Announcements

HW2

- Due this coming-up Monday

Thursday, 21 July

- Optimization Lab

Wednesday, 27 July

- Final Exam
- In-class
- Vote: (a) closed book or (b) 1 page (front and back) of notes?
- No electronic help of any kind is allowed

Thursday, 28 July

- Last lab (Message passing)
This week’s lab, you’ll be asked to implement 4 different runs of the ridiculousCalculation, such that:

- Not optimized, Not threaded, slow
- Not optimized, threaded, slow
- Optimized, Not threaded, fast
- Optimized, threaded, fast

---

**CSCI 322**
**Lab 6: Threading and Code Optimization**
**19 February 2015**

In this lab you’ll gain more experience with the threading functionality in C++, and also put to use a few of the code optimization techniques that have been discussed in lecture. The motivation (goal) for this lab is to determine the extent that optimizing code by hand improves overall program performance.

**I. Create a lab directory**

In your account on your N drive, create a lab 6 directory.

**II. Sample C++ non optimized, non-threaded program**

Download from the course website the file optimizingThreads.cpp. It is code for a C++ program that invokes a function, ridiculousCalc, which includes a for loop with non-trivial calculations. Spend a moment looking at that code, and notice the many math operations in the body of the for loop.

Compile the code and invoke the executable:

```
c++ optimizingThreads.cpp -o optimizingThreads -std=++0x
```

```
./optimizingThreads
```

The run-time and output should be similar to what you see on the right:

**III. Code optimization**

As you’ve seen in lecture, optimizing code by-hand can be done via the application of loop fission, loop fusion, loop peeling, etc. Most of these rely on first identifying lines of code that are dependent.

For this portion of the lab, create a new function, ridiculousCalcOptimized, that performs the same calculation as ridiculousCalc but via an optimized for loop. Do NOT use threading for this portion of the lab. Your goal is to reduce the run-time of the program.

Although the exact behavior of your program will depend on the optimization that you perform and the specs of your computer, strive to achieve a run-time improvement that is at least several hundredths of a second. A sample output, with a speedup of about 7 hundredths of a second, is shown right.
From last time

- Each critical section can be accessed by only one processes at a time
- The entry and exit protocol is what we need to implement such that:
  - **Mutual Exclusion**: At most one process at a time is executing its critical section
  - **Absence of Deadlock**: If two or more processes are trying to enter their critical sections, at least one will succeed
  - **Absence of Unnecessary delay**: A process trying to enter its critical section is allowed to do so when other processes are NOT in their critical sections and/or have terminated
  - **Eventual Entry**: A process trying to enter its critical section eventually will

**Q: What is a critical section?**
From last time

- Each critical section can be accessed by only one processes at a time
- The entry and exit protocol is what we need to implement such that:
  - **Mutual Exclusion**: At most one process at a time is executing its critical section
  - **Absence of Deadlock**: If two or more processes are trying to enter their critical sections, at least one will succeed
  - **Absence of Unnecessary delay**: A process trying to enter its critical section is allowed to do so when other processes are NOT in their critical sections and/or have terminated
  - **Eventual Entry**: A process trying to enter its critical section eventually will

Q: What is the major shortcoming of this approach?

```java
process CS1{
    while(true){
        while (in2){}
        in1=true;
        critical section;
        in1=false;
        noncritical section;
    }
}
```

Global Variables

```java
in1 = false
in2 = false
```
From last time

- Each critical section can be accessed by only one process at a time
- The entry and exit protocol is what we need to implement such that:
  - **Mutual Exclusion**: At most one process at a time is executing its critical section
  - **Absence of Deadlock**: If two or more processes are trying to enter their critical sections, at least one will succeed
  - **Absence of Unnecessary delay**: A process trying to enter its critical section is allowed to do so when other processes are NOT in their critical sections and/or have terminated
  - **Eventual Entry**: A process trying to enter its critical section eventually will

```java
process CS1{
    while (true) {
        while (in2) {}  // Global Variable
        in1 = true;
        critical section;
        in1 = false;
        noncritical section;
    }
}
```

Q: What is the major shortcoming of this approach?

