Programming Paradigms

Lecture 16:

Concurrency (Part 1)
Wake-up Exercise

What does this Java code print?

class Warmup {
    static boolean flag = false;
    static void raiseFlag() {
        flag = true;
    }
    public static void main(String ... {
        ForkJoinPool.commonPool()
            .execute(Warmup::raiseFlag);
        while (!flag) {};
        System.out.println(flag);
    }
}

https://ilias3.uni-stuttgart.de/vote/0ZT9
Wake-up Exercise

What does this Java code print?

class Warmup {
    static boolean flag = false;
    static void raiseFlag() {
        flag = true;
    }
    public static void main(String[] args) throws Exception {
        ForkJoinPool.commonPool()
            .execute(Warmup::raiseFlag);
        while (!flag) {};
        System.out.println(flag);
    }
}

raisesFlag: executed in concurrent thread
Wake-up Exercise

What does this Java code print?

class Warmup {
    static boolean flag = false;
    static void raiseFlag() {
        flag = true;
    }
    public static void main(String[] args)
        throws Exception {
        ForkJoinPool.commonPool()
            .execute(Warmup::raiseFlag);
        while (!flag) {};
        System.out.println(flag);
    }
}
Wake-up Exercise

What does this Java code print?

class Warmup {
    static boolean flag = false;
    static void raiseFlag() {
        flag = true;
    }
    public static void main(String[] args) throws Exception {
        ForkJoinPool.commonPool().execute(Warmup::raiseFlag);
        while (!flag) {};
        System.out.println(flag);
    }
}

Problem: No synchronization. Hence, main thread may read old value
Wake-up Exercise

What does this Java code print?

class Warmup {
    static boolean flag = false;
    static void raiseFlag() {
        flag = true;
    }
    public static void main(String[] args) throws Exception {
        ForkJoinPool.commonPool().execute(Warmup::raiseFlag);
        while (!flag) {};
        System.out.println(flag);
    }
}

Code may hang forever, print true, or print false!

https://ilias3.uni-stuttgart.de/vote/0ZT9
Overview

- Introduction
- Concurrent Programming Fundamentals
- Implementing Synchronization
- Language-level Constructs
Motivation

Why do we care about concurrency?

■ To capture the logical structure of a problem
  □ Inherently concurrent problems, e.g., server handling multiple requests

■ To exploit parallel hardware for speed
  □ Since around 2005: Multi-core processors are the norm

■ To cope with physical distribution
  □ Local or global groups of interacting machines
Terminology

- **Concurrent**
  - Two or more running tasks whose execution may be at some unpredictable point

- **Parallel**
  - Two or more tasks are actively executing at the same time
  - Requires multiple processor cores

- **Distributed**
  - Physically separated processors
Levels of Parallelism

- Signals propagating through circuits and gates
- Instruction-level parallelism
  - E.g., load from memory while another instruction executes
- Vector parallelism
  - E.g., GPUs execute a single instruction on a vector of data
- Thread-level parallelism
Levels of Parallelism

- Signals propagating through circuits and gates
- Instruction-level parallelism
  - E.g., load from memory while another instruction executes
  - Handled implicitly by hardware
- Vector parallelism
  - E.g., GPUs execute a single instruction on a vector of data
- Thread-level parallelism
Levels of Parallelism

- **Signals propagating through circuits and gates**
- **Instruction-level parallelism**
  - E.g., load from memory while another instruction executes
- **Vector parallelism**
  - E.g., GPUs execute a single instruction on a vector of data
- **Thread-level parallelism**

Specified by programmer in PL
Example: Independent Tasks

// Task Parallel Library in C#
Parallel.For(0, 100, i => {
    A[i] = foo(A[i]);
});
Example: Independent Tasks

// Task Parallel Library in C#
Parallel.For(0, 100, i => {
    A[i] = foo(A[i]);
});

Array of data
Function that updates each element independently
Example: Independent Tasks

```
// Task Parallel Library in C#
Parallel.For(0, 100, i => {
    A[i] = foo(A[i]);
});
```

- Array of data
- Function that updates each element independently

- No need to synchronize tasks
- Uses as many cores as possible (up to 100)
Example: Dependent Tasks

