From last time

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

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

Our goal: allow CS1 and CS2 to have turns gaining access to their critical sections.
From last time

```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;
    }
}
```

**TS (Test and Set)** is an atomic action procedure that relies on a shared variable `lock`.

**Our goal:** allow CS1 and CS2 to have turns gaining access to their critical sections.

**Task:** Be able to explain the two shortcomings of using TS (consider CS1 through CSn, where n is large), which motivates the following solution ...
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.**

Regardless of whether you use TS and Test and Test and Set, there is the chance that any process will repeatedly acquire the lock and starve all other processes.
From last time

bool in1=false
bool 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;
    }
}

Task: Be able to explain why this use of global (shared) variables implements a fair access policy for CS1 and CS2 gaining access to their critical sections

Q: What does the variable last specify?
Scenario 1: CS1 is executing the body of its while loop, and CS2 is evaluating the condition of its outer most while loop
From last time

```c
bool in1=true
bool in2=false
int last=1
```

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

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

The while condition evaluates to False, and the spin loop does not execute.
From last time

bool in1=true
bool in2=true
int last=2

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 enters its critical section

If at any time when CS1 is past its inner most while loop CS2 sets in2 to true, and last to 2 ...
From last time

bool in1=true
bool in2=true
int last=2

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 enters its critical section

If at any time when CS1 is past its inner most while loop CS2 sets in2 to true, and last to 2 ...
The while loop condition in CS2 evaluates to true and CS2 spins
From last time

bool in1=false
bool in2=true
int last=2

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 eventually completes its critical section and sets the value of in1 to false

Q: As soon as CS1 sets the value of in1 to false, what happens?

If at any time when CS1 is past its inner most while loop CS2 sets in2 to true, and last to 2 ...
The while loop condition in CS2 evaluates to true and CS2 spins
From last time

bool in1=false
bool in2=true
int last=2

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 eventually completes its critical section and sets the value of \texttt{in1} to false

Q: As soon as CS1 sets the value of \texttt{in1} to false, what happens?

The condition of the while loop in CS2 evaluates to false ...
From last time

```c
bool in1=false
bool in2=true
int last=2

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 eventually completes its critical section and sets the value of `in1` to false

Q: As soon as CS1 sets the value of `in1` to false, what happens?

The condition of the while loop in CS2 evaluates to false ...
And CS2 enters its critical section.
From last time

```c
bool in1=true
bool 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;
    }
}
```

While CS2 is in its critical section, CS1 might complete its non critical section, and proceed to set `in1` to `true` and `last` to `1` ...

The condition of the while loop in CS2 evaluates to false ... And CS2 enters its critical section
From last time

```c
bool in1=true
bool 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;
    }
}
```

While CS2 is in its critical section, CS1 might complete its non critical section, and proceed to set `in1` to `true` and `last to 1` ...

The condition of the while loop in CS2 evaluates to false ...
And CS2 enters its critical section

Which would cause CS1’s while loop condition to evaluate to true, causing spin
From last time

```c
bool in1=true
bool 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;
    }
}
```

Hence last refers to “who had the last attempt at gaining entry to their critical section”
From last time

```plaintext
bool in1=false
bool 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;
    }
}

Scenario 2 : CS1 is JUST About to execute the body of its while loop, and CS2 is JUST About to execute the body of its while loop
From last time

```c
bool in1 = true
bool 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 and CS2 each update `in1` and `in2`. 
Both CS1 and CS2 attempt to set the value of last

Q: What are the possible outcomes?
true

bool in1=true
bool in2=true
int last=2

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

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

Both CS1 and CS2 attempt to set the value of last

If CS1 sets \textcolor{blue}{last} to 1, and CS2 immediately sets \textcolor{blue}{last} to 2 ...
bool in1=true
bool in2=true
int last=2

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;
    }
}

Both CS1 and CS2 attempt to set the value of last

If CS1 sets last to 1, and CS2 immediately sets last to 2 ... CS2 spins, and CS1 enters its critical section
From last time

bool in1=true
bool 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;
    }
}

Both CS1 and CS2 attempt to set the value of last

If CS1 sets last to 1
Both CS1 and CS2 attempt to set the value of last

If CS1 sets last to 1, proceeds to check its while (before CS2 has set last to 2) ...

CS1’s while condition evaluates to true, so CS1 spins
Today

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

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 2 shortcomings of this approach, using the above code?
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.

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.

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

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.

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

Prevent the “next” calculation from proceeding until both “lower” calculations are done.

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.

During EACH iteration of the while loop, n new threads are created ... which causes excessive overhead.
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
}
```

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.

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

```plaintext
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)
You should by now recognize that this can be implemented using a while (infinite) loop.
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?
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 \( \text{count} \)

Q: In the worst case, how many other processes are accessing \( \text{count} \)?
Barriers

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

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

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

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 \( \text{count} \)

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

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
int count=0;

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

```c
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`
Barriers

```plaintext
int count=0;

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

```plaintext
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`.

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

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

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

<await ((arrive[1] + ... arrive[n]) == n)>
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?

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

Q: Is this an optimal solution? Why or why not?
Barriers

```cpp
int count=0;

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

```cpp
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`

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

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

This is an inefficient approach because each Worker is computing the sum
In class exercise

- 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

int arrive[1:n];

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

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

process Coordinator {
    // check arrives[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]`

Each worker sets only a single entry of `arrive`
Barriers

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

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 | 0 1 1 1 0 0 0 0 1 0 1 0 1 1 |
|      | 1 2 3 4 5 6 7 8 9 1 1 1 1 1 |

| arrive | 1 0 1 1 1 0 1 0 1 1 0 1 1 1 |
|        | 1 2 3 4 5 6 7 8 9 1 1 1 1 1 |
|        | 0 1 2 3 4 5 6 7 8 9 1 1 1 1 |
|        | 0 1 2 3 4 5 6 7 8 9 1 1 1 1 |
Barriers

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

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

Only one of a Worker

CSCI 322
Concurrent Programming
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)}>  
    }

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

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

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

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

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

process Worker_14
    while (true){
        // perform calc
        if (arrive[14] == 1) {
            // perform calc
            <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?

Only one of a Worker

Only the coordinator
Barriers

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

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

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

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

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 read (concurrently) each index of `arrive`?
Barriers

process Coordinator {
    // check arrive[i] == 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?

Only the Coordinator
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`?
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?

Only one of the Workers
### Barriers

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

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

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

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

---

**Table:**

<table>
<thead>
<tr>
<th>cont</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</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></td>
<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>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>arrive</th>
<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>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<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>
<td>1</td>
</tr>
</tbody>
</table>

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

Task: Discuss the shortcomings of the coordinator/workers approach
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 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**

```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) >
    }
}
```

- 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
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.
Barriers

When a “leaf” worker finishes its calculation, it “tells” its parent that it is done.
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

\[
\text{int cont}[1:n];
\]

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

\[
\text{int arrive}[1:n];
\]

process Worker[i=1 to n] {
    while (true) {
        // perform calc
        < 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”
Barriers

```java
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) >
    }
}
```

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

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?
Barriers

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?

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?
Barriers

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?
| worker | worker | worker | worker | worker | worker | worker | worker |

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

Dissemination barrier
(Last stage not shown)
Up Next

Dissemination Barriers
Data Parallel Algorithms
Monitors