Q: What is a drawback of this approach?

```
process CS1{
    while (true) {
        while (lock) {}  // Global Variable
        lock = true;
        critical section;
        lock = false;
        noncritical section;
    }
}
```

Global Variables
- `in1 = false`
- `in2 = false`

Global Variable
- `bool lock = false`
From last time

A really good question ... “Why not use a semaphore to achieve the same behavior as when two threads use in1, in2, and/or lock”
From last time

A really good question ... “Why not use a semaphore to achieve the same behavior as when two threads use in1, in2, and/or lock”

For the sake of demonstration, let us use a semaphore

Q: Does the use of this semaphore enforce mutual exclusion of the critical section code for CS1 and CS2?
From last time

A really good question ... “Why not use a semaphore to achieve the same behavior as when two threads use `in1`, `in2`, and/or `lock`”

For the sake of demonstration, let us use a semaphore

```
value = 0
```

Q: Does the use of this semaphore enforce mutual exclusion of the critical sections for `CS1` through `CS873`?
From last time

A really good question ... “Why not use a semaphore to achieve the same behavior as when two threads use in1, in2, and/or lock”

For the sake of demonstration, let us use a semaphore

Q: Does the use of this semaphore enforce mutual exclusion of the critical sections for CS1 through CS873?

Q: Is this efficient?

Q: Are mutual exclusion, absence of deadlock, avoiding unnecessary delay, and eventual entry enforced?
Today

Test and Set
Test and Test and Set
Tie Breaker Algorithm
Barriers
Test and Set

In practice, almost all computers – especially multiprocessors – have special instructions to implement critical section locks.

```cpp
bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
>
}
```

Q: What is the output of TS upon input of true? Upon input of false?
In practice, almost all computers – especially multiprocessors – have special instructions to implement critical section locks.

```c
bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
}> 
```

Q: What is the output of TS upon input of true? Upon input of false?

- `TS(true)`: bool initial = true
  - lock = true
  - return true
- `TS(false)`: bool initial = false
  - lock = true
  - return false

Q: What is the advantage of using TS?

It appears to ALWAYS set lock=true, and then return the “old” value of lock. How does this help our cause?
Test and Set

```c
bool lock = false;

bool TS(bool lock){
    < bool initial = lock;
    lock = true;
    return initial;
}

process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

At the onset, the value of `lock`, a shared variable, is set to false
Test and Set

Assume CS1 goes “first” ... checks TS (lock), and sets the value of initial to false
Test and Set

- Assume CS1 goes “first” ... checks TS (lock), and sets the value of initial to false, then changes lock to true
Test and Set

bool lock = true;

bool TS(bool lock){
    < bool initial = lock;
    lock = true;
    return initial;
}

process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

- Assume CS1 goes “first” ... checks TS (lock), and sets the value of initial to false, then changes lock to true, and returns false

Q: At this point, what if CS2 were scheduled to go immediately next (while CS1 is JUST about to enter its critical region)?
Test and Set

bool lock = true;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
    >
}

process CS1{
    while (true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while (true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

• Assume CS1 goes “first” ... checks TS (lock), and sets the value of initial to false, then changes lock to true, and returns false
• CS2 is spin locked, and CS1 enters its critical section
### Test and Set

```c
bool lock = false;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
>
}
```

```c
process CS1{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

```c
process CS2{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

- Assume CS1 goes “first” ... checks TS (lock), and sets the value of initial to false, then changes lock to true, and returns false
- CS2 is spin locked, and CS1 enters its critical section, and THEN sets the value of lock to false

Q: At this point, what if CS2 were scheduled to go immediately next?
Test and Set

```c
bool lock = false;

bool TS(bool lock){
    < bool initial = lock;
    lock = true;
    return initial;
}>

process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

• Assume CS1 goes “first” ... checks TS (lock), and sets the value of initial to false, then changes lock to true, and returns false
• CS2 is spin locked, and CS1 enters its critical section, and THEN sets the value of lock to false
• CS1 can now execute its non critical section, and the lock is now set to false, indicating that another process can enter its critical section
Test and Set

bool lock = false;
bool TS(bool lock){
  < bool initial = lock;
  lock = true;
  return initial;
}

process CS1{
  while (true) {
    while (TS(lock)) {};
    critical section;
    lock = false;
    noncritical section;
  }
}

process CS2{
  while (true) {
    while (TS(lock)) {};
    critical section;
    lock = false;
    noncritical section;
  }
}

• Assume CS1 goes “first” ... checks TS (lock) , and sets the value of initial to false, then changes lock to true, and returns false
• CS2 is spin locked, and CS1 enters its critical section, and THEN sets the value of lock to false
• CS1 can now execute its non critical section, and the lock is now set to false, indicating that another process can enter its critical section

Q: What is the importance of TS being atomic?
Test and Set

```cpp
bool lock = false;

bool TS(bool lock){
    bool initial = lock;
    lock = true;
    return initial;
}
```

```cpp
process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

