TDDA69 Data and Program Structure

Shared-State Concurrency

Cyrille Berger

LINKÖPING UNIVERSITY

Lectures

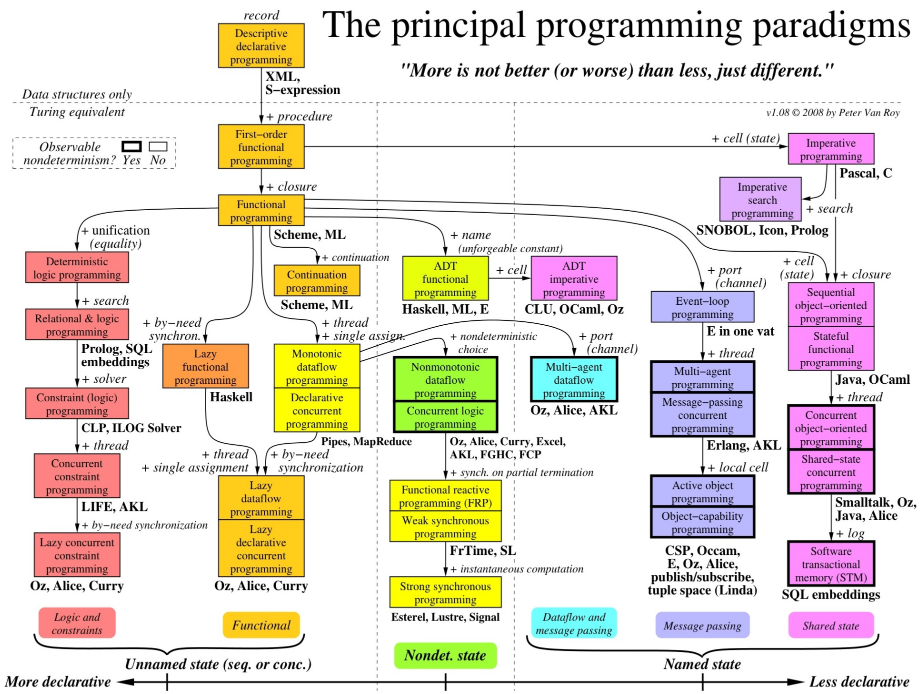
Lecture content

- Concurrent Computing Programming Models
- Shared-state Concurrency
  - Multithreaded Programming
    - The States Problems and Solutions
      - Atomic actions
    - Language and Interpreter Design Considerations
  - Multiprocess programming
    - Multiprocess programming in Python
- Single Instruction, Multiple Threads Programming

Concurrent Computing Programming Models

Concurrent computing programming

- In *Sequencial programming*: single computation executed at a given time
- In *concurrent computing*: several computations are executed at the same time
- Three basic approach to concurrency:
  - *Declarative concurrency*: streams in a functional language
  - *Message passing*: with active objects, used in *distributed computing*
  - *Shared-State Concurrency*: on a shared memory, with atomic operation



Declarative Concurrency

- Extend functional programming with *thread*

```
var a, b
a = thread { b + 1 }
b = 5
```

- *thread* are expressions run concurrently, unbound variable block the execution of a thread
- Keep all the benefits of pure functional

Execution models

- Data-driven concurrency: a thread is executed as soon as it has all the data
- Demand-driven conurrency (+by-need-synchronization): a thread is executed when its result is needed
- Streams: each thread performs a computation on a set of streams

Threaded Fibonacci

- Functions can create new threads:

```
function fib(x)
   <- x <= 2 ? 1 : thread
{ fib(x-1) }
            + fib(x-2)
```

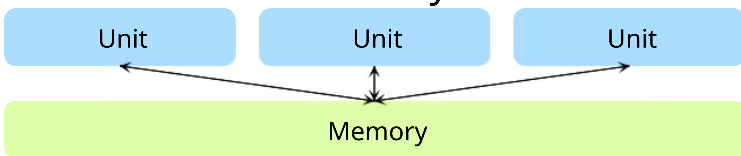

Message Passing Concurrency

- In declartive concurrent programming there is *no observable nondeterminism*
- Not applicable to client/server applications
  - No knowledge of the clients (number,...)
  - No control on when message arrive

