Concurrent Programming

Dr A Sahu
Dept of Computer Science & Engineering
IIT Guwahati

Outline
• Course Structure & Book
• Basic of thread and process
• Coordination and synchronization
• Example of Parallel Programming
  – Shared memory: C/C++ Pthread, C++11 thread, OpenMP, Cilk
  – Distributed Memory: MPI
• Concurrent Objects
  – Concurrent Queue, List, stack, Tree, Priority Queue, Hash, SkipList
• Use of Concurrent objects

Course Structure & Book
• Concurrent Programming
  – Threads and processes
  – Synchronization and monitors
  – Concurrent objects
  – Concurrent Programming in Java/MPI/CILK/C++.
• Book
  – Maurice Herlihy, Nir Shavit, Art of Multiprocessor Programming, Elsevier 2009

Concurrent Programming
• Programming to Simulate Concurrent behavior of system
  – Multi-threading
  – Doing many task simultaneously
• Platform of Concurrent Programming
  – May be uni-processor
  – May be shared or distributed memory multiprocessor
• Parallel Programming
  – Enhancing performance of application by running program in parallel on Multiprocessor

Process and Thread
• Process
  – A sequential computation with its own thread of control
  – Can be many threads of a Process
• Thread
  – A sequential computation is the sequence of the program points that are reached as control flow through source text
  – Light weight process
  – Many things shared by parent process

Communications
• Exchange of data between threads/processes
  – Either by explicit message passing
  – Or through the values of shared variable
• Between Process
  – Message passing
  – Message Passing Interface: MPI-send(), MPI_recv()
• Between thread
  – through the values of shared variable
Synchronizations
• Relates the thread of one process with others

If $P$ is point in the thread of a process $P$, and $q$ is point in the thread of another process $Q$.
Then Synchronization can be used to constrain the order in which $P$ reached to $p$ and $Q$ reaches to $q$.
Synchronization Involves: Exchange of control information between processes.

Time Shared and Multiprogramming
• Time shared programs appears to run in parallel
  – Even if it run on uni-processor system
  – Lets go back to Pentium PC, RR Scheduling
• Interrupts (Hardware)
  – Allowed the activity of a central CPU to be synchronized with data channels.
  – If a program $P$ needed to read a card, CPU could initiate the read action on a data channel and start executing other program $Q$. Once the card had been read, the channel sent INT to CPU to resume execution of $P$

Time Shared and Multiprogramming
• Reactive System: Potential for parallelism occurs in system
  – User Interface: KBD, Mice and Display supporting multiple window
  – Network, Game, Processor controller

Implicit Synchronization Example
• No need to specify
• Process networks in Unix (Pipe)
  $P_1 | P_2 | \ldots | P_n$
  – Each primitive process does a simple job, perhaps a trivial job
  – but short pipeline of processes can do what would otherwise done by substantial program
• Example
  $\$ bc | number | speak
  $\$ ls | wc -l
  $\$ ps -A | grep mozilla

Concurrency as Interleaving
• Concurrent computation
  – Can be described in terms of events, where an event is an un-interruptible action
  – Event: execution of assignment, call, expr evaluation

Concurrency as Interleaving
• Interleaving: The relative order of atomic events
  – An interleaving of two sequence $S$ and $T$ is any sequence $U$ formed from the events of $S$ and $T$
  – Subjected to constraints: events of $S$ retain their order in $U$ and so the event of $T$
• Example: $S=\{a,b,c,d,\ldots\}$, $T=\{1,2,3,4,\ldots\}$
  – One $U$ can be $\{1,a,b,c,d,2,3,4,e,5,f,\ldots\}$
Basic Coordination and Synchronization

• Sharing Data
  - Reader and Writer
  - 1R, 1W, MR, 1R1W, MR1W, 1RMW, MRMW
  - Synchronization necessary: One process should be writer
  - Mutual Exclusion: Critical Section Problem
• Barrier or Fence
  - Wait until some thing
  - Synchronized
  - Example: Phase wise executions

Sharing Data: Critical Section

• Sharing Data: Reader and Writer
• Locking and unlocking
  - Mutex
• Hardware Instruction to ensure locking
  - Atomic Instructions: TAS, LL/LD pair, XHNG, SWAP
  - TAS (test and set)
  - TTAS (try, test and set)
  - TTAS with Backup

C/C++ Pthread

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

void *thr_func ( void *ptr )
{
  char *message;
  message = (char *) ptr;
  printf("%s \n", message);
}

int main() {
  pthread_t thr1, thr2;
  const char *MSG1="Thr 1", *MSG2="Thr 2";
  int iret1, iret2;
  iret1 = pthread_create( &thr1, NULL, thr_func, (void*) MSG1);
  iret2 = pthread_create( &thr2, NULL, thr_func, (void*) MSG2);
  pthread_join(thr1, NULL);
  pthread_join(thr2, NULL);
  exit(0);
}

$ g++ -pthread pthread1.c –o pthread1

#define NTH 10
pthread_mutex_t M = 0;

void SimpleCnt() {
  for(int i=0;i<10;i++) counter++;
}

void MutexCnt() {
  for(int i=0;i<10;i++)
  {
    pthread_mutex_lock( &M );
    counter++;
    pthread_mutex_unlock( &M );
  }
}

int counter = 0;
int main() {
  thread_t th[NTH];
  int i, j;
  for(i=0; i < NTH; i++) {
    pthread_create(&th[i], NULL, SimpleCnt, NULL );
  }
  for(j=0; j < NTH; j++) {
    pthread_join( th[j], NULL);
  }
  printf("Final counter value: %d\n", counter);
}

Shared Variable: SimpleCnt

Shared Variable
int counter = 0;
int main() {
    thread_t th[NTH];
    int i, j;
    for(i=0; i < NTH; i++) {
        pthread_create(&th[i], NULL, MutexCnt, NULL);
    }
    for(j=0; j < NTH; j++) {
        pthread_join(th[j], NULL);
    }
    printf("Final counter value: %d\n", counter);
}

Shared Variable: MutexCnt
...lock introduces sequential bottleneck

Introduction to LOCK (Basic Spin-Lock)
...lock suffers from contention

Basic Spin-Lock
...lock suffers from contention
Notice: these are distinct phenomena

CS
Resets lock upon exit

Seq Bottleneck → no parallelism
Basic Spin-Lock

...lock suffers from contention

Contention \(\rightarrow\) ???