```cpp
process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

- Assume CS1 goes “first” ... checks TS (lock), and sets the value of initial to false, then changes lock to true, and returns false
- CS2 is spin locked, and CS1 enters its critical section, and THEN sets the value of lock to false
- CS1 can now execute its non critical section, and the lock is now set to false, indicating that another process can enter its critical section

Q: Does each process maintain its important properties (mutual exclusion, absence of deadlock, avoiding unnecessary delay, eventual entry)?
Test and Set

```c
bool lock = false;

bool TS(bool lock){
    < bool initial = lock;
    lock = true;
    return initial;
}
```

```c
process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

```c
process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

Observation: Exit protocols set the shared variable to its initial value (false)
Test and Set

```c
bool lock = false;

bool TS(bool lock){
  < bool initial = lock;
  lock = true;
  return initial;
}

process CS1{
  while(true){
    while (TS(lock)) {};
    critical section;
    lock = false;
    noncritical section;
  }
}

process CS2{
  while(true){
    while (TS(lock)) {};
    critical section;
    lock = false;
    noncritical section;
  }
}
```

Observation: Exit protocols set the shared variable to its initial value (false)

Q: What is a drawback of using TS?
Test and Set

```c
bool lock = false;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
}
```

```c
process CS1{
    while (true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

```c
process CS2{
    while (true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

Observation: Exit protocols set the shared variable to its initial value (false)

Q: What is a drawback of using TS?

(hint: does TS do “useful” work each time that it is executed?)
Test and Set

```c
bool lock = false;

bool TS(bool lock){
    < bool initial = lock;
    lock = true;
    return initial;
    >
}

process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

Observation: Exit protocols set the shared variable to its initial value (false)

Q: What is a drawback of using TS?

(hint: does TS do “useful” work each time that it is executed?)

If there are a gazillion processes relying on the shared variable lock, memory contention results. And, TS sets the value of lock = true EACH time that TS is invoked, even when the value of lock does not change.
Test and Set

```c
bool lock = false;
bool TS(bool lock){
    < bool initial = lock;
    lock = true;
    return initial;
}

process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

**In-class exercise 1**

Modify the code for each process so that TS is not unnecessarily invoked
Test and Test and Set

```c
bool lock = false;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
}

process CS1{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS1{
    while(true){
        while (lock) {};
        while (TS(lock)) {};
        while (lock) {};
    }
    critical section;
    lock = false;
    noncritical section
}

Q: What is the advantage of adding additional while loops (that check the value of the shared variable lock?)
**Test and Test and Set**

```c
bool lock = false;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
    >
}

process CS1{
    while(true) {
        while (lock) {};
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}
```

Q: What is the advantage of adding additional while loops (that check the value of the shared variable lock?)

A process only tests `lock`, which can be read from local cache, UNTIL there is a possibility that `TS` can succeed.

This is called **Test and Test and Set**.
Q: Can you think of a scenario where such an approach might be unfair (for either CS1 or CS2?)
bool lock = false;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
}

process CS1{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

process CS2{
    while(true) {
        while (TS(lock)) {};
        critical section;
        lock = false;
        noncritical section;
    }
}

Assume the following scenario

- CS1 runs, sets lock, and enters its critical section, then sets lock to false
Assume the following scenario

- CS1 runs, sets lock, and enters its critical section, then sets lock to false
- At this point, CS1 is in its non-critical section, and CS2 can enter its critical section ...
bool lock = false;

bool TS(bool lock) {
    < bool initial = lock;
    lock = true;
    return initial;
    >
}

process CS1{
    while(true) {
        while (TS(lock)) {
        }
        critical section;
        lock = false;
        noncritical section;
        }
}

process CS2{
    while(true) {
        while (TS(lock)) {
        }
        critical section;
        lock = false;
        noncritical section;
        }
}

Assume the following scenario

- CS1 runs, sets lock, and enters its critical section, then sets lock to false
- At this point, CS1 is in its non critical section, and CS2 can enter its critical section ... but CS1 races back and gains access first

We need an approach for “playing fair”

Q: How might you modify this approach so that processes take turns at entering their critical sections
### Tie-Breaker

**Global Variable(s):**

```plaintext
process CS1{
    while(true) {
        critical section;
        noncritical section;
    }
}
```

