

# 1. Introduction to Concurrent Programming

A *concurrent program* contains two or more threads that execute concurrently and work together to perform some task.

When a program is executed, the operating system creates a *process* containing the code and data of the program and manages the process until the program terminates.

Per-process state information:

- the process' state, e.g., ready, running, waiting, or stopped
- the program counter, which contains the address of the next instruction to be executed for this process
- saved CPU register values
- memory management information (page tables and swap files), file descriptors, and outstanding I/O requests

Multiprocessing operating systems enable several programs/processes to execute simultaneously.

A *thread* is a unit of control within a process:

- when a thread runs, it executes a function in the program - the “main thread” executes the “main” function and other threads execute other functions.
- per-thread state information: stack of activation records, copy of CPU registers (including stack pointer, program counter)
- threads in a multithreaded process share the data, code, resources, address space, and per-process state information

The operating system allocates the CPU(s) among the processes/threads in the system:

- the operating system selects the process and the process selects the thread, or
- the threads are scheduled directly by the operating system.

In general, each ready thread receives a time-slice (called a *quantum*) of the CPU.

- If a thread waits for something, it relinquishes the CPU.
- When a running thread's quantum has completed, the thread is *preempted* to allow another ready thread to run.

Switching the CPU from one process or thread to another is known as a *context switch*.

multiple CPUs  $\Rightarrow$  multiple threads can execute at the same time.

single CPU  $\Rightarrow$  threads take turns running

The *scheduling policy* may also consider a thread's priority. We assume that the scheduling policy is *fair*, which means that every ready thread eventually gets to execute.

## 1.2 Advantages of Multithreading

- Multithreading allows a process to overlap IO and computation.
- Multithreading can make a program more responsive (e.g., a GUI thread can respond to button clicks while another thread performs some task in the background).
- Multithreading can speedup performance through parallelism.
- Multithreading has some advantages over multiple processes:
  - threads require less overhead to manage than processes
  - intra-process thread communication is less expensive than inter-process communication (IPC).

Multi-process concurrent programs do have one advantage: each process can execute on a different machine. This type of concurrent program is called a *distributed program*.

Examples of distributed programs: file servers (e.g., NFS), file transfer clients and servers (e.g., FTP), remote login clients and servers (e.g., Telnet), groupware programs, and web browsers and servers.

The main disadvantage of concurrent programs is that they are difficult to develop. Concurrent programs often contain bugs that are notoriously hard to find and fix.

## 1.3 Threads in Java

(See Chapter1JavaThreads.pdf)

## 1.4 Threads in Win32

(See Chapter1Win32Threads.pdf)

## 1.5 Pthreads

(See Chapter1PthreadsThreads.pdf)

## 1.6 A C++ *Thread* Class

Details about Win32 and POSIX threads can be encapsulated in a C++ *Thread* class.

### 1.6.1 C++ Class *Thread* for Win32

(See Chapter1Win32ThreadClass.pdf)

### 1.6.2 C++ Class *Thread* for Pthreads

(See Chapter1PthreadThreadClass.pdf)

## 1.7 Thread Communication

In order for threads to work together, they must communicate.

One way for threads to communicate is by accessing shared memory. Threads in the same program can reference global variables or call methods on a shared object.

Listing 1.8 shows a C++ program in which the *main* thread creates two *communicatingThreads*. Each *communicatingThread* increments the global shared variable *s* ten million times. The *main* thread uses *join()* to wait for the *communicatingThreads* to complete, then it displays the final value of *s*.

We executed this program fifty times. In forty-nine of the executions, the value 20000000 was displayed, but the value displayed for one of the executions was 19215861.