Shared-state Concurrency

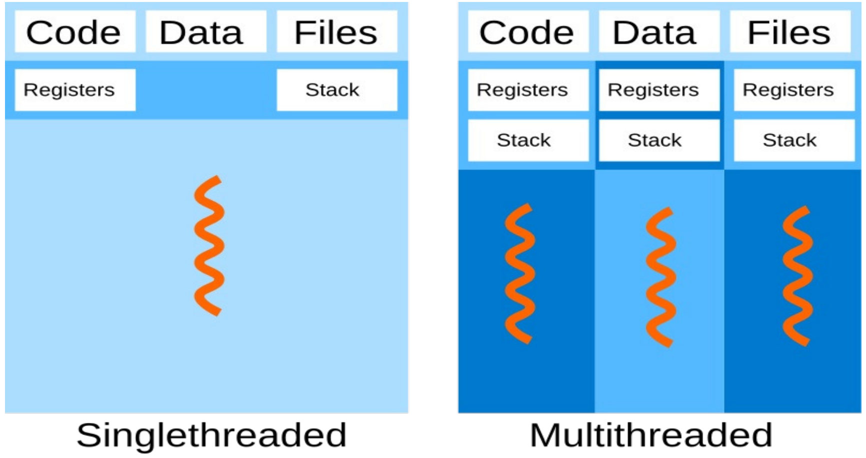
Parallel Programming

- In *parallel computing* several computations are executed at the **same time** and have access to **shared** memory



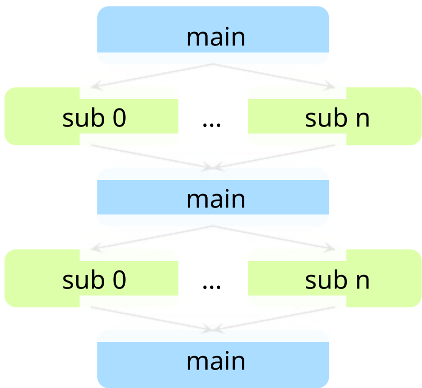
Multithreaded Programming

Singlethreaded vs Multithreaded



Multithreaded Programming Mod

- Start with a single root thread
- **Fork**: to create concurently executing threads
- **Join**: to synchronize threads
- Threads communicate through shared memory
- Threads execute assynchronously
- They may or may not execute on different processors



A multithreaded example

```
define thread1 = thread()
  {
    /* do some computation */
  };
define thread2 = thread()
  {
    /* do some computation */
  };
thread1.start();
thread2.start();
thread1.join();
thread2.join();
```

The States Problems and Solutions

LINKÖPING UNIVERSITY

Global States and multi-threading

- Example:

```
cell a = 0;
define thread1 = thread()
  {
    a = a + 1;
  });
define thread2 = thread()
  {
    a = a + 1;
  });
thread1();
thread2();
```

- What is the value of a ?
- This is called a *(data) race* condition

Atomic actions

LINKÖPING UNIVERSITY

Atomic operations

- An operation is said to be atomic, if it appears to happen instantaneously
  - *read/write*, *swap*, *fetch-and-add*...
- *test-and-set*: set a value and return the old one
  - To implement a lock:
    ```
    while(test-and-set(lock, 0, 1) == 1) {}
    ```
  - To unlock:
    ```
    lock = 0
    ```

Mutex

- *Mutex* is the short of *Mutual exclusion*
  - It is a technique to prevent two threads to access a shared resource at the same time
- Example:

```
cell a = 0;
define m = mutex();
define thread1 = thread()
  {
    m.lock();
    a = a + 1;
    m.unlock();
  };
define thread2 =
thread()
  {
    m.lock();
    a = a + 1;
    m.unlock();
  };
thread1.start();
thread2.start();
```

- Now a=2

Dependency

- Example:

```
cell a = 1;
define m = mutex();
define thread1 =
thread()
  {
    m.lock();
    a = a + 1;
    m.unlock();
  };
define thread2 =
thread()
  {
    m.lock();
    a = a * 3;
    m.unlock();
  };
thread1.start();
thread2.start();
```

- What is the value of a ? 4 or 6 ?

Condition variable

- A *Condition variable* is a set of threads waiting for a certain condition
- Example:

```
cell a = 1;
define m = mutex();
define cv =
condition_variable();
define thread1 = thread()
  {
    m.lock();
    a = a + 1;
    cv.notify();
    m.unlock();
  };
define thread2 = thread()
  {
    cv.wait();
    m.lock();
    a = a * 3;
    m.unlock();
  };
thread1.start();
thread2.start();
```

- a = 6

Deadlock

- What might happen:

```
var a = 0;
var b = 2;
var ma = new Mutex();
var mb = new Mutex();
var thread1 = new Thread(
  function()
  {
    ma.lock();
    mb.lock();
    b = b - 1;
    a = a - 1;
    ma.unlock();
    mb.unlock();
  });
var thread2 = new Thread(
  function()
  {
    mb.lock();
    ma.lock();
    b = b - 1;
    a = a + b;
    mb.unlock();
    ma.unlock();
  });
thread1.start();
thread2.start();
```

- thread1 waits for mb, thread2 waits for ma

Advantages of atomic actions

- Very efficient
- Less overhead, faster than message passing

Disadvantages of atomic actions

- Blocking
  - Meaning some threads have to wait
- Small overhead
- Deadlock
- A low-priority thread can block a high priority thread
- A **common** source of programming errors