```plaintext
process CS2{
    while(true) {
        critical section;
        noncritical section;
    }
}
```

---

### In class exercise 2

Add to the code for CS1 and CS2 so that neither CS1 nor CS2 can prevent the other process from having a turn at entering its critical section. Declare and use as many shared variables as you need.
Global Variable(s): bool in1=false, in2=false; int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

Q: At the onset of this program (before CS1 and CS2 have begun their outer-most while loops), which process will be allowed to enter its critical section first?

If CS1 is scheduled first?
If CS2 is scheduled first?
Tie-Breaker

Global Variable(s): bool in1=true, in2=false; int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

If the execution of CS1 is here, can CS1 proceed?
If the execution of CS1 is here, can CS1 proceed?

It depends what CS2 is currently executing, and whether CS2 has set the value of in2 to be true.
Tie-Breaker

Global Variable(s): bool in1=false, in2=false; int last=1

If both CS1 and CS2 execute this code ... what are the resulting values for `in1`, `in2`, and `last`?
Tie-Breaker

**Global Variable(s):** bool in1=true, in2=true; int last=1

```c
process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {];// critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {];// critical section;
        in2 = false;
        noncritical section;
    }
}
```

If both in1 and in2 are true, the first portion of the conditional for the while evaluates to true, but last can be ONLY 1 or 2, so only one of these will proceed (based on the value of last)

Assuming that last=1, what are the next steps?
Tie-Breaker

Global Variable(s): bool in1=true, in2=true; int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

CS1 loops, while CS2 proceeds into its critical section
Global Variable(s): bool in1=true, in2=false; int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

CS1 loops, while CS2 proceeds into its critical section

And eventually sets in2 to false

Q: What effect does that have on CS1?
Tie-Breaker

Global Variable(s): bool in1=true, in2=false; int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

CS1 loops, while CS2 proceeds into its critical section

And eventually sets in2 to false

Q: What effect does that have on CS1?

CS1 is permitted to proceed to its critical section
Tie-Breaker

Global Variable(s): bool in1=true, in2=false; int last=1

process CS1{
    while(true){
        in1 = true; last = 1;
        while (in2 and last == 1) {};
        critical section;
        in1 = false;
        noncritical section;
    }
}

process CS2{
    while(true){
        in2 = true; last = 2;
        while (in1 and last == 2) {};
        critical section;
        in2 = false;
        noncritical section;
    }
}

Q: Is this approach ideal? Why or why not?
Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of `co` to execute each iteration of a task on a separate thread.

```java
while (true) {
    co [i = 1 to n] // perform calculation i
    oc
}
```

The issue here is that different threads might end at different times, so we need a way to coordinate end times, and enforce that threads “wait” for the other threads.
Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of a co to execute each iteration of a task on a separate thread.

```plaintext
while (true) {
    co [i = 1 to n]
    // perform calc i
    oc
}
```
Barriers

Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of `co` to execute each iteration of a task on a separate thread.

```
while (true){
  co [i = 1 to n]
    // perform calc i
  oc
}
```

Assume calculation \( i \) is one part of an iterative algorithm, and calculation \( i \) during one iteration of the while loop depends on some calculation \( i \) of a previous iteration of the while loop .... Which may have been performed by a different thread

Q: What are shortcomings of this approach, using the above code?
Barriers

Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of `co` to execute each iteration of a task on a separate thread.

```java
while (true){
    co [i = 1 to n]
    // perform calc i
}
```

Threads at one “level” of execution may end at slightly different times, so we need a way to coordinate end times, and enforce that threads “wait” for the other threads.

Completes at time $t$

Completes at time $t+n$
Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of co to execute each iteration of a task on a separate thread.

Prevent the “next” calculation from proceeding until both “lower” calculations are done.
Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of `co` to execute each iteration of a task on a separate thread.

During EACH iteration of the while loop, \( n \) new threads are created ... which causes excessive overhead.

```java
while (true){
    co [i = 1 to n]
    // perform calc i
    oc
}
```
Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of co to execute each iteration of a task on a separate thread.

```
while (true){
    co [i = 1 to n]
    // perform calc i
    oc
}
```

We want to impose a “hold off and wait” mechanism that prevents a calculation $x$ from being performed before the calculations $y$ and $z$ are done, which provide the operands for $x$.
Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of `co` to execute each iteration of a task on a separate thread.

```java
while (true) {
    co [i = 1 to n]
    // perform calc i
    oc
}
```

```java
process Worker[i=1 to n]
    while (true) {
        // perform calc i
        // wait for all n
    }
}
```

Q: What are the advantages of the code on the right over the code that is shown on the left?
Iterative algorithms often depend on the result of a “previous” iteration. Implementing such an algorithm might rely on the use of `co` to execute each iteration of a task on a separate thread.

