

```c
int mylock = 0; // Free

Acquire() {
    do {
        while(mylock);   // Wait until might be free
    } while(test&set(&mylock)); // exit if get lock
}

Release() {
    mylock = 0;
}
```

• Simple explanation:
  – Wait until lock might be free (only reading – stays in cache)
  – Then, try to grab lock with test&set
  – Repeat if fail to actually get lock

• Issues with this solution:
  – Busy-Waiting: thread still consumes cycles while waiting
    » However, it does not impact other processors!

Better Locks using test&set

• Can we build test&set locks without busy-waiting?
  – Can’t entirely, but can minimize!
    – Idea: only busy-wait to atomically check lock value

```c
int guard = 0;
int value = FREE;

Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == BUSY) {
        put thread on wait queue;
        go to sleep() & guard = 0;
    } else {
        value = BUSY;
        guard = 0;
    }
}

Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait queue
        Place on ready queue;
    } else {
        value = FREE;
        guard = 0;
    }
}
```

• Note: sleep has to be sure to reset the guard variable
  – Why can’t we do it just before or just after the sleep?

Recall: Locks using Interrupts vs. test&set

Compare to “disable interrupt” solution

```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;
}
```

Basically we replaced:
  – disable interrupts → while (test&set(guard));
  – enable interrupts → guard = 0;
Recap: Locks using interrupts

```c
int value = 0;
Acquire() {
    // Short busy-wait time
disable interrupts;
    if (value == 1) {
        put thread on wait-queue;
go to sleep() & Enab Ints
    } else {
        value = 1;
        enable interrupts;
    }
}

Release() {
    // Short busy-wait time
disable interrupts;
    if anyone on wait queue {
        take thread off wait-queue
Place on ready queue;
    } else {
        value = 0;
        enable interrupts;
    }
}
```

Lock.Acquire();

…critical section;…

Lock.Release();

If one thread in critical
section, no other activity (including OS) can run!

Recap: Locks using test & set

```c
int guard = 0;
int value = 0;
Acquire() {
    // Short busy-wait time
while(test&set(guard));
if (value == 1) {
    put thread on wait-queue;
go to sleep() & guard = 0;
} else {
    value = 1;
    guard = 0;
}
}

Release() {
    // Short busy-wait time
while (test&set(guard));
if anyone on wait queue {
    take thread off wait-queue
Place on ready queue;
} else {
    value = 0;
    guard = 0;
}
}
```

Lock.Acquire();

…critical section;…

Lock.Release();

Threads waiting to
enter critical section
busy-wait

Higher-level Primitives than Locks

- Goal of last couple of lectures:
  - What is right abstraction for synchronizing threads that
    share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find
    bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s
    (10 years after started), systems running UNIX would crash
    every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple
  concurrent activities that are using shared state
  - This lecture and the next presents some ways of
    structuring sharing

Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value
  and supports the following two operations:
  - P(): an atomic operation that waits for semaphore to become
    positive, then decrements it by 1
    » Think of this as the wait() operation
  - V(): an atomic operation that increments the semaphore by 1,
    waking up a waiting P, if any
    » This of this as the signal() operation
- Note that P() stands for “proberen” (to test) and V() stands for
  “verhogen” (to increment) in Dutch
**Semaphores Like Integers Except**

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can’t read or write value, except to set it initially
  - Operations must be atomic
    » Two P’s together can’t decrement value below zero
    » Similarly, thread going to sleep in P won’t miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:

  ![Semaphore Diagram](image)

**Two Uses of Semaphores**

**Mutual Exclusion (initial value = 1)**
- Also called “Binary Semaphore”.
- Can be used for mutual exclusion:
  ```java
  semaphore.P();
  // Critical section goes here
  semaphore.V();
  ```

**Scheduling Constraints (initial value = 0)**
- Allow thread 1 to wait for a signal from thread 2
  - thread 2 schedules thread 1 when a given event occurs
- Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:
  ```java
  Initial value of semaphore = 0
  ThreadJoin {
    semaphore.P();
  }
  ThreadFinish {
    semaphore.V();
  }
  ```

**Producer-Consumer with a Bounded Buffer**

- Problem Definition
  - Producer puts things into a shared buffer
  - Consumer takes them out
  - Need synchronization to coordinate producer/consumer
- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty
- Example 1: GCC compiler
  - `cpp | cc1 | cc2 | as | ld`
- Example 2: Coke machine
  - Producer can put limited number of Cokes in machine
  - Consumer can’t take Cokes out if machine is empty

**Correctness constraints for solution**

- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb:
  **Use a separate semaphore for each constraint**
  - Semaphore fullBuffers; // consumer’s constraint
  - Semaphore emptyBuffers; // producer’s constraint
  - Semaphore mutex; // mutual exclusion
Full Solution to Bounded Buffer

Semaphore fullSlots = 0; // Initially, no coke
Semaphore emptySlots = bufSize; // Initially, num empty slots
Semaphore mutex = 1; // No one using machine

Producer(item) {
    emptySlots.P(); // Wait until space
    mutex.P(); // Wait until machine free
    Enqueue(item);
    mutex.V();
    fullSlots.V(); // Tell consumers there is more coke
}

Consumer() {
    fullSlots.P(); // Check if there's a coke
    mutex.P(); // Wait until machine free
    item = Dequeue();
    mutex.V();
    emptySlots.V(); // tell producer need more
    return item;
}

Discussion about Solution

- Why asymmetry?
  - Producer does: emptyBuffer.P(), fullBuffer.V()
  - Consumer does: fullBuffer.P(), emptyBuffer.V()

- Is order of P's important?
  - Yes! Can cause deadlock

- Is order of V's important?
  - No, except that it might affect scheduling efficiency

- What if we have 2 producers or 2 consumers?