Language and Interpreter Design Considerations

Common mistakes

- Forget to unlock a mutex
- Race condition
- Deadlocks
- Granularity issues: too much locking will kill the performance

Forget to unlock a mutex

- Most programming language have, either:
  - A guard object that will unlock a mutex upon destruction
  - A synchronization statement

```
some_rlock = threading.RLock()
with some_rlock:
  print("some_rlock is locked while this
executes")
```


Race condition

- Can we detect potential race condition during compilation?
- In the *rust* programming language
  - Objects are owned by a specific thread
  - Types can be marked with *Send* trait
    indicate that the object can be moved between threads
  - Types can be marked with *Sync* trait
    indicate that the object can be accessed by multiple threads safely

Safe Shared Mutable State in rust (1/3)

```
let mut data = vec![1, 2, 3];
for i in 0..3 {
  thread::spawn(move || {
      data[i] += 1;
  });
}
```

Gives an error: "capture of moved value: `data`"

Safe Shared Mutable State in rust (2/3)

```
let mut data = Arc::new(vec![1, 2, 3]);
for i in 0..3 {
  let data = data.clone();
  thread::spawn(move || {
      data[i] += 1;
  });
}
```

- Arc add reference counting, is movable and syncable
- Gives error: cannot borrow immutable borrowed content as mutable

Safe Shared Mutable State in rust (3/3)

```
let data = Arc::new(Mutex::new(vec![1, 2,
3]));
  for i in 0..3 {
    let data = data.clone();
    thread::spawn(move || {
      let mut data =
data.lock().unwrap();
        data[i] += 1;
      });
  }
```

- It now compiles and it works

Extend KL for safe shared cells (1/2)

- No need to extend the syntax, only change the meaning of a cell
  - Each cell is now associated with a thread

```
cell a = 0;
define thread1 = thread()
  {
    a = a + 1;
  });
define thread2 = thread()
  {
    a = a + 1;
  });
thread1();
thread2();
```

- It nows trigger an error because `a` belongs to the main thread and cannot be used in other threads

Extend KL for safe shared cells (2/2)

```
cell a = 0;
define m = mutex(a);
define thread1 = thread()
  {
    cell b = m.lock();
    b = b + 1;
    m.unlock();
  });
define thread2 = thread()
  {
    cell b = m.lock();
    b = b + 1;
    m.unlock();
  });
thread1();
thread2();
cell b = m.lock();
print(b)
m.unlock()
```

- The mutex takes ownership of `a`, the only way to access it is with `m.lock()` which returns a cell owned by the current thread

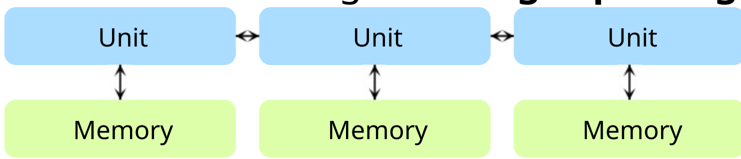
Deadlock?

- Runtime detection
- Prevention
  - still a hard problem
  - lock hierarchy
  - wait-for-graph

Multiprocess programming

Distributed Programming (1/2)

- In *distributed computing* several computations are executed at the **same time** and communicate through **messages passing**



Distributed Programming (2/2)

- Individual programs have differentiating roles.
- Distributed computing for large-scale data processing:
  - Databases respond to queries over a network.
  - Data sets can be partitioned across multiple machines.

Multiprocess programming in Python

Python's Global Interpreter Lock (1/2)

- CPython can only interpret one single thread at a given time
- The lock is released, when:
  - The current thread is blocking for I/O
  - Every 100 interpreter *ticks*
- True multithreading is not possible with CPython

Python's Global Interpreter Lock (2/2)

- CPython can only interpret one single thread at a given time
  - Single-threaded programms are faster (no need to lock in memory management)
  - Many C-library used as extensions are not thread safe
- To eliminate the GIL Python developers have the following requirements:
  - Simplicity
  - Do actually improve performance
  - Backward compatible
  - Prompt and ordered destruction

Python's Multiprocessing module

- The multiprocessing package offers both local and remote concurrency, effectively side-stepping the Global Interpreter Lock by using subprocesses instead of threads
- It implements transparent message passing, allowing to exchange Python objects between processes

Python's Message Passing (1/2)

- Example of message passing

```
from multiprocessing import Process

def f(name):
    print 'hello', name

if __name__ == '__main__':
    p = Process(target=f, args=('bob',))
    p.start()
    p.join()
```

- Output

hello bob

Python's Message Passing (2/2)

- Example of message passing with pipes

```
from multiprocessing import Process, Pipe

def f(conn):
    conn.send([42, None, 'hello'])
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print parent_conn.recv()
    p.join()
```