```java
while (true) {
    co [i = 1 to n]
    // perform calc i
    oc
}
```

```java
process Worker[i=1 to n]
    while (true) {
        // perform calc i
        // wait for all n
    }
}
```

**Barrier Synchronization**

- If each Worker is a thread, then at most \( n \) threads are EVER created.
- Each thread does not proceed to its next iteration (of its while loop) until all other threads (Workers) have completed their most-recent iteration of their while loop.

**Q: How do we implement this? (on the board discussion)**
Barriers

```c
int count=0;

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < count++; >
        <await (count == n) >
    }
```

You should by now recognize that this can be implemented using a while (infinite) loop.
Q: What are the shortcomings of this approach?

```c
int count=0;

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < count++; >
        <await (count == n) >
    }
}
```
Q: In the worst case, how many other processes are accessing count?

Q: What are the shortcomings of this approach?

- Count must be reset to 0 each time all processes have passed their barrier.
- If n is large, there are n processes constantly checking the value of count.

```c
int count=0;

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < count++; >
        <await (count == n) >
    }
```
Barriers

```c
int count=0;

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < count++; >
        <await (count == n) >
    }
}
```

Q: What are the shortcomings of this approach?

- Count must be reset to 0 each time all processes have passed their barrier
- If \( n \) is large, there are \( n \) processes constantly checking the value of `count`

Q: In the worst case, how many other processes are accessing `count`? \( n-1 \)

Q: Therefore under what conditions are barriers in combination with global counters suitable for use?
Barriers

```c
int count = 0;

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < count++; >
        <await (count == n) >
    }

Q: What are the shortcomings of this approach?

• Count must be reset to 0 each time all processes have passed their barrier
• If $n$ is large, there are $n$ processes constantly checking the value of count

Q: In the worst case, how many other processes are accessing count? $n - 1$

Q: Therefore under what conditions are barriers in combination with global counters suitable for use?

Barriers implemented with counters only are suitable for when $n$ is relatively small
Barriers

```c
int count=0;

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < count++; >
        <await (count == n) >
    }
```

Q: If $n$ is not small, how can we prevent the memory contention caused by many Workers accessing a shared variable (counter?)
Bars

```python
int count=0;

process Worker[i=1 to n]
  while (true){
    // perform calc i
    < count++; >
    <await (count == n) >
  }
}

int arrive[1:n];

process Worker[i=1 to n]
  while (true){
    // perform calc i
    < arrive[i] = 1; >
    <await (count == n)> 
  }
}
```

Let `arrive[1:n]` be an array of integers initialized to 0, and replace `count++` by `arrive[i] = 1`
Let `arrive[1:n]` be an array of integers initialized to 0, and replace `count++` by `arrive[i] = 1`.

Q: If we do this, how should we change the await statement?
Barriers

Let \( \text{arrive}[1:n] \) be an array of integers initialized to 0, and replace \( \text{count}++ \) by \( \text{arrive}[i] = 1 \)

**Q:** If we do this, how should we change the \textit{await} statement?

\[
\text{<await ((arrive[1] + \ldots arrive[n]) == n);>}
\]
Let \( \text{arrive}[1:n] \) be an array of integers initialized to 0, and replace \( \text{count}++ \) by \( \text{arrive}[i] = 1 \)

Q: If we do this, how should we change the \text{await} statement?

\[ \langle \text{await } ((\text{arrive}[1] + \ldots \text{arrive}[n]) == n) \rangle \]

Q: Is this an optimal solution? Why or why not?
Let `arrive[1:n]` be an array of integers initialized to 0, and replace `count++` by `arrive[i] = 1`

**Q:** If we do this, how should we change the `await` statement?

```plaintext
<await ((arrive[1] + ... arrive[n]) == n)>;
```

This is an inefficient approach because each Worker is computing the sum
Barriers

In class exercise 3

- Create another process to oversee all workers and which provides a mechanism for efficient barrier implementation
- Modify the worker code (add and/or remove statements) as needed
- The goal is to **minimize** the number of concurrent accesses to ALL shared resources

```c
int arrive[1:n];

process Worker[i=1 to n]
  while (true){
    // perform calc i
    < arrive[i] = 1; >
    < await (count == n)>
  }
}
Barriers

