Programming Languages

Tevfik Koşar

Lecture - XXV
April 25th, 2006

Roadmap

- Concurrent Programming
  - Shared Memory vs Message Passing
  - Divide and Compute
  - Threads vs Processes
  - Synchronization
Concatenation Programming

• So far, we have focused on sequential programming: all computational tasks are executed in sequence, one after the other.
• Next three lectures, we will focus on concurrent programming: multiple computational tasks are executed simultaneously, at the same time.

Sequential

Concurrency

• Implementation of concurrent tasks:
  - as separate programs
  - as a set of processes or threads created by a single program

• Execution of concurrent tasks:
  - on a single processor
    ➔ Multithreaded programming
  - on several processors in close proximity
    ➔ Parallel computing
  - on several processors distributed across a network
    ➔ Distributed computing
Communication Between Tasks

Interaction or communication between concurrent tasks can done via:

- **Shared memory:**
  - all tasks has access to the same physical memory
  - they can communicate by altering the contents of shared memory

- **Message passing:**
  - no common/shared physical memory
  - tasks communicate by exchanging messages

Motivation

- Increase the performance by running more than one tasks at a time.
  - divide the program to n smaller pieces, and run it n times faster using n processors

- To cope with independent physical devices.
  - do not wait for a blocked device, perform other operations at the background
Divide and Compute

\[ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \]

How many operations with sequential programming?

7

Step 1: \( x_1 + x_2 \)

Step 2: \( x_1 + x_2 + x_3 \)

Step 3: \( x_1 + x_2 + x_3 + x_4 \)

Step 4: \( x_1 + x_2 + x_3 + x_4 + x_5 \)

Step 5: \( x_1 + x_2 + x_3 + x_4 + x_5 + x_6 \)

Step 6: \( x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \)

Step 7: \( x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \)

Divide and Compute

\[ 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 \]

Step 1: \( 1 + 2 = 3 \)

Step 2: \( 3 + 3 = 6 \)

Step 3: \( 6 + 4 = 10 \)

Step 4: \( 10 + 5 = 15 \)

Step 5: \( 15 + 6 = 21 \)

Step 6: \( 21 + 7 = 28 \)

Step 7: \( 28 + 8 = 36 \)
Divide and Compute

\[ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \]

Step 1: parallelism = 4

Step 2: parallelism = 2

Step 3: parallelism = 1

Divide and Compute

\[ 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 \]

Step 1: parallelism = 4

Step 2: parallelism = 2

Step 3: parallelism = 1
Gain from parallelism

In theory:
- dividing a program into $n$ smaller parts and running on $n$ processors results in $n$ time speedup

In practice:
- This is not true, due to
  - Communication costs
  - Dependencies between different program parts
    - Eg. the addition example can run only in $\log(n)$ time not $1/n$

Prevent Blocking

- Do not wait for a blocked device, perform other operations at the background
  - During I/O perform computation
  - During continuous visualization, handle key strokes and I/O
    - Eg. video games
  - While listening to network, perform other operations
    - Listening to multiple sockets at the same time
  - Concurrent I/O, concurrent transfers
    - Eg. Web browsers
Threads vs Processes

Process Spawning:
Process creation involves the following four main actions:
- setting up the process control block,
- allocation of an address space and
- loading the program into the allocated address space and
- passing on the process control block to the scheduler

![Process Diagram](parent-process-child-processes-diagram)

Threads vs Processes

Thread Spawning:
- Threads are created within and belonging to processes
- All the threads created within one process share the resources of the process including the address space
- Scheduling is performed on a per-thread basis.
- The thread model is a finer grain scheduling model than the process model
- Threads have a similar lifecycle as the processes and will be managed mainly in the same way as processes are

![Thread Diagram](process-threads-diagram)
Threads vs Processes

- Heavyweight Process = Process
- Lightweight Process = Thread

Advantages (Thread vs. Process):
- Much quicker to create a thread than a process
- Much quicker to switch between threads than to switch between processes
- Threads share data easily

Disadvantages (Thread vs. Process):
- Processes are more flexible
  - They don’t have to run on the same processor
- No security between threads: One thread can stomp on another thread's data
- For threads which are supported by user thread package instead of the kernel:
  - If one thread blocks, all threads in task block.

Synchronization

- Mechanism that allows the programmer to control the relative order in which operations occur in different threads or processes.
Synchronization - Threads

```java
Int sum = 0;

Thread 1:
int t;
lock(sum);
sum = sum + x;
t = sum;
....
unlock(sum);
```

```java
Thread 2:
int t;
lock(sum);
sum = sum + y;
t = sum;
...
unlock(sum);
```

Use of semaphores for thread synchronization!

Synchronization - Processes

```
x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8
```

- Step 1: parallelism = 4
- Step 2: parallelism = 2
- Step 3: parallelism = 1

Wait for a message from other processes before continuing processing!
On a single processor machine

- You can have multiple threads
- You can also have multiple processes and have the effect of concurrency
  - timesharing

Data Parallelism

```plaintext
for (i = 1; i<=60; i++)
{
    a (i) = b (i) * c (i);
}
```

Decompose DATA into pieces. All the tasks perform the same computation operating with their own piece of data.
Function Parallelism

- a = b + 1;
- a = a + c;
- a = 2 * a;
- d = 5 * d;
- e = d - 1;
- e = e + 1;
- g = f * 3;
- f = f - 1;
- f = f * 4;

Decompose COMPUTATIONS into pieces (functions). Data are taken to the tasks where they are needed.