Concurrent programs exhibit non-deterministic behavior - two executions of the *same* program with the *same* input can produce *different* results.

```

int s=0; // shared variable s

class communicatingThread: public Thread {
public:
    communicatingThread (int ID) : myID(ID) {}
    virtual void* run();
private:
    int myID;
};

void* communicatingThread::run() {
    std::cout << "Thread " << myID << " is running" << std::endl;
    for (int i=0; i<10000000; i++) // increment s ten million times
        s = s + 1;
    return 0;
}

int main() {
    std::auto_ptr<communicatingThread> thread1(new communicatingThread (1));
    std::auto_ptr<communicatingThread> thread2(new communicatingThread (2));
    thread1->start(); thread2->start();
    thread1->join(); thread2->join();
    std::cout << "s: " << s << std::endl; // expected final value of s is 20000000
    return 0;
}

```

Listing 1.8 Shared variable communication.



*Non-deterministic results do not necessarily indicate the presence of an error.* Threads are frequently used to model real-world objects and the real-world is non-deterministic.

*Non-determinism adds flexibility to a design.* A bounded buffer compensates for differences in the rates at which the producers and consumers work. The order of deposits and withdraws is non-deterministic.

*Non-determinism and concurrency are related concepts.* Concurrent events A and B can be modeled as a non-deterministic choice between two interleavings of events: (A followed by B) or (B followed by A). But the possible number of interleavings explodes as the number of concurrent events increases.

*Non-determinism is an inherent property of concurrent programs.* The burden of dealing with non-determinism falls on the programmer, who must ensure that threads are correctly synchronized without imposing unnecessary constraints that only reduce the level of concurrency.

*Non-deterministic executions create major problems during testing and debugging.*

### **1.7.2 Atomic Actions**

A program's state contains a value for each variable defined in the program and other implicit variables, like the program counter.

An atomic action transforms the state of the program, and the state transformation is indivisible:  $\{x==0\} x=1; \{x==1\}$ .

A state transformation performed during an atomic action is indivisible if other threads can see the program's state as it appears before the action or after the action, but not some intermediate state while the action is occurring.

A context switch may occur while one thread is performing an atomic action as long as we don't allow the other threads to see or interfere with the action while it is in progress.

Individual machine instructions such as *load*, *add*, *subtract* and *store* are typically executed atomically; this is guaranteed by the memory hardware.

In Java, an assignment of 32 bits or less is guaranteed to be implemented atomically, so an assignment statement such as  $x = 1$  for a variable  $x$  of type *int* is an atomic action. In general, however, the execution of an assignment statement may not be atomic.

**1.7.2.1 Non-atomic arithmetic expressions and assignment statements.** An interleaving of the machine instructions from two or more expressions or assignment statements may produce unexpected results.

Example 3. Assume that  $y$  and  $z$  are initially 0.

| <u>Thread1</u> | <u>Thread2</u> |
|----------------|----------------|
| $x = y + z;$   | $y = 1;$       |
|                | $z = 2;$       |

Assume (incorrectly) that each statement is an atomic action, what are the expected final values?

The machine instructions for *Thread1* and *Thread2*:

| <u>Thread1</u>  | <u>Thread2</u>        |
|-----------------|-----------------------|
| (1) load r1, y  | (4) assign y, 1       |
| (2) add         | r1, z (5) assign z, 2 |
| (3) store r1, x |                       |

Possible interleavings:

- (1), (2), (3), (4), (5)  $\Rightarrow$  x is 0
- (4), (1), (2), (3), (5)  $\Rightarrow$  x is 1
- (1), (4), (5), (2), (3)  $\Rightarrow$  x is 2 \*
- (4), (5), (1), (2), (3)  $\Rightarrow$  x is 3

Example 4. Assume that the initial value of  $x$  is 0.

|                |                |
|----------------|----------------|
| <u>Thread1</u> | <u>Thread2</u> |
| $x = x + 1;$   | $x = 2;$       |

The machine instructions for *Thread1* and *Thread2* are:

|                 |                 |
|-----------------|-----------------|
| <u>Thread1</u>  | <u>Thread2</u>  |
| (1) load r1, x  | (4) assign x, 2 |
| (2) add r1, 1   |                 |
| (3) store r1, x |                 |