- Output
  [42, None, 'hello']
- Transparent message passing is possible thanks to *serialization*

Serialization

- A *serialized object* is an object represented as a sequence of bytes that includes the object's data, its type and the types of data stored in the object.

pickle

- In Python, serialization is done with the *pickle* module
  - It can serialize user-defined classes
    - The class definition must be available before deserialization
  - Works with different version of Python
  - By default, use an ASCII format
- It can serialize:
  - Basic types: booleans, numbers, strings
  - Containers: tuples, lists, sets and dictionnary (of pickable objects)
  - Top level functions and classes (only the name)
  - Objects where __dict__ or __getstate()__ are pickable
- Example:
  - `pickle.loads(pickle.dumps(10))`

Shared memory

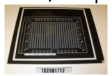
- Memory can be shared between Python process with a *Value* or *Array*.

```
from multiprocessing import Process, Value, Array, RLock

def f(n, a, m):
    with m:
        n.value = 3.1415927
        for i in range(len(a)):
            a[i] = -a[i]

if __name__ == '__main__':
    num = Value('d', 0.0)
    arr = Array('i', range(10))
    m   = RLock()
    p = Process(target=f, args=(num, arr, m))
    p.start()
    p.join()

    print num.value
    print arr[:]
```

Single Instruction, Multiple Threads Programming

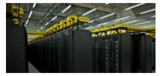
SIMD, SIMT, SMT (1/2)

- SIMD: Single Instruction, Multiple Data
  - Elements of a short vector (4 to 8 elements) are processed in parallel



- SIMT: Single Instruction, Multiple Threads
  - The same instruction is executed by multiple threads (from 128 to 3048 or more in the future)

- SMT: Simultaneous Multithreading
  - General purpose, different instructions are executed by different threads



SIMD, SIMT, SMT (2/2)

- SIMD:

```
PUSH [1, 2, 3, 4]
PUSH [4, 5, 6, 7]
VEC_ADD_4
```

- SIMT:

```
execute([1, 2, 3, 4], [4, 5, 6, 7], lambda a,b,ti: a[ti]=a[ti] + max(b[ti], 5))
```

- SMT:

```
a = [1, 2, 3, 4]
b = [4, 5, 6, 7]
...
Thread.new(lambda : a = a + b)
Thread.new(lambda : c = c * b)
```


Why the need for the different models?

- Flexibility:
  - SMT > SIMT > SIMD
- Less flexibility give higher performance
  - Unless the lack of flexibility prevent to accomplish the task
- Performance:
  - SIMD > SIMT > SMT

Single Instruction, Multiple Threads Programming

- With SIMT, the same instructions is executed by multiple threads on different registers
- Is it a problem for control flow?

Single instruction, multiple flow paths (1/2)

- Using a *masking* system, it is possible to support control flow
  - Threads are always executing the instruction of both part of the if/else blocks

```
data = [-2, 0, 1, -1, 2], data2 = [...]
function f(thread_id, data, data2)
{
  if(data[thread_id] < 0)
  {
    data[thread_id] = data[thread_id]-data2[thread_id];
  } else if(data[thread_id] > 0)
  {
    data[thread_id] = data[thread_id]+data2[thread_id];
  }
}
```

- Assignement is only performed according to the mask

Single instruction, multiple flow paths (1/2)

- Benefits:
  - Multiple flows are needed in many algorithms
- Drawbacks:
  - Only one flow path is executed at a time, non running threads must wait
  - Randomize memory access

    Elements of a vector are not accessed sequentially

Programming Language Design for SIMT

- General purpose programming language are not suitable
- OpenCL, CUDA are the most common
  - Very low level, C/C++-derivative
- Some work has been done to be able to write in Python and run on a GPU with CUDA

```
@jit(argtypes=[float32[:], float32[:], float32[:]],
target='gpu')
def add_matrix(A, B, C):
  A[cuda.threadIdx.x] = B[cuda.threadIdx.x]
                                + C[cuda.threadIdx.x]
```

with limitation on standard function that can be called

Benefits of concurrent computing

- Faster computation
- Responsiveness
  - Interactive applications can be performing two tasks at the same time: rendering, spell checking...
- Availability of services
  - Load balancing between servers
- Controllability
  - Tasks can be suspended, resumed and stopped.

Disadvantages of concurrent computing

- Concurrency is hard to implement properly
- Safety
  - Easy to corrupt memory
- Deadlock
  - Tasks can wait indefinitely for each other
- Non-deterministic
- Not always faster!
  - The memory bandwidth and CPU cache is limited

Summary

- Concurrent programming
- Declarative Concurrent Programming, streams
- Message Passing Concurrent Programming
- The challenges of Shared State Concurrent Programming