Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

Summary:
Dijkstra seeks to solve a common problem involving the critical execution of various concurrent processes involving writing and reading the same data. In solving this problem, nothing can be assumed about the speed, synchronicity or progress through their critical region. The author suggests, to solve this problem, an array of booleans, each indicating the status of those locations which they which to access. By moving past these regions and checking back later, the problem is solved, according to the author.
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

Summary:
Dijkstra seeks to solve a common problem involving the critical execution of various concurrent processes, involving writing and reading the same data. In solving this problem, nothing can be assumed about the speed, synchronicity or progress through their critical region. The author suggests, to solve this problem, an array of booleans, each indicating the status of those locations which they which to access. By moving past these regions and checking back later, the problem is solved, according to the author.
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

Summary

*Analysis of Programs for Parallel Processing* by A.J. Bernstein analyzes whether or not a program that is divided and performed in parallel gives the same results as that of a program that is run without parallelizing. According to Bernstein, even though the majority of programs written nowadays are written in a sequential manner, many people have the desire to process it in parallel. This is because a significant amount of time can be saved if certain tasks are performed concurrently with the main part of the program. Bernstein then uses a computer architecture diagram to demonstrate how parallel programming looks like in perspective of main memory, processors, and slave memory (consists of information meant to be stored). The author lastly lists the conditions in which a program can be parallelized by using regular expressions.
Announcement

• Homework 4 due date has been extended to Sunday, 11:59pm
• Homework 5 will be made available 29 February; it will be due on March 7
• You’ll have the entirety of the last week of classes to work on the final project.
• Article Review is due this coming-up Sunday.

The paper accurately conveyed the idea it was trying to, but would have been easier to follow if the general readers/writers solutions was written earlier in the paper as it elucidated the rest of the paper. Perhaps it would have been better to use binary notation instead of decimal notation in the examples, as this is more common in Computer Science, especially at lower levels. It would have been helpful to have some actual C-esq or Java-esq pseudo-code, which is more common in more recent papers. The pseudo-code that was provided had many logic symbols, which some may find harder to parse than standard pseudo-code. Otherwise this paper was copacetic, and hard to find any issue with.
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

The paper accurately conveyed the idea it was trying to, but would have been easier to follow if the general readers/writers solutions was written earlier in the paper as it elucidated the rest of the paper. Perhaps it would have been better to use binary notation instead of decimal notation in the examples, as this is more common in Computer Science, especially at lower levels. It would have been helpful to have some actual C-esq or Java-esq pseudo-code, which is more common in more recent papers. The pseudo-code that was provided had many logic symbols, which some may find harder to parse than standard pseudo-code. Otherwise this paper was copacetic, and hard to find any issue with.

Provides concrete, constructive feedback that could be used to improve the article
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

Lab 5: why does a threaded version of “the” program take longer to run than the non-threaded?
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

**Lab 5: why does a threaded version of “the” program take longer to run than the non-threaded?**

<table>
<thead>
<tr>
<th>for (y : 1 to 8) performCalc</th>
<th>co [y : 1 to 8] performCalc</th>
</tr>
</thead>
</table>

If `performCalc` is a 1 second calculation, then how long will the serial execution take?

If `performCalc` is a 1 second calculation, then how long will the concurrent (threaded) execution take, assuming that 0.6 seconds are needed to create each thread and 0.6 seconds to delete each thread?
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

Lab 5: why does a threaded version of “the” program take longer to run than the non-threaded?

```plaintext
for (y : 1 to 8) performCalc
```

If `performCalc` is a 1 second calculation, then how long will the serial execution take?

1 * 8 = 8 seconds

```plaintext
co [y : 1 to 8] performCalc
```

If `performCalc` is a 1 second calculation, then how long will the concurrent (threaded) execution take, assuming that 0.6 seconds are needed to create each thread and 0.6 seconds to delete each thread?

0.6 * 8 + 1 + 0.6 * 8 = 10.6 seconds
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

Lab 5: why does a threaded version of “the” program take longer to run than the non-threaded?

“the overhead of creating and managing a thread adds more work than performing the simple operation in a thread”

“the overhead of threading every single iteration ends up overcoming whatever tiny gains we made by having multiple CPUs handle these tiny iterations”
Announcement

- Homework 4 due date has been extended to Sunday, 11:59pm
- Homework 5 will be made available 29 February; it will be due on March 7
- You’ll have the entirety of the last week of classes to work on the final project.
- Article Review is due this coming-up Sunday.

Lab 5: why does a threaded version of “the” program take longer to run than the non-threaded?

```
for (y : 1 to 8)
  performCalc
```

If `performCalc` invokes a static or singleton method ...
Announcement – Final Project

- **Motivate** the need for a concurrent implementation of an algorithm of your choice
- **Implement** the solution using the language of your choice
- **Report** on the results (success/failure of the implementation as judged using run-time results)
- **Propose** next steps

- Submit a **one paragraph** project idea
- Report must contain at a minimum the sections specified in the final project handout

- Done individually or in groups of 2.
- Project “idea” due first
- Final report due via Canvas on 11 March
Sample Final Project Ideas

