Computer Science 507
Software Engineering
The College of Saint Rose
Spring 2014

Topic Notes: Dangers of Concurrency

As programmers here in the multicore era, understanding concurrency is essential. While it is not a major focus of this course, it seems appropriate to have at least a brief discussion of the issues.

Any single program that is going to make use of multiple processing cores at the same time will need to consist of multiple *processes* or *threads of control*. These threads are often introduced and managed by the programmer, though compilers can help in some cases.

Any modern programming language is likely to include support for multithreading, either at the language level or through libraries.

Multicore Programming

Multithreaded programs are well suited to take advantage of the multiple processing cores found in most modern computers. But writing a program to do so correctly and efficiently is challenging.

The following are some challenges that must be overcome:

- Dividing activities – the program needs to be broken down into separate tasks that can be run concurrently.
- Load balance – the tasks need to perform similar amounts of work since the entire program would run only as quickly as the slowest (most heavily loaded) task.
- Data partitioning – the data needed by each task should be associated with that task.
- Data dependency – if the execution of one task depends on a result computed by a second, synchronization must be performed to make sure the result from the second task is available before the first attempts to use it.
- Testing and debugging – execution of multithreaded programs includes tasks that execute concurrently or in different interleavings. It is very difficult to ensure that a program works correctly for all possible interleavings.

These issues will be discussed extensively in the Parallel and High Performance Computing course when it is offered for the first time, hopefully next school year.

For today, we will just look at some examples in Java.

Java includes a class named `Thread` that supports concurrent programming.

Our goal here is mainly to discuss complications that arise with threads and provide some indications of how to handle them.

Interference

A thread activates an extra "brain" for your program. What happens if those brains give your program contradictory messages?

An important class of problems can arise with concurrency when there are several threads that might try to update the same variable at the same time.

Consider an example of a bank with two ATMs which can be used to deposit and withdraw money.

See Example: ATM1

One of the ATMs will repeatedly withdraw $100 from the account while the other will repeatedly deposit $100 in the account (see the difference in parameters in the constructors for the ATM's). When the user pushes the button, the `actionPerformed` method repeats the construction and execution of the ATM objects.

The main items of interest here are the `getBalance` and `setBalance` methods. They do the obvious things.

The `run` method repeatedly deducts `change` from the account by first executing `sharedBank.getBalance(` to get the balance and then executing `sharedBank.setBalance(balance+change,ATM_ID,change)` to update the balance.

The final balance in the account should be $1000, the same amount started with, as one of the ATM objects withdraws $100 ten times, while the other deposits $100 ten times. If you run this code enough, however, you will discover that the answer does not always turn out to be $1000! What is causing the problems? Look at the program to see if you can determine what is going wrong before reading further.

The error occurs because two different threads (objects of type `Thread`) are updating the same variable, `balance`. Each gets the balance from the bank, adds in its change, and then tells the bank the new balance. However, it can happen that both ATMs get the balance before either of them has the opportunity to update the balance.

For example, suppose ATM1 gets the balance of $1000, while ATM2 "simultaneously" gets the balance of $1000 (they aren't actually happening simultaneously if there is only one processor, but for our purposes it can be helpful to think that way). Now ATM1 adds $100 to the balance and updates the balance to $1100. ATM2 then removes $100 from the balance that it originally got ($1000) and updates the balance to $900. Thus if the interleaving of operations of ATM1 and ATM2 are such that both get balances before either registers the new balance, the final balance will not reflect one of the two operations.

Clearly this is a problem, yet we would like to have the operations of the two ATMs interleaved. (We could just run ATM1 to conclusion before starting up ATM2, but this does not model the usage of ATM's properly.)

We would like to ensure that if ATM1 queries the balance with the intent to change it and set a new balance, that ATM2 does not read the original balance. It is when both read the old balance and both update that one of the transactions is lost. We attempt to remedy this by replacing the `setBalance` method with a `changeBalance` method:

See Example: ATM2

Now rather than having separate methods to get and set the balance, we have a single method which takes the amount of change and updates the balance. Because the getting and setting are no longer separated by distinct method calls, the chances of interference are not as great. However there is still the opportunity of interference between the calculation of the new balance and the update of the value. We have artificially increased the chances of this by adding the `pause` between the two.

Even if we remove that, we reduce the time between the calculation of the old balance and setting of the new balance, but still allows the (at least theoretical) possibility of interference between the calculation of `balance+change` and the assignment of that value to `balance`. To be absolutely safe, we must ensure that only one thread at a time can execute the method `changeBalance`. We can do this in Java by using the keyword `synchronized`.

If we attach the keyword `synchronized` to methods in a class, then Java will ensure that only one thread at a time will be executing any of those methods. For example we can label both `getBalance` and `changeBalance` as `synchronized`.

