COMP3031: Parallel and Concurrent Programming

Qiong Luo

HKUST

Concurrency can happen at different levels:

- Instruction level - executing two ore more machine instructions simultaneously.
- Statement level.
- Unit level - executing two or more subprograms simultaneously.
- Program level - executing two or more programs simultaneously.

and can occur either

- physically on separate processors or
- logically through a single processor.

Motivations for Studying Concurrency

- Concurrency is a natural way to conceptualize and solve many problems
  - Concurrency is ubiquitous, and many problems lend themselves naturally to concurrency in much the same way that recursion is a natural way to solve a problem.
  - This is particularly true in simulation systems: the systems being simulated contain many independent entities.
  - Think of cars in a traffic simulation system, airplanes in a air-control system,...
- Computing devices with multiple processors are becoming more common, and we need software to make effective use of these devices.

Subprogram-Level Concurrency

Basic Concepts:

- Task or process: a piece of code that can be run concurrently.
  - Heavyweight tasks: have their own address space and run-time stacks.
  - Lightweight tasks: have to share the same address space on the machine/processor where they are run.
  - States of a process: new; runnable or ready; running; blocked; dead.
- Synchronization: a mechanism that controls the order in which tasks execute
  - Cooperation synchronization: task A's job depends on task B.
  - Competition synchronization: task A and B compete for a same resource.
- Scheduler: decides which task to run (where there are more ready tasks than processors).

Examples

- A global variable x, and two tasks - A: x = x++; B: x = 2x. How many possible results if they are performed concurrently:
- Producer-consumer problem: a number of producers produce some products and put them into a queue, and a number of consumers take them to consume from the queue. How should they be synchronized.

Semaphores

Semaphores are special data structures devised by Dijkstra in 1965 for synchronizing concurrent tasks. In modern terms, one could think of them as a class with the following components:

- A private data member counter of int type, and a queue of tasks;
- two public methods:
  - wait - if `counter > 0` then `counter--` else put the caller in the queue.
  - release - if the queue is empty then `counter++` else dequeue a waiting task.

Semaphores: Cooperation Synchronization

Allow simultaneous access to the VALUE queue:

```
semaphore fullspots, emptyspots;
fullspots.counter = 0;
emptyspots.counter = BUFLEN;
task producer;
  while (true) {
   -- produce VALUE --
   wait(emptyspots);
   DEPOSIT(VALUE);
   release(fullspots); }
task consumer;
  while (true) {
   wait(fullspots);
   FETCH(VALUE);
   release(emptyspots);
   -- consume VALUE -- }
```

Semaphores: Competition Synchronizations

Only one task can access VALUE queue at any time:

```
semaphore fullspots, emptyspots, access;
fullspots.counter = 0;
emptyspots.counter = BUFLEN;
access.counter = 1;
task producer;
  while (true) {
   -- produce VALUE --
   wait(emptyspots);
   wait(access); // wait for access to the queue
   DEPOSIT(VALUE);
   release(access); //relinguish access
   release(fullspots); }
```

```
task consumer;
  while (true) {
   wait(fullspots);
   wait(access);
   FETCH(VALUE);
   release(access);
   release(emptyspots);
   -- consume VALUE -- }
```

Java Threads

- In Java, tasks are called threads. They are lightweight tasks, meaning that they can only be run on the same machine.
- Java has a build-in class called Thread, from which user-defined threads are derived.
- Threads are initiated by "start()" method, which in turn calls the "run()" method.
- start() returns immediately, but run() is the life-span of the thread.
- Some other methods in Thread:
  - yield() - voluntarily surrender the execution of the thread;
  - sleep(n) - suspend the thread for n milliseconds;
  - thr.join(n) - wait for n millisecond for the thread thr to complete;

Competition Synchronization

In Java, competition synchronization is done by "synchronized methods" or "synchronized statements": in the following, "synchronized" is a reserved word

- "synchronized method()" locks the object until the method is finished;
- during the execution of method(), no other synchronized method is allowed to access the object.

```
class ManageBuf {
  ...
  public synchronized void deposit(int item) { ... }
  public synchronized int fetch() { ... }
  ... }
```

- "synchronized(expression) statement" locks the object specified by "expression" when "statement" is executed.