```c
int cont[1:n];

process Coordinator {
    // check arrive[i:1->n]
    // if all arrive[i] = 1
    // set cont[i] = 1
}
```

Create another process, call it `Coordinator`, and have it calculate the sum, and have each worker check the value updated by the coordinator in an array `cont[1:n]`. 
Barriers

int cont[1:n];

process Coordinator {
    // check arrive[i:1->n]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n] {
    while (true) {
        // perform calc i
        <arrive[i] = 1; >
        <await (cont[i]==1)>}
}

Create another process, call it Coordinator, and have it calculate the sum, and have each worker check the value updated by the coordinator in an array cont[1:n]

Each worker sets only a single entry of arrive
int cont[1:n];

process Coordinator {
    // check arrive[i:1->n]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n] {
    while (true){
        // perform calc i
        < arrive[i] = 1; >
        < await (cont[i]==1)>
    }
}

Create another process, call it Coordinator, and have it calculate the sum, and have each worker check the value updated by the coordinator in an array cont[1:n]

Each worker sets only a single entry of arrive

Each Worker checks only a SINGLE entry of the array cont
### Barriers

```plaintext
int cont[1:n];

process Coordinator {
  // check arrive[i:1->n]
  // if all arrive[i] = 1
  //    set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n]
  while (true){
    // perform calc i
    < arrive[i] = 1; >
    <await (cont[i]==1)>  
  }
```

**Q:** How many memory locations does each Worker read?

**Q:** How many memory locations does each Worker write to?

**Q:** What is the maximum number of threads (Coordinator or Worker) that concurrently access any single entry of a shared variable or index of a shared array?
Barriers

process Coordinator {
    // check arrive[1:1->n]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

process Worker_1
    while (true){
        // perform calc i
        < arrive[1] = 1; >
        <await (cont[1]==1)> 
    }
}

process Worker_9
    while (true){
        // perform calc i
        < arrive[9] = 1; >
        <await (cont[9]==1)> 
    }
}

process Worker_14
    while (true){
        // perform calc i
        < arrive[14] = 1; >
        <await (cont[14]==1)> 
    }
}

cont

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

arrive

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Barriers

Q: How many processes read (concurrently) each index of cont?
Barriers

Q: How many processes read (concurrently) each index of cont?

Only one Worker
Barriers

**process Coordinator**

```c
// check arrive[1:1->n]
// if all arrive[1] = 1
// set cont[1] = 1
```

**process Worker_1**

```c
while (true){
    // perform calc i
    < arrive[1] = 1; >
    <await (cont[1] == 1)>
}
```

**process Worker_9**

```c
while (true){
    // perform calc i
    < arrive[9] = 1; >
    <await (cont[9] == 1)>
}
```

**process Worker_14**

```c
while (true){
    // perform calc i
    < arrive[14] = 1; >
    <await (cont[14] == 1)>
}
```

---

**Q:** How many processes read (concurrently) each index of `cont`?

**Q:** How many processes write (concurrently) each index of `cont`?

---

**arrive**

<table>
<thead>
<tr>
<th>1 0 1 1</th>
<th>1 0 1 1</th>
<th>1 0 1 1</th>
<th>1 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4</td>
<td>5 6 7 8</td>
<td>9 1 1 1</td>
<td>1 0 1 2 3 4</td>
</tr>
</tbody>
</table>

**cont**

<table>
<thead>
<tr>
<th>0 1 1 1 0 0 0 0</th>
<th>1 0 1 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8</td>
<td>9 1 1 1 1 1</td>
</tr>
<tr>
<td>0 1 2 3 4</td>
<td></td>
</tr>
</tbody>
</table>
Q: How many processes read (concurrently) each index of cont?

Only one Worker

Q: How many processes write (concurrently) each index of cont?

Only the coordinator
Barriers

Only 1 process is reading from, and only 1 process is writing to any one index of \texttt{cont}
Barriers

```
process Coordinator {
    // check arrive[1:1->n]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

process Worker_1
while (true){
    // perform calc
    < arrive[1] = 1; >
    <await (cont[1]==1)>}

process Worker_9
while (true){
    // perform calc
    < arrive[9] = 1; >
    <await (cont[9]==1)>}

