Programming Paradigms

Concurrency (Part 1)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2024/2025

Overview

- Basics
- Creating and Managing Threads
- Implementing Synchronization
- Language Support for Synchronization
- Memory Consistency

Motivation

Why do we care about concurrency?

- To capture the logical structure of a problem
  - Inheritently 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
- Vector parallelism
  - E.g., GPUs execute a single instruction on a vector of data
- Thread-level parallelism

Handled implicitly by hardware

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;
}
```

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;
}
```

Count how many zeros written to the array

Problem: Data race

Thread 1	Thread 2
r1 := $\boxed{\text{zero\_count}}$	r1 := $\boxed{\text{zero\_count}}$
r1 := r1 + 1	r1 := r1 + 1
$\boxed{\text{zero\_count}}$:= r1	$\boxed{\text{zero\_count}}$:= 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

Another Example

```
// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
  if (account_from.balance < amount) return NOPE;
  atomic {
    account_to.balance += amount;
    account_from.balance -= amount;
  }
  return YEP;
}
```

Another Example

```
// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
  if (account_from.balance < amount) return NOPE;
  atomic {
    account_to.balance += amount;
    account_from.balance -= amount;
  }
  return YEP;
}
```

Ensures that code block is executed atomically, i.e., no other thread executes it at the same time

Another Example

```
// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
  if (account_from.balance < amount) return NOPE;
  atomic {
    account_to.balance += amount;
    account_from.balance -= amount;
  }
  return YEP;
}
```

Ensures that code block is executed atomically, i.e., no other thread executes it at the same time

Quiz: Could a program invoking `transfer` multiple times concurrently have a data race?

Another Example

```
// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
  if (account_from.balance < amount) return NOPE;
  atomic {
    account_to.balance += amount;
    account_from.balance -= amount;
  }
  return YEP;
}
```

Yes, there still is a data race:
Concurrent and unsynchronized <u>read</u>
and <u>write</u> of `account_from.balance`

Another Example

Fixed version:

```
// code to transfer money between accounts
// written in a toy language
fun transfer(amount, account_from, account_to) {
  atomic {
    if (account_from.balance < amount) return NOPE;
    account_to.balance += amount;
    account_from.balance -= amount;
  }
  return YEP;
}
```

Overview

- Basics
- Creating and Managing Threads
- Implementing Synchronization
- Language Support for Synchronization
- Memory Consistency

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

OS level

- **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

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

Examples

	Shared memory	Message passing
Language	Java, C#, C/C++	Erlang, Go
Extension	OpenMP	Java/Scala actors
Library	pthreads, Windows threads	MPI

Quiz: Terminology

Which of the following sentences are true?

- Concurrency means two or more tasks are actively executing at the same time.
- A data race can occur even if only one thread writes to a shared variable.
- Vector parallelism should be avoided to ensure correctness.
- In PLs with message passing, threads exchange message via a shared memory region.

Quiz: Terminology

Which of the following sentences are true?

- ~~Concurrency means two or more tasks are actively executing at the same time.~~
- A data race can occur even if only one thread writes to a shared variable.
- ~~Vector parallelism should be avoided to ensure correctness.~~
- ~~In PLs with message passing, threads exchange message via a shared memory region~~.

Thread Creation & Management

Different PLs provide different constructs:

- **Co-begin**
- **Parallel loops**
- **Fork (with optional join)**
- **POSIX threads aka pthreads**
- **Message passing**

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

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

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

Example: C with OpenMP

```
#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

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

- Ex. 2: C# with Task Parallel Library

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

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

Example: C with OpenMP

```c
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);
```

Example: C with OpenMP

All variables (except for `i`)
are shared by default

```
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);
```

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 +

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
  }
}
```

Share values with thread via arguments

```
// ...

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
  }
}
```

Share values with thread via arguments

```
// ...

ImageRenderer rend = new ImageRenderer(...);
rend.start();
// ...
rend.join();
```

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#

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

// ...
```

Just a normal method ...

... turned into a thread

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

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(...));
```

Not a `Thread` anymore, but only a `Runnable`

Pool with 4 threads used to execute any number of tasks

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:

```
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

POSIX Threads (“pthreads”)

- **Low-level API for creating and managing threads**
  - Very expressive and flexible, but non-trivial to use correctly
- **Supported by C and C++ across all Unix/Linux flavors**

POSIX Threads: Creation

Creating a new thread:

```c
#include <pthread.h>

pthread_t thread_id;
pthread_create(&thread_id, NULL, thread_function, arg);
```

Unique ID to refer to thread

Optional attributes (e.g., scheduling priority)

Function to execute in thread

Arguments to pass to function

POSIX Threads: Termination

- **Terminate the currently running thread:**

  ```
  pthread_exit((void *) return_value);
  ```

  - Implicitly called when thread's function terminates

- **Block and wait for another thread to join:**

  ```
  pthread_join(thread_id, NULL);
  ```

Quiz: POSIX Threads

```
#include <pthread.h>
#include <stdio.h>

void *print_message(void *arg) {
  printf("%s", (char *)arg);
  return NULL;
}
int main() {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, print_message, "a");
  pthread_create(&thread2, NULL, print_message, "b");
  printf("c");
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

How many different outputs could this C code have?

Quiz: POSIX Threads

Solution: All 6 permutations of the 3 strings

```
#include <pthread.h>
#include <stdio.h>

void *print_message(void *arg) {
  printf("%s", (char *)arg);
  return NULL;
}
int main() {
  pthread_t thread1, thread2;
  pthread_create(&thread1, NULL, print_message, "a");
  pthread_create(&thread2, NULL, print_message, "b");
  printf("c");
  pthread_join(thread1, NULL);
  pthread_join(thread2, NULL);
  return 0;
}
```

How many different outputs could this C code have?

Message Passing

- **Avoids pitfalls of shared memory (e.g., race conditions) and need for low-level concurrency mechanisms**
- **Two key constructs**
  - Independent threads
    - E.g., “actors” in Java/Scala or “goroutines” in Go
  - Messages exchanged between threads
    - E.g., via “channels” in Go and Rust

Message Passing: Scala

```
import akka.actor._
case class Greet(message: String)
class Greeter extends Actor {
    def receive = {
        case Greet(message) => println(message)
    }
}
object Main extends App {
    val system = ActorSystem("HelloSystem")
    val greeter = system.actorOf(Props[Greeter],
                                  name = "greeter")
    greeter ! Greet("Hello, World!")
}
```

Overview

- Basics
- Creating and Managing Threads ✔
- Implementing Synchronization
- Language Support for Synchronization
- Memory Consistency