Possible interleavings:

- (1), (2), (3), (4)  $\Rightarrow$  x is 2
- (4), (1), (2), (3)  $\Rightarrow$  x is 3
- (1), (2), (4), (3)  $\Rightarrow$  x is 1 \*

If there are  $n$  threads ( $Thread_1, Thread_2, \dots, Thread_n$ ) such that  $Thread_i$  executes  $m_i$  atomic actions, then the number of possible interleavings of the atomic actions is:

$$(m_1 + m_2 + \dots + m_n)! / (m_1! * m_2! * \dots * m_n!)$$

Andrews defined a condition called At-Most-Once under which expression evaluations and assignments will appear to be atomic. A critical reference in an expression is a reference to a variable that is changed by another thread.

*At-Most-Once:*

- An assignment statement  $x = e$  satisfies the At-Most-Once property if either:
  - (1)  $e$  contains at most one critical reference and  $x$  is neither read nor written by another thread, or
  - (2)  $e$  contains no critical references, in which case  $x$  may be read or written by other threads.
- An expression that is not in an assignment satisfies At-Most-Once if it contains no more than one critical reference.

“There can be at most one shared variable and the shared variable can be referenced at most one time. “

Assignment statements that satisfy At-Most-Once *appear* to execute atomically even though they are not atomic.

Example 5. Assume that  $x$  and  $y$  are initially 0.

|                |                |
|----------------|----------------|
| <u>Thread1</u> | <u>Thread2</u> |
| $x = y + 1;$   | $y = 1;$       |

- The expression in *Thread1*'s assignment statement references  $y$  (one critical reference) but  $x$  is not referenced by *Thread2*,
- The expression in *Thread2*'s assignment statement has no critical references.

Both statements satisfy At-Most-Once.

**1.7.2.2 Non-atomic groups of statements.** Another type of undesirable non-determinism in a concurrent program is caused by interleaving groups of statements, even though each statement may be atomic.

Example 5. Variable *first* points to the first *Node* in the list. Assume the list is !empty

```
class Node {
public:
    valueType value;
    Node* next;
}
Node* first;           // first points to the first Node in the list;
void deposit(valueType value) {
    Node* p = new Node; // (1)
    p->value = value;   // (2)
    p->next = first;    // (3)
    first = p;         // (4) insert the new Node at the front of the list
}
valueType withdraw() {
    valueType value = first->value; // (5) withdraw the first value in the list
    first = first->next;           // (6) remove the first Node from the list
    return value;                 // (7) return the withdrawn value
}
```

The following interleaving of statements is possible:

```
valueType value = first->value; // (5) in withdraw
Node* p = new Node(); // (1) in deposit
p->value = value // (2) in deposit
p->next = first; // (3) in deposit
first = p; // (4) in deposit
first = first->next; // (6) in withdraw
return value; // (7) in withdraw
```

At the end of this sequence:

- withdrawn item is still pointed to by *first*
- deposited item has been lost.

To fix this problem, each of methods *deposit* and *withdraw* must be implemented as an atomic action.

## 1.8 Testing and Debugging Multithreaded Programs

The purpose of testing is to find program failures.

A *failure* is an observed departure of the external result of software operation from software requirements or user expectations [IEEE90]. Failures can be caused by hardware or software faults or by user errors.

A software *fault* (or defect, or bug) is a defective, missing, or extra instruction or a set of related instructions, that is the cause of one or more actual or potential failures [IEEE 88].

*Debugging* is the process of locating and correcting faults.

The conventional approach to testing and debugging a sequential program:

- (1) Select a set of test inputs
- (2) Execute the program once with each input and compare the test results with the intended results.
- (3) If a test input finds a failure, execute the program *again* with the same input in order to collect debugging information and find the fault that caused the failure.
- (4) After the fault has been located and corrected, execute the program *again* with each of the test inputs to verify that the fault has been corrected and that, in doing so, no new faults have been introduced (a.k.a “regression testing”).