Review: Test-and-Set

- Boolean value
- Test-and-set (TAS)
  - Swap `true` with current value
  - Return value tells if prior value was `true` or `false`
- Can reset just by writing `false`
- TAS aka “getAndSet”

Review: Test-and-Set

```java
import java.util.concurrent.atomic;

public class AtomicBoolean {
    boolean value;

    public synchronized boolean getAndSet(boolean newValue) {
        boolean prior = value;
        value = newValue;
        return prior;
    }
}
```

Swap old and new values

Test-and-Set Locks

- Locking
  - Lock is free: value is false
  - Lock is taken: value is true
- Acquire lock by calling TAS
  - If result is false, you win
  - If result is true, you lose
- Release lock by writing false

Test-and-set Lock

```java
class TASlock {
    AtomicBoolean state =
        new AtomicBoolean(false);

    void lock() {
        while (state.getAndSet(true)) {}
    }

    void unlock() {
        state.set(false);
    }
}
```

Keep trying until lock acquired

Graph

No speedup

Because of sequential bottleneck
Mystery #1

![Diagram](image)

**Test-and-Test-and-Set Locks**

- **Lurking stage**
  - Wait until lock “looks” free
  - Spin while read returns true (lock taken)
- **Pouncing state**
  - As soon as lock “looks” available
  - Read returns false (lock free)
  - Call TAS to acquire lock
  - If TAS loses, back to lurking

Mystery #2

```java
class TTASlock {
    AtomicBoolean state = new AtomicBoolean(false);
    void lock() {
        Then try to acquire it
        while (true) {
            while (state.get()) {}
            if (!state.getAndSet(true))
                return;
        }
    }
}
```

Mystery

- **Both**
  - TAS and TTAS
  - Do the same thing (in our model)
- **Except that**
  - TTAS performs much better than TAS
  - Neither approaches ideal
- **Approach : Similar to CSMA BUS protocol**
  - If many people are waiting for shared lock/Lock is busy. Let me wait for some time then try
  - Waiting time may be fixed or increased exponentially.

Exponential Backoff Lock

```java
public class Backoff implements lock {
    public void lock() {
        int delay = MIN_DELAY;
        while (true) {
            while (state.get()) {}
            if (!lock.getAndSet(true))
                return;
            sleep(random() % delay);
            if (delay < MAX_DELAY)
                delay = 2 * delay;
        }
    }
}
```
**Spin-Waiting Overhead**

![Spin-Waiting Overhead Diagram](image)

- **TTAS Lock**
- **Backoff lock**

**Parallel Programming**

- Shared memory
  - Pthread, C++11 thread
  - Java
  - OpenMP
  - Cilk
- Distributed Memory
  - MPI

**Example: C++11 concurrency**

- Asynchronous tasks and threads
- Promises and tasks
- Mutexes and condition variables
- Atomics

**Spawning asynchronous tasks**

- Two ways: `std::async` and `std::thread`
- It’s all about things that are **Callable**: Functions and Member functions
  - Functions and Member functions
  - Objects with `operator()` and Lambda functions //anonymous function

**Hello World with `std::async`**

```cpp
#include <future> // for std::async
#include <iostream>

void write_message(std::string const& message) {
    std::cout << message;
}

int main() {
    auto f = std::async(write_message,
        "hello world from std::async\n");
    write_message("hello world from main\n");
    f.wait();
}
```

```bash
$ g++ -std=c++0x -pthread test.cpp
```

**Hello World with `std::thread`**

```cpp
#include <thread> // for std::thread
#include <iostream>

void write_message(std::string const& message) {
    std::cout << message;
}

int main() {
    std::thread t(write_message,
        "hello world from std::thread\n");
    write_message("hello world from main\n");
    t.join();
}
```
async : Launch Policies

• `std::launch::async` => “as if” in a new thread.
• `std::launch::deferred` => executed on demand.
• `std::launch::async | std::launch::deferred` => implementation chooses (default).

std::launch::async

```cpp
#include <future>
#include <iostream>

void write_message(std::string const& message) {
    std::cout << message;
}

int main() {
    auto f = std::async(
        std::launch::async,
        write_message,
        "hello world from std::async\n";
    write_message("hello world from main\n")
    f.wait();
}
```

Returning values with std::async

```cpp
#include <future>
#include <iostream>

int find_the_answer() { return 42; }

int main() {
    auto f = std::async(find_the_answer);
    std::cout << "the answer is" << f.get() << "\n";
}
```

Java threading...

```java
class NewThread implements Runnable {
    Thread t;
    NewThread() {
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: "+ t);
        t.start(); // Start the thread
    }

    public void run() {
        for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: "+ i);
        }
    }
}
```

Java threading...

```java
public class ThreadDemo {
    public static void main(String args[]) {
        new NewThread();
        System.out.println("Main Thread: "+ i);
    }
}
```

In C/C++/Java Thread

• We need to indentify parallelism
  – How to do extract parallelism manually
  – Parallel Decomposition
• Code in threaded model
• OS is responsible for running it efficiently
  – Less control over runtime