Recall: Multithreaded Stack Switching

• Consider the following code blocks:

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

• Suppose we have 2 threads running same code:
  – Threads S and T
  – Assume S and T have been running for a while

Recall: How does Thread get started?

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

Starting today: Pintos Projects

• Pintos:
  – Real OS
  – Emulated machine

• Working in Groups of four (4)
  – Work as one!
  – 10x homework

• Three Projects
  – P1: threads & scheduler
  – P2: user process
  – P3: file system
MT Kernel 1T Process ala Pintos/x86

• Each user process/thread associated with a kernel thread, described by a 4KB page object containing TCB and kernel stack for the kernel thread

Proc Regs
SP
KS
IP
PL: #

In User thread, w/ Kernel thread waiting

• x86 CPU holds interrupt SP in register
• During user thread execution, associated kernel thread is “standing by”

Proc Regs
SP
KS
IP
PL: 3

In Kernel Thread: No User Component

• Kernel threads execute with small stack in thread structure
• Pure kernel threads have no corresponding user-mode thread

Proc Regs
SP
KS
IP
PL: 0

User → Kernel (exceptions, syscalls)

• Mechanism to resume k-thread goes through interrupt vector

Proc Regs
SP
KS
IP
PL: 0
**Kernel → User**

- Interrupt return (iret) restores user stack, IP, and PL

**Pintos Interrupt Processing**

- IntrNN_stub
  - push 0x20 (int #)
  - jmp intr_entry
  - push 0x21 (int #)
  - jmp intr_entry

- Wrapper for generic handler
- intr_entry:
  - save regs as frame
  - set up kernel env.
  - call intr_handler
- intr_exit:
  - restore regs
  - iret

**User → Kernel via interrupt vector**

- Interrupt transfers control through the Interrupt Vector (IDT in x86)
- iret restores user stack and priority level (PL)

**Switch to Kernel Thread for Process**

- PL: 3
- PL: 0
**Pintos Interrupt Processing**

- `intrNN_stub()`: Wrapper for generic handler
- `intr_entry`: Save regs as frame, set up kernel env., call `intr_handler`
- `intr_exit`: Restore regs, iret

**Timer may trigger thread switch**

- **thread_tick**
  - Updates thread counters
  - If quanta exhausted, sets yield flag
- **thread_yield**
  - On path to rtn from interrupt
  - Sets current thread back to READY
  - Pushes it back on `ready_list`
  - Calls `schedule` to select next thread to run upon iret
- **Schedule**
  - Selects next thread to run
  - Calls `switch_threads` to change regs to point to stack for thread to resume
  - Sets its status to RUNNING
  - If user thread, activates the process
  - Returns back to `intr_handler`

**Thread Switch (switch.S)**

- `switch_threads`: Save regs on current small stack, change SP, return from destination threads call to `switch_threads`

**Pintos Return from Processing**

- `intrNN_stub()`: Wrapper for generic handler
- `intr_entry`: Save regs as frame, set up kernel env., call `intr_handler`
- `intr_exit`: Restore regs, iret

**Timer Intr_handler(*frame)**

- `timer_intr(*frame)`
  - `tick++`
  - `thread_tick()`
Kernel → Different User Thread

• iret restores user stack and priority level (PL)

Famous Quote WRT Scheduling: Dennis Richie

Dennis Richie,
Unix V6, slp.c:

“If the new process paused because it was swapped out, set the stack level to the last call to savu(ussav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu.”

“You are not expected to understand this.”

Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society (tuhs.org); gif by Eddie Koehler.

Included by Ali R. Butt in CS3204 from Virginia Tech

Administrivia

• Project 1 available today!
  – Get started looking at it with your group

• TA preference signup form due Today (Tuesday 2/12) at 11:59PM
  – Everyone in a group must have the same TA!
    » Preference given to same section

• Starting This Friday:
  – Attend your new (permanent) section
  – Get to know your TA!

Goals for Rest of Today

• Synchronization Operations
• Higher-level Synchronization Abstractions
  – Semaphores, monitors, and condition variables
• Programming paradigms for concurrent programs
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

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 ⇒ Input state determines results
    - Reproducible ⇒ 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”

Why allow cooperating threads?

- 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() {
    connection = AcceptCon();
    ProcessFork(ServiceWebPage(), connection);
  }

- What are some disadvantages of this technique?
Threaded Web Server

- Instead, use a single process
- Multithreaded (cooperating) version:
  ```c
  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
  - What about Denial of Service attacks or digg / Slashdot 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
  ```c
  master() {
    allocThreads(worker,queue);
    while(TRUE) {
      con=AcceptCon();
      Enqueue(queue,con);
      wakeUp(queue);
    }
  }
  ```
  ```c
  worker(queue) {
    while(TRUE) {
      con=Dequeue(queue);
      if (con==null)
        sleepOn(queue);
      else
        ServiceWebPage(con);
    }
  }
  ```

ATM Bank Server

- ATM server requirements:
  - 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
  ```c
  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 programming GPUs (Graphics Processing Unit)

Can Threads Make This Easier?

- Threads yield overlapped I/O and computation without having to “deconstruct” 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:
    ```c
    Thread 1
    load r1, acct->balance
    add r1, amount1
    store r1, acct->balance
    ```
    ```c
    Thread 2
    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:
  ```c
  Thread A
  x = 1; y = 2;
  ```
  ```c
  Thread B
  ```
- However, what about (Initially, y = 12):
  ```c
  Thread A
  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?
  ```c
  Thread A
  x = 1; x = 2;
  ```
  ```c
  Thread B
  ```
  - X could be 1 or 2 (non-deterministic!)
  - Could even be 3 for serial processors:
    - Thread A writes 0001, B writes 0010 → scheduling order ABABABBA 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
  r1=0  load r1, M[i]
  r1=1  add r1, r1, 1
  M[i]=1  store r1, M[i]
  ```

  Thread B
  ```
  r1=0  load r1, M[i]
  r1=-1  sub r1, r1, 1
  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/X-ray 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>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td>Leave for store</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>Buy milk</td>
<td></td>
</tr>
<tr>
<td>3:30</td>
<td>Arrive home, put milk away</td>
<td></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;
    }
  }
  ```
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):

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (noMilk) {</td>
<td>if (noMilk) {</td>
</tr>
<tr>
<td>if (noNote) {</td>
<td>if (noNote) {</td>
</tr>
<tr>
<td>leave Note;</td>
<td>buy Milk;</td>
</tr>
<tr>
<td>buy Milk;</td>
<td>remove Note;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>leave Note;</td>
</tr>
<tr>
<td></td>
<td>buy Milk;</td>
</tr>
<tr>
<td></td>
<td>remove Note;</td>
</tr>
</tbody>
</table>

- 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:

```python
Thread A
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:

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>leave note A;</td>
<td>leave note B;</td>
</tr>
<tr>
<td>if (noMilk) {</td>
<td>if (noMilk) {</td>
</tr>
<tr>
<td>if (noNoteA) {</td>
<td>if (noNoteB) {</td>
</tr>
<tr>
<td>buy Milk;</td>
<td>buy Milk;</td>
</tr>
<tr>
<td>}</td>
<td>remove note A;</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>remove note B;</td>
</tr>
</tbody>
</table>

- 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:

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

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

Case 1

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

```
leave note A;
while (note B) {
    do nothing;
}
if (noMilk) {
    buy milk;
}
remove note A;
if (noMilk) {
    buy milk;
}
remove note B;
```
Case 1

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

```java
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 2

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

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

Wait for note B to be remove

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

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

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

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

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