This process breaks down when it is applied to concurrent programs.

### 1.8.1 Problems and Issues

Let CP be a concurrent program. Multiple executions of CP with the *same* input may produce *different* results. This non-deterministic execution behavior creates the following problems during the testing and debugging cycle of CP:

Problem 1. When testing CP with input X, a single execution is insufficient to determine the correctness of CP with X. Even if CP with input X has been executed successfully many times, it is possible that a future execution of CP with X will produce an incorrect result.

Problem 2. When debugging a failed execution of CP with input X, there is no guarantee that this execution will be repeated by executing CP with X.

Problem 3. After CP has been modified to correct a fault detected during a failed execution of CP with input X, one or more successful executions of CP with X during regression testing do not imply that the detected fault has been corrected or that no new faults have been introduced.

There are many issues that must be dealt with in order to solve these problems:

*Program Replay:* Programmers rely on debugging techniques that assume program failures can be reproduced. Repeating an execution of a concurrent program is called “program replay”.

*Program Tracing:* Before an execution can be replayed it must be traced. But what exactly does it mean to replay an execution?

- If a C++ program is executed twice and the inputs and outputs are the same for both executions, are these executions *identical*? Are the context switches always on the same places? Does this matter?
- Now consider a concurrent program. Are the context switches among the threads in a program important? Must we somehow trace the points at which the context switches occur and then repeat these switch points during replay?
  - What should be replayed?
  - Consider the time and space overhead for capturing and storing the trace.
  - In a distributed system, there is an “observability problem”: it is difficult to accurately observe the order in which actions on different computers occur during an execution.

*Sequence Feasibility*: A sequence of actions that is allowed by a program is said to be a *feasible* sequence. Testing involves determining whether or not a given sequence is feasible or infeasible. “Good” sequences are expected to be feasible while “bad” sequences are expected to be infeasible. How are sequences selected?

- Allow sequences to be exercised non-deterministically (*non-deterministic testing*). Can non-deterministic testing be used to show that bad sequences are infeasible?
- Force selected sequences to be exercised (*deterministic testing*). Choosing sequences that are effective for detecting faults is hard to do. A *test coverage criterion* can be used to guide the selection of tests and to determine when to stop testing.

*Sequence Validity*: Sequences allowed by the specification, i.e. “good” sequences, are called *valid* sequences; other sequences are called *invalid* sequences. A goal of testing is to find valid sequences that are infeasible and invalid sequences that are feasible.

*The Probe Effect:* Modifying a concurrent program to capture a trace of its execution may interfere with the normal execution of the program.

- The program will behave differently after the trace routines have been added. (But without this interference, it is possible that some failures cannot be observed.)
- Some failures that would have been observed without adding the trace routines will no longer be observed.

One way to address the probe effect is to make sure that every feasible sequence is exercised at least once during testing. Reachability testing can exercise all of the feasible sequences of a program, when the number of sequences is not “too large”.

Three different problems:

- Observability problem: the difficulty of accurately tracing a given execution,
- Probe effect: the ability to perform a given execution at all.
- Replay problem: repeating an execution that has already been observed.

*Real-Time:* The probe effect is a major issue for real-time concurrent programs. The correctness of a real-time program depends not only on its logical behavior, but also on the time at which its results are produced.

For an example of program tracing and replay, we’ve modified the C++ program in Listing 1.8 so that it can trace and replay its own executions. The new program is shown in Listing 1.9.

