CMSC 330: Organization of Programming Languages

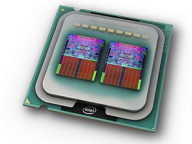
Multithreaded Programming Patterns in Java

Overview

- Introduction
- Java threads review
  - Topics from CMSC 132
- Producer / consumer pattern
- Conditions
  - wait( )
  - notifyAll( )

Multiprocessors

- Description
  - Multiple processing units (multiprocessor)
  - From single microprocessor to large compute clusters
  - Can perform multiple tasks in parallel simultaneously



Intel Core 2 Quad 6600

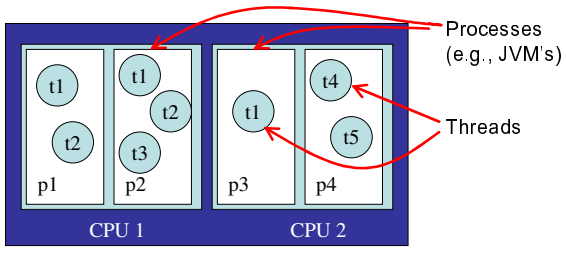
32 processor Pentium Xeon

106K processor IBM BlueGene/L

Concurrency

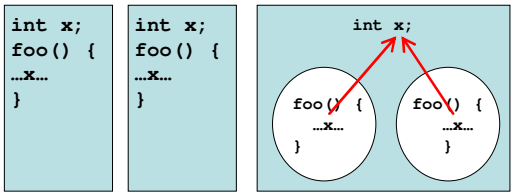
- Important & pervasive topic in CS
- Currently covered in
  - CMSC 132 – object-oriented programming II
    - Java threads, data races, synchronization
  - CMSC 313 – low level programming / computer systems
    - C pthreads
  - CMSC 411/430 – architectures / compilers
    - Instruction level parallelism
  - CMSC 412 – operating systems
    - Concurrent processes
  - CMSC 424 – database design
    - Concurrent transactions
  - CMSC 433 – programming language technologies
    - Advanced synchronization
  - CMSC 451 – algorithms
    - Parallel algorithms

Computation Abstractions



A computer

Processes vs. Threads



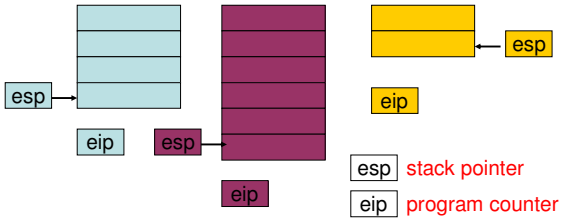
Processes do not share data

Threads share data within a process

So, What Is a Thread?

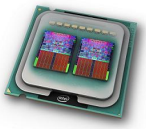
- Conceptually
  - Parallel computation occurring within a process
- Implementation view
  - A program counter and a stack
  - Heap and static area are shared among all threads
- All programs have at least one thread (main)

Implementation View



- Per-thread stack and instruction pointer
  - Saved in memory when thread suspended
  - Put in hardware esp/eip when thread resumes

Programming Threads



- Thread creation is inexpensive
- Threads reside on same physical processor
- Threads share memory, resources
  - Except for local thread variables
- Shared-memory programming paradigm
  - Threads communicate via shared data
  - Synchronization used to avoid data races
- Limited scalability (10's of threads)

Programming Processes

- Process creation is expensive
  - Request to operating system
- Processes may reside on separate processors
- Processes do not share memory
- Message-passing programming paradigm
  - Messages using I/O streams, sockets, network, files
- Processes must cooperate to communicate
  - Actions performed to send and receive data
- Highly scalable (1000's of processors)

Programming Languages & Threads

- Threads are available in many languages
  - C, C++, Java, Ruby, OCaml…
- In older languages (e.g., C and C++), threads are a platform specific add-on
  - Not part of the language specification
  - Implemented as code libraries (e.g., pthreads)
- In newer languages (e.g., Java, Ruby), threads are part of the language specification
  - Not dependent on operating system
  - Can utilize special keywords, syntax

Overview

- Introduction
- Java threads review ←
  - Topics from CMSC 132
- Producer / consumer pattern
- Conditions
  - wait( )
  - notifyAll( )

Java Threads Review

- Thread class & Runnable interface
  - Used to create / manipulate threads
- Run-time scheduler
  - Preemptive / non-premptive
  - Thread states (new, runnable, blocked, dead)
- Data race
  - Concurrent accesses to same shared object
    - Where at least one access is a write
  - Result may change depending on thread schedule
  - Very difficult to detect & correct

Java Threads Review (cont.)

