Review: Execution Stack Example

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

B() {
  C();
}

C() {
  A(2);
}

A(1);
exit;

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

addrY:
  printf(tmp);
  B() {
    C();
  }
  C() {
    A(2);
  }
  A(1);
  exit;

addrU:
  C() {
    A(2);
  }
  A(1);
  exit;

addrV:
  A(2);
  A(1);
  exit;

addrZ:
  A(1);
  exit;

• Stack holds function arguments, return address
• Permits recursive execution
• Crucial to modern languages

Stack
Pointer

Stack Growth
Review: Execution Stack Example

- Stack holds function arguments, return address
- Permits recursive execution
- Crucial to modern languages

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

Lec 3.5
1/25/12
Anthony D. Joseph and Ion Stoica CS162 ©UCB Spring 2012
Lec 3.6
1/25/12
Anthony D. Joseph and Ion Stoica CS162 ©UCB Spring 2012
Lec 3.7
1/25/12
Anthony D. Joseph and Ion Stoica CS162 ©UCB Spring 2012
Lec 3.8
1/25/12
Anthony D. Joseph and Ion Stoica CS162 ©UCB Spring 2012

Output:
Review: Execution Stack Example

Lec 3.9

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

B() {
  C();
}

C() {
  A(2);
}

A(1);
exit;


Stack Pointer

Stack Growth

Output:


Stack Pointer

Stack Growth

Output:


Stack Pointer

Stack Growth

Output:
Review: Execution Stack Example

```
A(int tmp) {
    if (tmp<2)
        B();
    printf(tmp);
}
B() {
    C();
}
C() {
    A(2);
    exit;
}
addrX: A(int tmp) {
    ...
    addrY: printf(tmp);
    ...
    addrU: B() {
        ...
        addrV: C();
    }
    ...
    addrW: A(1);
    ...
}
```

Output:

- 2
- 1

Single-Threaded Example

- Imagine the following C program:

```c
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

Goals for Today

- Thread Dispatching
- Cooperating Threads
- Concurrency examples
- Need for synchronization

Use of Threads

- Version of program with Threads:

```c
main() {
    CreateThread(ComputePI("pi.txt"));
    CreateThread(PrintClassList("clist.text"));
}
```

- What does “CreateThread” do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This should behave as if there are two separate CPUs

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne. Many slides generated from lecture notes by Kubiatowicz.
Memory Footprint of Two-Thread Example

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks

- Questions:
  - How do we position stacks relative to each other?
  - What maximum size should we choose for the stacks?
  - What happens if threads violate this?
  - How might you catch violations?

Per Thread State

- Each Thread has a Thread Control Block (TCB)
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: State, priority, CPU time
  - Various Pointers (for implementing scheduling queues)
  - Pointer to enclosing process? (PCB)?
  - Etc (add stuff as you find a need)

- OS Keeps track of TCBs in protected memory
  - In Array, or Linked List, or ...

Lifecycle of a Thread (or Process)

- As a thread executes, it changes state:
  - new: The thread is being created
  - ready: The thread is waiting to run
  - running: Instructions are being executed
  - waiting: Thread waiting for some event to occur
  - terminated: The thread has finished execution
- “Active” threads are represented by their TCBs
  - TCBs organized into queues based on their state

Ready Queue And Various I/O Device Queues

- Thread not running ⇒ TCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy
**Dispatch Loop**

- Conceptually, the dispatching loop of the operating system looks as follows:

  ```
  Loop {
    RunThread();
    ChooseNextThread();
    SaveStateOfCPU(curTCB);
    LoadStateOfCPU(newTCB);
  }
  ```

- This is an *infinite* loop
  - One could argue that this is all that the OS does

---

**Running a thread**

Consider first portion: \( \text{RunThread()} \)

- How do I run a thread?
  - Load its state (registers, PC, stack pointer) into CPU
  - Load environment (virtual memory space, etc)
  - Jump to the PC

- How does the dispatcher get control back?
  - Internal events: thread returns control voluntarily
  - External events: thread gets *preempted*

---

**Review: Yielding through 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
  ```
  computePI() {
    while(TRUE) {
      ComputeNextDigit();
      yield();
    }
  }
  ```
  - Note that `yield()` must be called by programmer frequently enough!

---

**Review: Stack for Yielding Thread**

- How do we run a new thread?
  ```
  run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* deallocates finished threads */
  }
  ```
  - Finished thread not killed right away. Why?
    - Move them in "exit/terminated" state
    - ThreadHouseKeeping() deallocates finished threads

---
Review: Stack for Yielding Thread

- How do we run a new thread?
  
  ```
  run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* deallocates finished threads */
  }
  ```

- How does dispatcher switch to a new thread?
  - Save anything next thread may trash: PC, regs, stack
  - Maintain isolation for each thread

Review: Two Thread Yield Example

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

- Suppose we have 2 threads:
  - Threads S and T

Review: Preemptive Multithreading

- Use the timer interrupt to force scheduling decisions
  ```
  TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
  }
  ```

- This is often called preemptive multithreading, since threads are preempted for better scheduling
  - Solves problem of user who doesn't insert `yield();`

Detour: Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-maskable interrupt line (NMI) can't be disabled
Announcements

• We are using Piazza instead of the newsgroup
  – Got to http://www.piazza.com/berkeley/spring2012/cs162
  – Make an account and join Berkeley, CS 162
  – Please ask questions on Piazza instead of emailing TAs

• Section assignments posted on Piazza
  – Attend new sections THIS week

• Suggestions for in-class question technology?
  – Email cs162@cory

• Question for the break:
  – Propose best practices for managing a home computer
    (things break, viruses, we live in an earthquake zone, …)

5min Break

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
  – Multiprocessors – chop up program into parallel pieces

• Advantage 3: Modularity
  – Chop large problem up into simpler pieces
    » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
  – Makes system easier to extend

Threaded Web Server

• Multithreaded version:
  serverLoop()
  {
    connection = AcceptCon();
    ThreadCreate(ServiceWebPage(),connection);
  }

• Advantages of threaded version:
  – 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 if too many requests come in at once?
Thread Pools

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

```java
master() {
  allocThreads(slave, queue);  while (TRUE) {
      con=Dequeue(queue);  if (con==null)
          sleepOn(queue);  else 
      ServiceWebPage(con);
      wakeUp(queue);  }
}
slave(queue) {
    while (TRUE) {
      con=Dequeue(queue);  if (con==null)
          sleepOn(queue);  else
      ServiceWebPage(con);
    }
}
```

ATM Bank Server

- 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:

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

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();
    }
}
```
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:
  ```
  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
  load r1, acct->balance
  add r1, amount1
  store r1, acct->balance
  ```
  ```
  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:
  ```
  Thread A
  x = 1;
  y = 2;
  ```
  ```
  Thread B
  x = 1;
  y = y*2;
  ```
- However, What about (Initially, y = 12):
  ```
  Thread A
  x = 1;
  x = y+1;
  ```
  ```
  Thread B
  y = y*2;
  ```
  - What are the possible values of x?
    ```
    Thread A
    x = 1;
    x = y+1;
    ```
    ```
    Thread B
    y = 2;
    y = y*2;
    ```
  
Problem is at the lowest level

- Most of the time, threads are working on separate data, so scheduling doesn’t matter:
  ```
  Thread A
  x = 1;
  ```
  ```
  Thread B
  y = 2;
  ```
- However, What about (Initially, y = 12):
  ```
  Thread A
  x = 1;
  x = y+1;
  ```
  ```
  Thread B
  y = y*2;
  ```
  - What are the possible values of x?
    ```
    Thread A
    x = 1;
    x = y+1;
    ```
    ```
    Thread B
    y = 2;
    ```
    ```
    x = 13
    ```
    ```
    y = 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
• 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

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.”

Space Shuttle Example

• Original Space Shuttle launch aborted 20 minutes before scheduled launch
• Shuttle has five computers:
  – Four run the “Primary Avionics Software System” (PASS)
    » Asynchronous and real-time
    » Runs all of the control systems
    » Results synchronized and compared 440 times per second
  – The Fifth computer is the "Backup Flight System" (BFS)
    » Stays synchronized in case it is needed
    » Written by completely different team than PASS
• Countdown aborted because BFS disagreed with PASS
  – A 1/67 chance that PASS was out of sync one cycle
  – Bug due to modifications in initialization code of PASS
    » A delayed init request placed into timer queue
    » As a result, timer queue not empty at expected time to force use of hardware clock
  – Bug not found during extensive simulation

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 | Thread B |
    |---------|---------|
    | i = 0;  | i = 0;  |
    | while (i < 10) | while (i > -10) |
    | i = i + 1; | i = i - 1; |
    | printf("A wins!"); | 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 it both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?
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