An exploration of solving common summations using threaded solutions compared to direct algebraic methods. Solutions will be provided in the C++ language. Summations are prime targets for optimization as each iteration is usually independent and can therefore be run in parallel without semaphores or mutexes, however the results still need to be summed eventually, usually with calls to the std::future interface. Series included will be Harmonic numbers, Arithmetic Series, Geometric Series, and possibly a Trigonometric series. Sums will be carefully optimized. Results will be displayed in a simple line graph format.

\[ \sum_{k=a}^{b} r^k = \frac{r^a - r^{b+1}}{1 - r}, \]

Consider the above Geometric series: is it faster to simply sum the left hand side or solve the right hand side? Could threading allow the left hand equation to have an edge? This will be explored in the paper at length, and maybe we'll get to explore different compiler optimization settings when applied to this problem if time permits (perhaps the compiler itself transforms the right hand side into the left hand side or vice-versa in single threaded situations).
Sample Final Project Ideas

An exploration of solving common summations using threaded solutions compared to direct algebraic methods. Solutions will be provided in the C++ language. Summations are prime targets for optimization as each iteration is usually independent and can therefore be run in parallel without semaphores or mutexes, however the results still need to be summed eventually, usually with calls to the std::future interface. Series included will be Harmonic numbers, Arithmetic Series, Geometric Series, and possibly a Trigonometric series. Sums will be carefully optimized. Results will be displayed in a simple line graph format.

\[ \sum_{k=a}^{b} r^k = \frac{r^a - r^{b+1}}{1 - r}, \]

Consider the above Geometric side? Could threading allow the leftover or maybe we’ll get to explore different permutations (perhaps the compiler itself thread situations).

For my project I am planning to calculate if a number is prime concurrently using the trial division method for testing for prime numbers. How trial division works is to test if a number p is prime, I would check all numbers between 2 and \( \sqrt{p} \). The trial division can be concurrently run by breaking the problem into the different segments between 2 and \( \sqrt{p} \) based on the number of processors the computer has. If the program detects if a number is not prime it would signal the other threads and would then move on to the next prime number calculation. For my high level structure I would require a way for each thread to check if the other threads disproved if a number is prime or not.
Sample Final Project Ideas

An exploration of solving common summations using threaded solutions compared to direct algebraic methods. Solutions will be provided in the C++ language. Summations are prime targets for optimization as each iteration is usually independent and can therefore be run in parallel without semaphores or mutexes, however the results still need to be summed eventually, usually with calls to the std::future interface. Series included will be Harmonic numbers, Arithmetic Series, Geometric Series, and possibly a Trigonometric series. Sums will be carefully optimized. Results will be displayed in a simple line graph format.

\[ \sum_{k=a}^{b} r^k = \frac{r^a - r^{b+1}}{1 - r}, \]

Consider the above Geometric series or side? Could threading allow the limit to be reached faster and maybe we'll get to explore different compilers and compilers (perhaps the compiler itself thread situations).

For my project I am planning to calculate if a number is prime concurrently using the trial division method for testing for prime numbers. How trial division works is to test if a number is prime, I would check all numbers between 2 and \( \sqrt{p} \). The trial division can be concurrently run by breaking the problem into the different segments between 2 and \( \sqrt{p} \) based on the number of processors the computer has. If the program detects if a number is not prime, it would signal the other threads and would then move on to the next prime number calculation. For my high level structure, I would require a way for each thread to check if the other threads disproved if a number is prime or not.

I plan on doing my project on using concurrency to optimize the run-time of the merge sort algorithm. I choose the merge sort algorithm, because the algorithm seems to be a natural fit for optimization through concurrency. Merge sort splits up a problem into multiple sub problems, which I believe could be accomplished by multiple threads. Sorting is very commonly used, and thus I feel like the optimization of a sorting algorithm could prove to be quite useful.
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 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) {}  
        in1 = true;
        critical section;
        in1 = false;
        noncritical section;
    }
}
```

Q: What is the major shortcoming of this approach?

Global Variables

```plaintext
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){}
        in1=true;
        critical section;
        in1=false;
        noncritical section;
    }
}
```

Global Variables
- in1 = false
- in2 = false

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

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

```
semaphore
value = 0
```

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

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.

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?
Test and Set

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

<table>
<thead>
<tr>
<th>bool lock = \texttt{false};</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool TS(bool lock)</td>
</tr>
<tr>
<td>\textless bool initial = lock;</td>
</tr>
<tr>
<td>lock = true;</td>
</tr>
<tr>
<td>return initial;</td>
</tr>
<tr>
<td>\textgreater</td>
</tr>
</tbody>
</table>

```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;
    }
}
```

At the onset, the value of \texttt{lock}, a shared variable, is set to 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;
    }
}
```

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

```c
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;
    >
}

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

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;
}
```

```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
- 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

```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

Q: What is the importance of TS being atomic?
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: Does each process maintain its important properties (mutual exclusion, absence of deadlock, avoiding unnecessary delay, eventual entry)?
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;
    }
}

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;
}
```

```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?
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;
    }
}

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

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

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

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

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

```cpp
process CS2{
    while (true) {
        while (lock) {};
        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?)
Test and Test and Set

**bool lock = false;**

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

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

```cpp
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.**
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;
    }
}

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 ...
Tie-Breaker

```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 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):

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

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

In class exercise: 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?
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?
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?

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

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

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.

```plaintext
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.
Up Next

Barriers