- Synchronization
  - Locks ensure exclusive access
    - Provides mutual exclusion
      - Only 1 thread can obtain lock at a time
    - Lock associated w/ every Java object
  - Use synchronized keyword to acquire lock
    - Code blocks – synchronized (o) { … } // lock for Object o
    - Methods – synchronized foo( ) { … } // lock for this
  - Thread blocks when trying to acquire locked lock
    - Thread returns when lock is finally acquired
    - May deadlock if threads try to acquire each other's lock

Synchronization Example (Java 1.4)

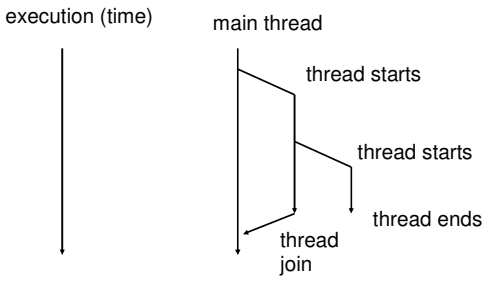
```
public class Example extends Thread {
  private static int cnt = 0;
  static Object value = new Object();
  public void run() {
    synchronized (value) {
      int y = cnt;
      cnt = y + 1;
    }
  }
  …
}
```

***Value**, any Java object*

***Acquires** the lock associated w/ value; only succeeds if not held by another thread, otherwise blocks*

***Releases** the lock*

Thread Creation



Creating Threads in Java 1

- Thread Class Approach
  - Extend Thread class and override run method
- Example

```
public class MyT extends Thread {
   public void run( ) {
      …                    // work for thread
   }
}
MyT t = new MyT( ) ;       // create thread
t.start( );                // begin running thread
…                          // thread executing in parallel
t.join();                  // waits for thread to exit
```


Creating Threads in Java 2

- Runnable Approach
  1. Define class implementing Runnable interface

```
public interface Runnable {
   public void run( );
}
```

  2. Put work to be performed in run( ) method
  3. Create instance of the "worker" class
  4. Create thread to run it
     - Create Thread object
       - Pass worker object to Thread constructor
     - Or hand the worker instance to an executor
       - Alternative methods for running threads

Creating Threads in Java 2 (cont.)

- Example

```
public class MyT implements Runnable {
   public void run( ) {
      …   // work for thread
   }
}
Thread t = new Thread(new MyT( ));  // create thread
t.start();                  // begin running thread
…                           // thread executing in parallel
t.join();                   // waits for thread to exit
```


Passing Parameters to Threads

- run( ) doesn't take parameters
- We "pass parameters" to the new thread by storing them as private fields
  - In the Runnable object

Overview

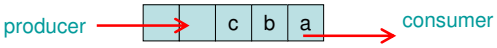
- Introduction
- Java threads review
  - Topics from CMSC 132
- Producer / consumer pattern
- Conditions
  - wait( )
  - notifyAll( )

Producer / Consumer Problem

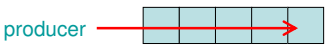
- Suppose we are communicating with a shared variable
  - E.g., a fixed size buffer holding messages
- One thread produces input to the buffer
- One thread consumes data from the buffer
- Rules
  - Producer can't add input to the buffer if it's full
  - Consumer can't take input from the buffer if it's empty

Producer / Consumer Idea

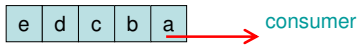
- If buffer is partially full, producer or consumer can run



- If buffer is empty, only producer can run



- If buffer is full, only consumer can run



Broken Producer/Consumer Example

```
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
  synchronized (value) {
    while (valueReady)
      ;
    value = o;
    valueReady = true;
  }
}
```

```
Object consume() {
  synchronized (value) {
    while (!valueReady)
      ;
    Object o = value;
    valueReady = false;
    return o;
  }
}
```