See Example: ATM3

Now if a thread associated with one ATM object is executing either of these methods, then no other thread can execute either of the methods. For example, if `ATM1` is executing `changeBalance`, then `ATM2` will not be allowed to execute either `changeBalance` or `getBalance`. Instead it will wait until `ATM1` has finished executing that method and then execute the desired method. (The operating system is given the responsibility of scheduling the threads' access to the processor.)

A thread executing either `changeBalance` or `getBalance` has no impact on another thread's attempts at executing any of the non-synchronized methods of the program. Thus the user-interface thread can by executing the `actionPerformed` or `startATMs` method while `ATM1` is executing `changeBalance`.

Because of the use of `synchronized`, neither thread can interfere with the other, ensuring that the final answer is the correct one. However, there is one disadvantage of using synchronized methods – they cut down on the amount of concurrency in the system. This may slow down the execution of the program, as one thread may be waiting for an operation to complete (e.g., a write to the screen or a read from a file), while another might be ready to do something. The second thread may be ready to use the processor, but if it is ready to execute a synchronized method and the other thread is executing a synchronized method of the same object, then it may be blocked from executing.

This example may seem a bit contrived – we carefully made sure the pause times for the two ATMs were the same and added a random pause inside the methods that change the balance to increase the chances of interference. However, the interference could happen in each of our cases (except the one with the `synchronized` modifiers) even without the careful attempts to increase the

chances. Has anyone ever had some big program, even maybe a commercial program, crash in an unexpected and non-reproducable manner? Perhaps a browser or even Windows itself? There's a pretty good chance that a lot of those kinds of crashes are the result of concurrency not being dealt with carefully enough. Most of the time, things are fine – but once in a while just the right combination of things is happening and there you go. Crash and burn.

There are many other complications involved in the use of concurrency – too many to go into detail here. Concurrency can be quite challenging, and inattention to details may result in programs that don't work as expected. Most of the programs that we have had you write which involve active objects have been designed so that no interference is possible. We urge you to worry about the possibilities of this happening when you use active objects. More advanced Computer Science courses study concurrency in much more detail, including other problems that may arise and the techniques to deal with them.

Cooperating Processes

An *Independent* process is not affected by other running processes.

Cooperating processes may affect each other, hopefully in some controlled and useful way.

Why cooperating processes?

- information sharing
- computational speedup
- modularity or convenience

It's hard to find a computer system where processes do not cooperate. Consider the commands you type at the Unix command line. Your shell process and the process that executes your command must cooperate. If you use a pipe to hook up two commands, you have even more process cooperation.

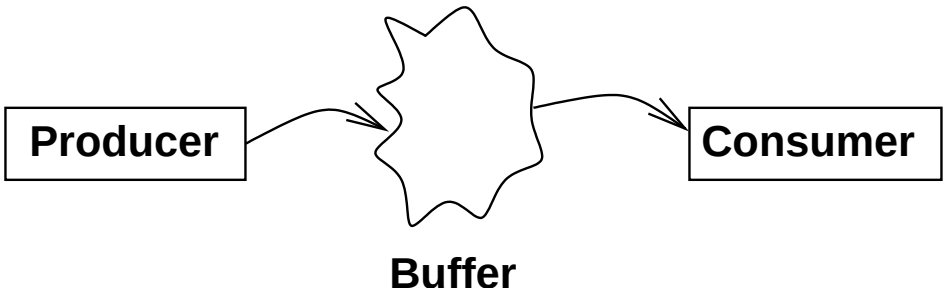
For the processes to cooperate, they must have a way to communicate with each other. Two common methods:

- shared variables – some segment of memory which is accessible to both processes
- message passing – a process sends an explicit message that is received by another

We will consider a shared-memory communication.

Producer-Consumer Problem

The classic example for studying cooperating processes is the Producer-Consumer problem.



One or more produces processes is "producing" data. This data is stored in a buffer to be "consumed" by one or more consumer processes.

The buffer may be:

- *unbounded* – We assume that the producer can continue producing items and storing them in the buffer at all times. However, the consumer must wait for an item to be inserted into the buffer before it can take one out for consumption.
- *bounded* – The producer must also check to make sure there is space available in the buffer.

Bounded Buffer, buffer size n

For simplicity, we will assume the objects being produced and consumed are `int` values.