// As before, but foo now is:
int zero_count;
public static int foo(int n) {
    int rtn = n - 1;
    if (rtn == 0) zero_count ++;
    return rtn;
}
// As before, but foo now is:
int zero_count;
public static int foo(int n) {
    int rtn = n - 1;
    if (rtn == 0) zero_count ++;
    return rtn;
}
Data Race

Thread 1

r1 := zero_count
r1 := r1 + 1
zero_count := r1

Thread 2

r1 := zero_count
r1 := r1 + 1
zero_count := r1

--- data race
Data Races

Definition of data race

- Two accesses to the same shared memory location
- At least one access is a write
- Ordering of accesses is non-deterministic
Overview

- Introduction
- Concurrent Programming Fundamentals
- Implementing Synchronization
- Language-level Constructs
Processes, Threads, Tasks

- **Process**: Operating system construct that may execute threads
- **Thread**: Active entity that the programmer thinks of as running concurrently with other threads
- **Task**: Unit of work that must be performed by some thread
Processes, Threads, Tasks

- **Process**: Operating system construct that may execute threads
- **Thread**: Active entity that the programmer thinks of as running concurrently with other threads
- **Task**: Unit of work that must be performed by some thread

OS level

PL level

Logical level
Processes, Threads, Tasks

- **Process**: Operating system construct that may execute threads
- **Thread**: Active entity that the programmer thinks of as running concurrently with other threads
- **Task**: Unit of work that must be performed by some thread

Terminology differs across PLs and systems

More general than, e.g., Java’s “threads”
Communication

- Constructs to **pass information between threads**
  - **Shared memory**: Some variables accessible by multiple threads
  - **Message passing**: No shared state, but threads send messages to each other
  - Some PLs provide both
Synchronization

- **Mechanisms to control relative order of operations in different threads**
  - Explicit in shared-memory model
    - Must synchronize to ensure that variable read sees newest value stored in the variable
  - Implicit in message-passing model
    - Sender receives message after it has been sent
Spinning vs. Blocking

- Two forms of synchronization
- **Spinning** (also: busy-waiting)
  - Thread re-evaluates some condition until it becomes true (because of some other thread)
- **Blocking**
  - Waiting threads stops computation until some condition becomes true
  - Scheduler reactives the thread
## Examples

<table>
<thead>
<tr>
<th>Language</th>
<th>Shared memory</th>
<th>Message passing</th>
<th>Distributed computing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java, C#</td>
<td>Go</td>
<td></td>
<td>Erlang</td>
</tr>
<tr>
<td>C/C++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extension</td>
<td>OpenMP</td>
<td></td>
<td>Remote procedure call</td>
</tr>
<tr>
<td>Library</td>
<td>pthreads, Windows</td>
<td>MPI</td>
<td>Internet libraries</td>
</tr>
<tr>
<td>threads</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread Creation Syntax

- How to create a thread of execution?
- Five answers in popular PLs
  - Co-begin
  - Parallel loops
  - Launch-at-elaboration
  - Fork (with optional join)
  - Implicit receipt
Co-begin

- Compound statement where **all statements are executed concurrently**
- Example (pseudo-code):

  ```
  co-begin
  stmt_1
  stmt_2
  ...
  stmt_n
  end
  ```
Example: C with OpenMP

```c
#pragma omp sections
{
    #pragma omp section
    { printf("thread 1 here\n"); } 

    #pragma omp section
    { printf("thread 2 here\n"); } 
}
```
Example: C with OpenMP

```c
#pragma omp sections
{
    #pragma omp section
    { printf("thread 1 here\n"); }

    #pragma omp section
    { printf("thread 2 here\n"); }
}
```

**Pragmas: Compiler directives**
(# sign must be in first column)
Parallel Loops

- Loop whose iterations execute concurrently instead of sequentially

- Ex. 1: C with OpenMP

  ```c
  #pragma omp parallel for
  for (int i = 0; i < 3; i++) {
    printf("thread %d here\n", i);
  }
  ```