Threads wait with lock held – no way to make progress

Broken Producer/Consumer Example

```
boolean valueReady = false;
Object value;
```

```
void produce(object o) {
  while (valueReady)
    ;
  synchronized (value) {
    value = o;
    valueReady = true;
  }
}
```

```
Object consume() {
  while (!valueReady)
    ;
  synchronized (value) {
    Object o = value;
    valueReady = false;
    return o;
  }
}
```

valueReady accessed without a lock held – data race

Inefficient Producer/Consumer Example

```
boolean valueReady = false;
Object value;
```

Constantly acquiring / releasing lock — busy wait

```
void produce(Object o) {
  boolean done = false;
  while (!done) {
    synchronized (value) {
      if (!valueReady) {
        value = o;
        valueReady = true;
        done = true;
      }
    }
  }
}
```

```
Object consume() {
  Object o = null;
  boolean done = false;
  while (!done) {
    synchronized (value) {
      if (valueReady) {
        o = value;
        valueReady = false;
        done = true;
      }
  } }
  return o;
}
```


Overview

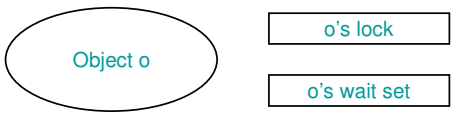
- Introduction
- Java threads review
  - Topics from CMSC 132
- Producer / consumer pattern
- Conditions ←
  - wait( )
  - notifyAll( )

Solving Producer / Consumer Problem

- Difficult to use locks directly
  - Very hard to get right
  - Problems often very subtle
- Proper approach – use condition variables
  - Definition: a set of threads associated w/ a lock
    - Also known as a wait set
    - Waiting for some condition to become true
  - Allows threads to sleep while waiting to acquire lock
  - Can wake up sleeping threads before releasing lock
- In Java 1.4
  - Each monitor has a single condition variable

Wait and NotifyAll (Java 1.4)

- Use synchronize on object to get associated lock



- Objects also have an associated wait set
  - Can be viewed as an implicit condition variable

Wait and NotifyAll (cont.)

- o.wait()
  - Must hold lock associated with o
  - Release that lock
    - And no other locks
  - Adds this thread to wait set for lock
  - Blocks the thread
- o.notifyAll()
  - Must hold lock associated with o
  - Resumes all threads on lock's wait set
  - Those threads must reacquire lock before continuing
    - This is part of the function; you don't need to do it explicitly

Producer/Consumer Example

```
boolean valueReady = false;
Object value;
```

```
void produce(Object o) {
  synchronized (value) {
    while (valueReady)
      wait();
    value = o;
    valueReady = true;
    notifyAll();
  }
}
```

```
Object consume() {
  synchronized (value) {
    while (!valueReady)
      wait();
    Object o = value;
    valueReady = false;
    notifyAll();
    return o;
  }
}
```


Using Conditions Correctly

- wait( ) must be called in a while loop
  - Conditions may not be met when await returns
  - Some other thread may have awoken first
    - And changed condition (e.g., consumed item in buffer)
- Avoid holding other locks when waiting
  - wait( ) only gives up lock on object you are waiting on
  - Reduces possibility of deadlock

Broken Producer/Consumer Example

```
boolean valueReady = false;
Object value;
```

```
void produce(Object o) {
  synchronized (value) {
    if (valueReady)
      wait();
    value = o;
    valueReady = true;
    notifyAll();
  }
}
```

```
Object consume() {
  synchronized (value) {
    if (!valueReady)
      wait();
    Object o = value;
    valueReady = false;
    notifyAll();
    return o;
  }
}
```

- Illegal access if multiple producers or consumers

Aspects of Synchronization

- Atomicity
  - Locking to obtain mutual exclusion
  - What we most often think about
- Visibility
  - Ensuring that changes to object fields made in one thread are seen in other threads
- Ordering
  - Ensuring that you aren't surprised by the order in which statements are executed

Key Ideas

- Multiple threads can run simultaneously
  - Either truly in parallel on a multiprocessor
  - Or can be scheduled on a single processor
    - A running thread can be pre-empted at any time
- Threads can share data
  - In Java, only fields can be shared
  - Need to prevent data races
    - Rule of thumb 1: You must hold a lock when accessing shared data
    - Rule of thumb 2: You must not release a lock until shared data is in a valid state
  - Overuse use of synchronization can create deadlock
    - Rule of thumb: No deadlock if only one lock