### **1.8.2 Class *TDThread* for Testing and Debugging**

Threads in Listing 1.9 are created by inheriting from C++ class *TDThread* instead of class *Thread*. Class *TDThread* provides the same interface as our C++ *Thread* class, plus several additional functions that are used internally during tracing and replay.

```

sharedVariable<int> s(0); // shared variable s

class communicatingThread: public TDThread {
public:
    communicatingThread(int ID) : myID(ID) {}
    virtual void* run();
private:
    int myID;
};

void* communicatingThread::run() {
    std::cout << "Thread " << myID << " is running" << std::endl;
    for (int i=0; i<2; i++)    // increment s two times (not 10 million times)
        s = s + 1;
    return 0;
}

int main() {
    std::auto_ptr<communicatingThread> thread1(new communicatingThread (1));
    std::auto_ptr<communicatingThread> thread2(new communicatingThread (2));
    thread1->start(); thread2->start();
    thread1->join(); thread2->join();
    std::cout << "s: " << s << std::endl;    // the expected final value of s is 4
    return 0;
}

```

Listing 1.9 Using classes *TDThread* and *sharedVariable<>*.

The main purpose of class *TDThread* is to automatically and unobtrusively generate an integer identifier (ID) and a name for each thread.

- Thread IDs are based on the order in which threads are constructed during execution. The first thread constructed is assigned ID 1, the second thread is assigned ID 2, and so on.
- Thread names are generated automatically, e.g., `main_thread1`, `main_thread2`, `main_thread1_thread1`, `main_thread1_thread2`. A more descriptive name can be provided by manually supplying a name to *TDThread*'s constructor. For example:

```

class namedThread : public TDThread {
    namedThread() : TDThread("namedThread") {} // "namedThread" is the name
    ...
}

```

User-supplied thread names must be unique, or they will be rejected.

### 1.8.3 Tracing and Replaying Executions with Class Template *sharedVariable*<>

Class template *sharedVariable*<> provides functions that allow every shared variable access to be traced and replayed. Class *sharedVariable*<> provides the usual operators (+, -, =, <, etc) for primitive types. It also traces and replays the read and write operations in the implementations of these operators.

- The IDs generated by class *TDThread* for threads *thread1* and *thread2* are 1 and 2
- In trace mode, class *sharedVariable*<*int*> records the order in which these two threads read and write shared variable *s*.
- Each increment “*s = s+1*” involves a read of *s* followed by a write of *s*, and each thread increments *s* twice.
- The final value of *s* is expected to be 4 under the incorrect assumption that the increments are atomic.

A possible trace for Listing 1.9 is

```
Read(thread1,s)    // thread1 reads s; s is 0
Read(thread2,s)    // thread2 reads s; s is 0
Write(thread2,s)   // thread2 writes s; s is now 1
Write(thread1,s)   // thread1 writes s; s is now 1
Read(thread1,s)    // thread1 reads s; s is 1
Write(thread1,s)   // thread1 writes s; s is now 2
Read(thread2,s)    // thread2 reads s; s is 2
Write(thread2,s)   // thread2 writes s; s is now 3
```

An execution that produces this trace will display the output 3, not 4.

In replay mode, *sharedVariable*<*int*> forces the threads to execute their read and write operations on *s* in the order recorded in the trace, always producing the output 3.

## 1.9 Thread Synchronization

The examples in this chapter showed the failures that can occur when accesses to shared variables are not properly synchronized.

One type of synchronization is called *mutual exclusion*.

Mutual exclusion ensures that a group of atomic actions, called a critical section, cannot be executed by more than one thread at a time. That is, a critical section must be executed as an atomic action.

The increment statements executed by thread1 and thread2 in Listing 1.9 require mutual exclusion to work properly.

Failure to correctly implement critical sections is a fault called a *data race* [Netzer 1992].

Another type of synchronization is called *condition synchronization*.

Condition synchronization ensures that the state of a program satisfies a particular condition before some action occurs.

For the linked list in Example 5 earlier, there is a need for both condition synchronization and mutual exclusion.

- The list must not be in an empty condition when method *withdraw* is allowed to remove an item,
- mutual exclusion is required for ensuring that deposited items are not lost and are not withdrawn more than twice.