  Decrease # of empty slots
  Increase # of occupied slots

  Decrease # of empty slots
  Increase # of occupied slots

Motivation for Monitors and Condition Variables

- Semaphores are a huge step up; just think of trying to do the bounded buffer with only loads and stores:
  - Problem is that semaphores are dual purpose:
    » They are used for both mutex and scheduling constraints
    » Example: the fact that flipping of P's in bounded buffer gives deadlock is not immediately obvious. How do you prove correctness to someone?

- Cleaner idea: Use locks for mutual exclusion and condition variables for scheduling constraints

- Definition: Monitor: a lock and zero or more condition variables for managing concurrent access to shared data
  - Some languages like Java provide this natively
  - Most others use actual locks and condition variables

Monitor with Condition Variables

- Lock: the lock provides mutual exclusion to shared data
  - Always acquire before accessing shared data structure
  - Always release after finishing with shared data
  - Lock initially free

- Condition Variable: a queue of threads waiting for something inside a critical section
  - Key idea: make it possible to go to sleep inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can’t wait inside critical section
Simple Monitor Example (version 1)

- Here is an (infinite) synchronized queue

```java
Lock lock;
Queue queue;

AddToQueue(item) {
    lock.Acquire(); // Lock shared data
    queue.enqueue(item); // Add item
    lock.Release(); // Release Lock
}

RemoveFromQueue() {
    lock.Acquire(); // Lock shared data
    item = queue.dequeue(); // Get next item or null
    lock.Release(); // Release Lock
    return(item); // Might return null
}

- Not very interesting use of “Monitor”
  - It only uses a lock with no condition variables
  - Cannot put consumer to sleep if no work!
```

Condition Variables

- How do we change the RemoveFromQueue() routine to wait until something is on the queue?
  - Could do this by keeping a count of the number of things on the queue (with semaphores), but error prone

- **Condition Variable**: a queue of threads waiting for something inside a critical section
  - Key idea: allow sleeping inside critical section by atomically releasing lock at time we go to sleep
  - Contrast to semaphores: Can’t wait inside critical section

- Operations:
  - `Wait(&lock)`: Atomically release lock and go to sleep. Re-acquire lock later, before returning.
  - `Signal()`: Wake up one waiter, if any
  - `Broadcast()`: Wake up all waiters

- Rule: Must hold lock when doing condition variable ops!
  - In Birrell paper, he says can perform signal() outside of lock – IGNORE HIM (this is only an optimization)

Complete Monitor Example (with condition variable)

- Here is an (infinite) synchronized queue

```java
Lock lock;
Condition dataready;
Queue queue;

AddToQueue(item) {
    lock.Acquire(); // Get Lock
    queue.enqueue(item); // Add item
    dataready.signal(); // Signal any waiters
    lock.Release(); // Release Lock
}

RemoveFromQueue() {
    lock.Acquire(); // Get Lock
    while (queue.isEmpty()) {
        dataready.wait(&lock); // If nothing, sleep
    }
    item = queue.dequeue(); // Get next item
    lock.Release(); // Release Lock
    return(item);
}
```

Mesa vs. Hoare monitors

- Need to be careful about precise definition of signal and wait. Consider a piece of our dequeue code:

```java
while (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

- Why didn’t we do this?

```java
if (queue.isEmpty()) {
    dataready.wait(&lock); // If nothing, sleep
}
item = queue.dequeue(); // Get next item
```

- Answer: depends on the type of scheduling
  - **Hoare-style** (most textbooks):
    - Signaler gives lock, CPU to waiter; waiter runs immediately
    - Waiter gives up lock, processor back to signaler when it exits critical section or if it waits again
  - **Mesa-style** (most real operating systems):
    - Signaler keeps lock and processor
    - Waiter placed on ready queue with no special priority

- Practically, need to check condition again after wait
Summary (1/2)

• 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
• Talked about hardware atomicity primitives:
  – Disabling of Interrupts, test&set, swap, compare&swap, load-locked & store-conditional
• Showed several constructions of Locks
  – Must be very careful not to waste/tie up machine resources
    » Shouldn’t disable interrupts for long
    » Shouldn’t spin wait for long
  – Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable

Summary (2/2)

• Semaphores: Like integers with restricted interface
  – Two operations:
    » P(): Wait if zero; decrement when becomes non-zero
    » V(): Increment and wake a sleeping task (if exists)
  » Can initialize value to any non-negative value
  – Use separate semaphore for each constraint
• Monitors: A lock plus one or more condition variables
  – Always acquire lock before accessing shared data
  – Use condition variables to wait inside critical section
    » Three Operations: Wait(), Signal(), and Broadcast()