- Ex. 2: C# with Task Parallel Library

  ```csharp
  Parallel.For(0, 3, i => {
    Console.WriteLine("Thread "+ i + " here");
  });
  ```
Synchronization in Parallel Loops

- What about data races in parallel loops?
- Most PLs: Developer’s responsibility
- Some PLs: Implicit synchronization

  - E.g., forall loops in Fortran 95:
    - Synchronization on every assignment
      - All reads on right-hand side are before writes on the left-hand side
forall (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end forall
for all (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end for all

Reads and writes of array elements
Example: Fortran 95

Assignments: Implicit synchronization points

forall (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end forall

Reads and writes of array elements
Example: Fortran 95

forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall

At first, all threads read from B and C
Example: Fortran 95

forall (i=1:n-1)
A(i) = B(i) + C(i)
A(i+1) = A(i) + A(i+1)
end forall

At first, all threads read from $B$ and $C$
Then, all writes to $A(i)$ happen
Example: Fortran 95

```fortran
forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall
```

At first, all threads read from B and C.
The next writes to A(i) happen.

Next, all threads read the just written values from A.
Example: Fortran 95

forall (i=1:n-1)
  A(i) = B(i) + C(i)
  A(i+1) = A(i) + A(i+1)
end forall

At first, all threads read from \( B \) and \( C \)
Then, all writes to \( A(i) \) happen
Next, all threads read the just written values from \( A \)
Finally, the threads write updated values to \( A(i+1) \)
Quiz: Parallel Loops

forall (i=1:n-1)
    A(i) = B(i) + C(i)
    A(i+1) = A(i) + A(i+1)
end forall

What is the value of A after executing the loop with these initial values:

- A is 0, 0, 0
- B is 1, 2, 3
- C is 3, 2, 3
- n is 3

(Note: Arrays indices starts at 1 in Fortran)

https://ilias3.uni-stuttgart.de/vote/0ZT9
\[ T^1 \]
\[ i = 1 \]
\[ 1 + 3 = 4 \]
\[ A(1) = 4 \]
\[ 4 + 4 = 8 \]
\[ A(2) = 8 \]

\[ T^2 \]
\[ i = 2 \]
\[ 2 + 2 = 4 \]
\[ A(2) = 4 \]
\[ 4 + 0 = 4 \]
\[ A(3) = 4 \]

\[ \rightarrow A = [4, 8, 4] \]
Data Sharing in Parallel Loops

- Some PLs: Can specify which variables are shared among threads
- E.g., OpenMP
  - **Shared data**: All threads access same data
  - **Private data**: Each thread has its own copy
  - **Reduction**: Reduce a private variable across all threads at end of loop
double A[N];
double sum = 0;

#pragma omp parallel for 
    default(shared) reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += A[i];
}
printf("sum: %f\n", sum);
double A[N];
double sum = 0;

#pragma omp parallel for 
    default(shared) reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += A[i];
}
printf("sum: %f\n", sum);

All variables (except for i) are shared by default
Example: C with OpenMP

double A[N];
double sum = 0;

#pragma omp parallel for 
   default(shared) reduction (+: sum)
for (int i = 0; i < N; i++) {
    sum += A[i];
}
printf("sum: %f\n", sum);

All variables (except for i) are shared by default

Exception from default:
- Each thread has private copy of sum initialized before entering loop
- At end of loop, combine all copies with +
Launch-at-Elaboration

- Associate a thread with a specific subroutine
- Start thread when subroutine gets called
- At end of subroutine, wait for thread to complete
- Thread shares local variables with the subroutine
procedure P is
    task T is
        Put_Line ("In task T");
    end T;
begin
    Put_Line ("In default task of P");
end P;
Example: Ada

“Task” is Ada’s terminology for “thread”

```ada
procedure P is
  task T is
    Put_Line("In task T");
  end T;
begin
  Put_Line("In default task of P");
end P;
```

“Task” is Ada’s terminology for “thread”

Runs concurrently with (implicit) task of P
Fork/Join

- **Fork**: Explicit creation of a thread
- **Join**: Wait for a previously forked thread to terminate
Example: Java

class ImageRenderer extends Thread {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by the thread
    }
}

// ...

ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();
Example: Java

Threads: Subclasses of Thread

class ImageRenderer extends Thread {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by the thread
    }
}

// ...

ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();
Example: Java

Threads: Subclasses of Thread

class ImageRenderer extends Thread {
    ImageRenderer(someArg) {
    }
    public void run() {
        // code run by the thread
    }
}
// ...
ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();

Share values with thread via arguments
Example: Java

```java
class ImageRenderer extends Thread {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by the thread
    }
}
// ...
ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();
```

Threads: Subclasses of Thread

Share values with thread via arguments

Lifetime of thread
Example: C#

class ImageRenderer {  
    public void Render() {  
        // code to be run by the thread  
    }  
}  

// ...

ImageRenderer rendObj = new ImageRenderer();
Thread rend = new Thread(
    new ThreadStart(rendObj.Render));
rend.Start();
// ...
rend.Join();
Example: C#

```csharp
class ImageRenderer {
    public void Render() {
        // code to be run by the thread
    }
}

// ...

ImageRenderer rendObj = new ImageRenderer();
Thread rend = new Thread(
    new ThreadStart(rendObj.Render));
rend.Start();
// ...
rend.Join();
```

Just a normal method ...

... turned into a thread
Thread Pools

- **Goal:** Separate tasks to execute from how they are executed in threads
- **Thread pool:** Set of (idle) threads that can execute tasks
  - Reduces cost of creating and starting threads by reusing them
  - Let pool implementation decide how exactly to schedule tasks for execution
Example: Java

class ImageRenderer implements Runnable {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by this task
    }
}

// ...

Executor pool = Executors.newFixedThreadPool(4);
pool.execute(new ImageRenderer(...));
Example: Java

class ImageRenderer implements Runnable {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by this task
    }
}
// ...
Executor pool = Executors.newFixedThreadPool(4);
pool.execute(new ImageRenderer(...));

Not a Thread anymore, but only a Runnable
Example: Java

class ImageRenderer implements Runnable {
    ImageRenderer(someArg) { ... }
    public void run() {
        // code run by this task
    }
}

// ...

Executor pool = Executors.newFixedThreadPool(4);
pool.execute(new ImageRenderer(...));
Spawn and Sync in Cilk

- Extension of C language
- Programmer expresses tasks and their dependencies
  - `spawn` calls a function to be executed as a logically concurrent task
  - `sync` joins all tasks spawned by the calling task
- Scheduler assigns tasks to processor cores through work stealing
Example: Fibonacci

Sequential implementation:

```c
int fib (int n) {
    if (n < 2) return 1;
    else {
        int res = 0;
        res += fib (n - 1);
        res += fib (n - 2);

        return res;
    }
}
```
Example: Fibonacci

Parallel implementation with Cilk:

cilk int fib (int n) {
    if (n < 2) return 1;
    else {
        int res = 0;
        res += spawn fib (n - 1);
        res += spawn fib (n - 2);
        sync;
        return res;
    }
}
Example: Fibonacci

Parallel implementation with Cilk:

cilk int fib (int n) {
    if (n < 2) return 1;
    else {
        int res = 0;
        res += spawn fib (n - 1);
        res += spawn fib (n - 2);
        sync;
        return res;
    }
}

Execute in parallel with parent

Wait until children have returned
Implicit Receipt

- In remote procedure call (RPC)-based systems
- Create thread in response to an incoming request from some other address space
  - E.g., from another machine
Quiz: Concurrency

Which of the following sentences are true?

■ Concurrency means different machines perform computations at the same time.
■ In OpenMP’s parallel loops, all data is private to the respective thread.
■ A thread in a thread pool may execute any number of tasks.
■ The scheduler re-actives a busy-waiting thread.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Quiz: Concurrency

Which of the following sentences are true?

- Concurrency means different machines perform computations at the same time.
- In OpenMP’s parallel loops, all data is private to the respective thread.
- A thread in a thread pool may execute any number of tasks.
- The scheduler re-actives a busy-waiting thread.

https://ilias3.uni-stuttgart.de/vote/0ZT9
Overview

■ Introduction
■ Concurrent Programming Fundamentals
■ Implementing Synchronization
■ Language-level Constructs