process Worker_14
while (true){
    // perform calc
    < arrive[14] = 1; >
    <await (cont[14]==1)>}
```

Only 1 process is reading from, and only 1 process is writing to any one index of `cont`.

Q: How many processes read (concurrently) each index of `arrive`?
Barriers

### Process Coordinator
```c
process Coordinator {
    // check arrive[i] = 1
    // if all arrive[i] = 1
    // set cont[i] = 1
}
```

### Process Worker_1
```c
process Worker_1 {
    while (true){
        // perform calc
        <arrive[1] = 1;>
        <await (cont[1]==1)>
    }
}
```

### Process Worker_9
```c
process Worker_9 {
    while (true){
        // perform calc
        <arrive[9] = 1;>
        <await (cont[9]==1)>
    }
}
```

### Process Worker_14
```c
process Worker_14 {
    while (true){
        // perform calc
        <arrive[14] = 1;>
        <await (cont[14]==1)>
    }
}
```

---

**Cont**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Arrive**

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Q:** How many processes read (concurrently) each index of **arrive**?

Only the Coordinator

---

Only 1 process is reading from, and only 1 process is writing to any one index of **cont**.
Barriers

Only 1 process is reading from, and only 1 process is writing to any one index of `cont`.

Q: How many processes write (concurrently) each index of `arrive`?
Bars

Only 1 process is reading from, and only 1 process is writing to any one index of `cont`.

Q: How many processes write (concurrently) each index of `arrive`?

Only one of the Workers
Barriers

process Coordinator {
    // check arrive[1:n]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

process Worker_1
    while (true)
        // perform calc
        <arrive[1] = 1>
        <await (cont[1] == 1)>
    }

process Worker_9
    while (true)
        // perform calc
        <arrive[9] = 1>
        <await (cont[9] == 1)>
    }

process Worker_14
    while (true)
        // perform calc
        <arrive[14] = 1>
        <await (cont[14] == 1)>
    }

Only 1 process is reading from, and only 1 process is writing to any one index of cont

Only 1 process is reading from, and only 1 process is writing to any one index of arrive
Barriers

```c
int cont[1:n];

process Coordinator {
    // checks arrive[i]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < arrive[i] = 1; >
        <await (cont[i] == 1) >
    }
}
```

Q: Is the Coordinator/Worker(s) approach optimal?

Task: Discuss the shortcomings of the coordinator/workers approach
### Barriers

```plaintext
int cont[1:n];

process Coordinator {
  // checks arrive[i]
  // if all arrive[i] = 1
  // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n]
  while (true){
    // perform calc i
    <arrive[i] = 1;>
    <await (cont[i] == 1)>
  }
}
```

- A processor has been utilized on which the Coordinator process is executed.
- The Coordinator process has to check ALL entries of `arrive`, and although that can be done quickly, it is done continually. **Q: How is this implemented?**
Barriers

```c
int cont[1:n];

process Coordinator {
    // checks arrive[i]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n]
    while (true){
        // perform calc i
        <arrive[i] = 1;>
        <await (cont[i] == 1)>
    }
}
```

- A processor has been utilized on which the Coordinator process is executed.
- The Coordinator process has to check ALL entries of `arrive`, and although that can be done quickly, it is done continually. Q: How is this implemented?

```c
for [i=1:n] <await (arrive[i] == 1)>;
for [i=1:n] cont[i] = 1;
```

Q: Is this inefficient, and if so why?
Barriers

In iterative algorithms, each worker is most often performing the same task, except on a different data set. Thus there is a high likelihood that each Worker will finish with its calculation at the same time as the other Workers, but Coordinator inspects each `arrive` entry in turn.

```c
int cont[1:n];

process Coordinator {
    // checks arrive[i]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

for [i=1:n] <await (arrive[i] == 1);>
for [i=1:n] cont[i] = 1;

Q: Is there a better solution?
```
Solution: Combine the actions of the coordinator and worker so that each worker is also a coordinator.
When a “leaf” worker finishes its calculation, it “tells” its parent that it is done.

```c
int cont[1:n];

process Coordinator {
    // checks arrive[i]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n] {
    while (true) {
        // perform calc i
        < arrive[i] = 1; >
        <await (cont[i] == 1) >
    }
}
```
Barriers

An interior node worker informs its parent that it is “done” when it finishes its own calculation, and both of its children's’ calculations have concluded.
Barriers

The root node waits until its calculation is done, and it hears back from its immediately 2 “children”
int cont[1:n];

process Coordinator {
    // checks arrive[i]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < arrive[i] = 1; >
        <await (cont[i] == 1) >
    }
}

The root node waits until its calculation is done, and it hears back from its immediately 2 “children”

When that happens, the “root” worker knows that ALL workers have reached their barriers

Speedup: for n workers, \(\log_2{n}\) due to tree structure
Barriers

```
int cont[1:n];

process Coordinator {
    // checks arrive[i]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n] {
    while (true){
        // perform calc
        < arrive[i] = 1; >
        <await (cont[i] == 1) >
    }
}
```

Q: How should the root worker inform all its children that they should proceed to the next iteration of their while loop?
Barriers