Cooperation Synchronization

- Cooperation synchronization in Java is done with "wait", "notify", and "notifyAll" methods in Object - the root class of all Java classes.
- Every object has a wait list of all the threads that have called "wait" on that object.
- The "notify" method is called to free one of the waiting threads, randomly.
- The "notifyAll" method awakens all the threads on the object's waiting list.
- The "wait" method can throw "InterruptedException", so any code that uses "wait" must also catch this exception:

```
try {
  while (!condition)
    wait();
  -- codes after 'condition' becomes true
} catch (InterruptedException myProblem) { ... }
```

Producer-Consumer in Java

```
//A circular queue for storing int values with synchronized
//methods for inserting and removing values from the queue
class Queue {
  private int [] que;
  private int nextIn, nextOut, filled, queSize;

  public Queue(int size) { que = new int [size];
    filled = 0; nextIn = 0; nextOut = 0; queSize = size; }

  public synchronized void deposit (int item) {
    try {
      while (filled == queSize) wait();
      que [nextIn] = item; nextIn = (nextIn++ % queSize);
      filled++;  notifyAll(); }
    catch (InterruptedException e) {} }
```

Producer-Consumer in Java (Cont'd)

```
  public synchronized int fetch ( ) {
    int item = 0;
    try { while (filled == 0) wait();
          item=que[nextOut]; nextOut=(nextOut++ % queSize);
          filled--;  notifyAll(); }
    catch (InterruptedException e) {}
    return item; }
 } // end of Queue class
class Producer extends Thread {
  private Queue buffer;
  public Producer(Queue que) { buffer = que; }
  public void run() {
    int new_item;
    while (true) { //-- create a new_item
                   buffer.deposit(new_item); }}}
```

Producer-Consumer in Java (Cont'd)

```
class Consumer extends Thread {
  private Queue buffer;
  public Consumer(Queue que) { buffer = que; }
  public void run() {
    int stored_item;
    while (true) {
      stored_item = buffer.fetch();
      //-- consume stored_item
    }}}

Queue buff1 = new Queue(100);
Producer prod1 = new Producer(buff1);
Consumer cons1 = new Consumer(buff1);
prod1.start(); cons1.start();
```

Statement-Level Concurrency

- Statement-level concurrency is used to inform the compiler of ways it can map the program onto a multiprocessor machine.
- CUDA = a programming language for GPU - later.
- HPF - high-performance Fortran (1993), an extension of Fortran 90 to include statements on how concurrent programs should be run on a multiprocessor machine.
- These include specifications about the number of processors, statements about how data should be distributed over the memories of these processors, and the alignments of data.

HPF Statements

- `!HPF$ PROCESSOR procs (n)` specifies there are n processors available.
- `!HPF$ DISTRIBUTE (kind) ONTO procs :: lists`:
  - Here 'kind' can be either BLOCK or CYCLIC, and 'lists' is the names of the array variables that are to be distributed.
  - BLOCK means that each array is to be divided into n (the number of processors) groups consisting of contiguous collections of array elements, and distributed over the memories of the n processors. Example: a 500 element array BLOCK distribute over 5 processors means that the first 100 elements on the first processor, the second 100 elements on the second processor, etc..
  - CYCLIC means to distribute the 1st element on the 1st processor, 2nd element on the 2nd processor, ..., nth element on the nth processor, (n+1)th element on the 1st processor, ...
- `!HPF$ ALIGN list1(index) WITH list2(index+1)` says that for all values of index, store list1(index) and list2(index+1) in the memory of the same processor.

Example

```
REAL list1 (1000), list2(1000)
INTEGER list3(500), list4(501)
!HPF$ PROCESSOR procs (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: list1, list2
!HPF$ ALIGN list3(index) WITH list4(index+1)
FORALL (index = 1:1000) list1(index) = list2(index)+1.0
FORALL (index = 1:500) list3(index) = list4(index+1)
```

FORALL statement specifies a collection of statements to be executed concurrently.