This solution leaves one buffer entry empty at all times:

- Shared data

```
int buffer[n];
int in=0;
int out=0;
```

- Producer process

```
while (1) {
  ...
  produce item;
  ...
  while (((in+1)%n) == out); /* busy wait */
  buffer[in]=item;
  in=(in+1)%n;
}
```

- Consumer process

```
while (1) {
  while (in==out); /* busy wait */
  item=buffer[out];
  out=(out+1)%n;
  ...
  consume item;
  ...
}
```

Is there any danger with this solution in terms of concurrency? Remember that these processes can be interleaved in any order – the system could preempt the producer at any time and run the consumer.. Things to be careful about are shared references to variables.

Note that only one of the processes can *modify* the variables `in` and `out`. Both use the values, but only the producer modifies `in` and only the consumer modifies `out`. Try to come up with a situation that causes incorrect behavior – hopefully you cannot.

Perhaps we want to use the entire buffer...let's add a variable to keep track of how many items are in the buffer, so we can tell the difference between an empty and a full buffer:

- Shared data

```
int buffer[n];
int in=0;
int out=0;
int counter=0;
```

- Producer process

```
while (1) {
  ...
  produce item;
  ...
  while (counter==n); /* busy wait */
  buffer[in]=item;
  in=(in+1)%n;
  counter=counter+1;
}
```

- Consumer process

```
while (1) {
  while (counter==0); /* busy wait */
  item=buffer[out];
```

```
    out=(out+1)%n;
    counter=counter-1;
    ...
    consume item;
    ...
  }
```

We can now use the entire buffer. However, there is a potential danger here. We modify `counter` in both the producer and the consumer.

Everything looks fine, but let's think about how a computer actually executes those statements to increment or decrement `counter`.

`counter++` really requires three machine instructions: (i) load a register with the value of `counter`'s memory location, (ii) increment the register, and (iii) store the register value back in `counter`'s memory location. There's no reason that the operating system can't switch the process out in the middle of this.

Consider the two statements that modify `counter`:

	Producer		Consumer
P_1	`R0 = counter;`	C_1	`R1 = counter;`
P_2	`R0 = R0 + 1;`	C_2	`R1 = R1 - 1;`
P_3	`counter = R0;`	C_3	`counter = R1;`

Consider one possible ordering: $P_1\ P_2\ C_1\ P_3\ C_2\ C_3$, where `counter=17` before starting. Uh oh.

What we have here is a *race condition*.

You may be thinking, "well, what are the chances, one in a million that the scheduler will choose to preempt the process at exactly the wrong time?"

Doing something millions or billions of times isn't really that unusual for a computer, so it would come up..

Some of the most difficult bugs to find in software (often in operating systems) arise from race conditions.

This sort of interference comes up in painful ways when "real" processes are interacting.

Consider two processes modifying a linked list, one inserting and one removing. A context switch at the wrong time can lead to a corrupted structure:

```
struct node {
  ...
  struct node *next;
}

struct node *head, *tail;
```

```
void insert(val) {
  struct node *newnode;

  newnode = getnode();
  newnode->next = NULL;
  if (head == NULL){
     head = tail = newnode;
   } else {     //  <==== THE WRONG PLACE
     tail->next = newnode;
     tail = newnode;
   }
}

void remove() {
  // ... code to remove value ...
  head = head->next;
  if (head == NULL) tail = NULL;
  return (value);
}
```

If the process executing insert is interrupted at "the wrong place" and then another process calls remove until the list is empty, when the insert process resumes, it will be operating on invalid assumptions and the list will be corrupted.

In the bounded buffer, we need to make sure that when one process starts modifying `counter`, that it finishes before the other can try to modify it. This requires *synchronization* of the processes.

Process synchronization is one of the major topics of an OS, and one of the biggest reasons I think every programmer should take an OS course.

If there were mutliple producers or consumers, we would have the same issue with the modification of `in` and `out`, so we can't rely on the "empty slot" approach in the more general case.

We need to make those statements that increment and decrement `counter` *atomic*.

We say that the modification of `counter` is a *critical section*.

Critical Sections

The Critical-Section problem:

- n processes, all competing to use some shared data
- each process has a code segment (the critical section) in which shared data is accessed

```
while (1) {
```

```
    <CS Entry>
    critical section
    <CS Exit>
    non-critical section
}
```

- Need to ensure that when one process is executing in its critical section, no other process is allowed to do so

Example: Intersection/traffic light analogy

Example: one-lane bridges during construction

Any solution to the critical section problem must satisfy three conditions:

1. *Mutual exclusion*: If process P_i is executing in its critical section, then no other processes can be executing in their critical sections. "One at a time."
2. *Progress*: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely. "no unnecessary waiting."
3. *Bounded waiting*: A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted. "no starvation." (We must assume that each process executes at non-zero speed, but make no assumptions about relative speeds of processes)

In an operating systems course, we would study ways to use hardware and operating system support to provide the capabilities for programming languages to offer synchronization techniques (such as mutual exclusion locks).