```c
int cont[1:n];

process Coordinator {
    // checks arrive[i]
    // if all arrive[i] = 1
    // set cont[i] = 1
}

int arrive[1:n];

process Worker[i=1 to n]
    while (true){
        // perform calc i
        < arrive[i] = 1; >
        <await (cont[i] == 1) >
    }
}
```

Q: How should the root worker inform all its children that they should proceed to the next iteration of their while loop?

Send a broadcast message

Q: What is the shortcoming of this approach?
Barriers

• Different workers (root, interior, or leaf) perform different amounts of work in terms of signaling others.
• But if each worker is doing “about” the same amount of computation as any other worker, all will “finish” performing their \( \text{calc i} \) at about the same time.

If every worker finishes its “calculation of \( i \)“ at the same time, we don’t want to wait for messages to be propagated up and down the entire tree structure.

Q: Is there a yet more efficient approach to minimize the time that a worker has to wait (due to propagation of “go” messages) before proceeding past its barrier?
Scenario: All workers finish at the same time
Goal: Want all workers to proceed past a barrier as soon as possible

Assume you have a two-process barrier: a barrier that is used to check if two processes have finished.

Q: Can we use such a barrier? And if so, how?
Bars

Assume you have a two-process barrier: a barrier that is used to check if two processes have finished.

Q: Can we use such a barrier? And if so, how?

In class exercise: For 8 Workers, what is the minimum number of 2-process barriers that you can use to ensure that ALL workers are aware of when all other workers have completed their calculation.
Use 4 of these 2-process barriers to coordinate pairs of workers

Q: Is that enough? Do we need to impose more barriers? Why or why not?
Use 4 of these 2-process barriers to coordinate pairs of workers

Q: Is that enough? Do we need to impose more barriers? Why or why not?

We want to prevent a set of processes from racing back and performing their “next” calculation before all others have finished theirs

What else can we do?
Barriers

Is that it? Or do we need more checks?
Barriers

Butterfly Barrier: at Stage $s$, each worker synchronizes with a worker a distance $2^{s-1}$ away.

Q: Can a butterfly barrier be used in all scenarios?
### Barriers

| worker | worker | worker | worker | worker | worker | worker | worker | worker | worker |

S1

S2

S3

**Butterfly Barrier**: at Stage $s$, each worker synchronizes with a worker a distance $2^{s-1}$ away

**Q**: Can a butterfly barrier be used in all scenarios?

**A**: No (not without some creative tweaking)
Barriers

S1

S2

Dissemination barrier

(Last stage not shown)
Returning the Exam

<table>
<thead>
<tr>
<th>Q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>EC1</th>
<th>EC2</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Correct</td>
<td>100</td>
<td>82</td>
<td>52</td>
<td>95</td>
<td>95</td>
<td>86</td>
<td>91</td>
<td>86</td>
<td>30</td>
<td>30</td>
<td>95</td>
<td>78</td>
<td>86</td>
<td>34</td>
<td>65</td>
<td>62</td>
<td>34</td>
<td>17</td>
<td>13</td>
</tr>
<tr>
<td>D/A/E</td>
<td>E</td>
<td>E</td>
<td>A</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>D</td>
<td>D</td>
<td>E</td>
<td>E</td>
<td>E</td>
<td>D</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>
Returning the Exam

Original code

```c
for (int x=0; x<n; x++){
    p[x] = 0.0
    for (int y=0; y<n; y++){
        s[y] = 0.0;
        for (int z=0; z<n; z++){
            p[x] = p[x] + q[y]*r[z];
        }
    }
}
```
Returning the Exam

Original code

```java
for (int x=0; x<n; x++){
    p[x] = 0.0
    for (int y=0; y<n; y++){
        s[y] = 0.0;
        for (int z=0; z<n; z++){
            p[x] = p[x] + q[y]*r[z];
        }
    }
}
```

If the outer-most loop is parallelized

<table>
<thead>
<tr>
<th>CPU0</th>
<th>CPU1</th>
<th>CPU2</th>
<th>CPU3</th>
</tr>
</thead>
<tbody>
<tr>
<td>i=0</td>
<td>i=1</td>
<td>i=2</td>
<td>i=3</td>
</tr>
</tbody>
</table>

Q: What code does each CPU run?
If the outer-most loop is parallelized

Can any of the CPU's overwrite the “results” of any other CPU?
Returning the Exam

Original code

```java
for (int x=0; x<n; x++){
    p[x] = 0.0
    for (int y=0; y<n; y++){
        s[y] = 0.0;
        for (int z=0; z<n; z++){
            p[x] = p[x] + q[y]*r[z];
        }
    }
}
```

If the outer-most loop is parallelized

If the two outer-most loops are parallelized

Can any of the CPU’s overwrite the “results” of any other CPU?

Can any of the CPU’s overwrite the “results” of any other CPU?
Up Next

Dissemination Barriers
Data Parallel Algorithms
Monitors