CSCI 322
Principles of Concurrent Programming

Filip Jagodzinski
Announcements

Midterm Exam and HW1

• Will be returned on Tuesday

Monday, 18 July

• No lecture
Announcements

HW2

• Is available on the course website

• Due 25 July (Monday, last week of class)

• Similar in structure to HW1; includes book “questions” and a programming task.

• Demo of non-threaded program
From last time

Control Dependence

```c
a = [ -2, -3, 3, 4, 5, 6 ]
for (i=1; i<n; i++){
    if (a[i-1] < 0){
        a[i] = -1;
    }
}
```
From last time

Control Dependence

Thread 1

```java
int i = 0;
for (i=1; i<n; i++){
    if (a[i-1] < 0){
        a[i] = -1;
    }
}
```

Thread 2

```java
int i = 3;
for (i=3; i<n; i++){
    if (a[i-1] < 0){
        a[i] = -1;
    }
}
```

a = [-2, -3, 3, 4, 5, 6]  
i = 0, 1, 2, 3, 4, 5  

If Thread 1 < Thread 2

If Thread 2 < Thread 1
From last time

Control Dependence

```
a = [-2, -3, 3, 4, 5, 6]
for (i=1; i<n; i++){
    if (a[i-1] < 0){
        a[i] = -1;
    }
}
```

If Thread 1 < Thread 2

```
a = [-2, -1, -1, -1, -1, -1]
```

If Thread 2 < Thread 1

```
a = [-2, -1, -1, 4, 5, 6]
```

---

```
i = 0 1 2 3 4 5
a = [-2, -3, 3, 4, 5, 6]
```

<table>
<thead>
<tr>
<th>Thread 1</th>
</tr>
</thead>
</table>
| for (i=1; i<3; i++){
    if (a[i-1] < 0){
        a[i] = -1;
    }
}
|

<table>
<thead>
<tr>
<th>Thread 2</th>
</tr>
</thead>
</table>
| for (i=3; i<n; i++){
    if (a[i-1] < 0){
        a[i] = -1;
    }
}
|
From last time

Control Dependence

\[
a = [-2, -3, 3, 4, 5, 6]
\]

for (i=1; i<n; i++){  
    if (a[i-1] < 0){  
        a[i] = -1;  
    }  
}

Previously we’ve looked at race conditions (when two or more entities are “trying” to simultaneously access a shared variable), or when too much parallelism is attempted (multiple threads receive the “same” for loop iterations) and threads write over the outputs of other threads.

Here, even if Threads 1 and 2 run and finish on separate days (no race conditions), AND the threads are assigned different \( i \) values, dependence still exists among the Threads due to the LOGIC of the code.
From last time

Q: How would you optimize this code?

S0: y = 78;
S1: x = y + 32 -16;
S2: y = 32 + 45.1 - 77;
S3: x = 45 * 12.7;
From last time

Q: How would you optimize this code?

This approach is suitable if the program is executed non-concurrently.

But if the code is executed concurrently, what can go wrong?
Q: How would you optimize this code?

```c
int j =2;
int k = 3;
for (int i = 0; i<N; i+=2){
    a[i] = i * j+k;
}
```
From last time

Q: How would you optimize this code?
Q: How many operations and loads are performed?

int j = 2;
int k = 3;
for (int i = 0; i < N; i += 2) {
    a[i] = i * j + k;
}
Q: How would you optimize this code?
Q: How many operations and loads are performed?

```java
int j = 2;
int k = 3;
for (int i = 0; i < N; i += 2) {
    a[i] = i * j + k;
}
```

Two operations and 4 loads
One assignment
From last time

Q: How would you optimize this code?
Q: How many operations and loads are performed?

```
int j = 2;
int k = 3;
for (int i = 0; i<N; i+=2){
    a[i] = i * j + k;
}
```

Two operations and 4 loads
One assignment

```
int j = 2;
int k = 3;
int m = k;
int n = 2 * j;
for (int i = 0; i<N; i+=2){
    a[i] = m;
    m = m + n;
}
```

One operation and 3 loads
Two assignments
## From last time

### Test promotion

### Loop peeling

For each of these, be able to define/explain the procedure, and be able to perform the specified task (for example, “perform loop peeling to improve this code”)

### Loop fission
From last time

Test promotion (loop unswitching): move a loop-independent test OUT of the loop

Loop peeling

Loop fission
From last time

Test promotion (loop unswitching) :
move a loop-independent test OUT of the loop

Loop peeling : If numerical code deals with boundary conditions in the first or last iterations of a loop, then those components can be taken out of the loop

Loop fission
From last time

Test promotion (loop unswitching): move a loop-independent test OUT of the loop

Loop peeling: If numerical code deals with boundary conditions in the first or last iterations of a loop, then those components can be taken out of the loop

Loop fission: a loop is broken into multiple “smaller” loops over the same index range, and each “smaller” loop performs only a part of the original loop’s body
Need for Barrier-like mechanisms
Critical Sections
Code Optimization

```java
int a[], b[], c[], d[];
for (i=0; i<600; i++){
    a[i] = a[i] + b[i];
    b[i] = c[i] * x + y;
    c[i] = 1/b[i];
    d[i] = Math.sqrt(c[i]);
}
```
Code Optimization

```java
int a[], b[], c[], d[];
for (i=0; i<600; i++){
    a[i] = a[i] + b[i];
    b[i] = c[i]*x + y;
    c[i] = 1/b[i];
    d[i] = Math.sqrt(c[i]);
}
```

Each node corresponds to a fission
The “solution” informs where fission can be performed, but it does NOT inform you the order of execution of the “fissioned” for loops.

Q: Is the above the correct ordering of loops?
Q: Should the yellow loop proceed before or after the green loop?
At the $i$th iteration of the for loop, S1 reads from $b[i]$, and then immediately $b[i]$ is altered (S2).
If the green loop were to complete on a thread before the yellow loop begins, then the original code (left) might produce an array \( a \) that is different than the code on the right.

Q: How should the fused loops be executed?
If the green loop were to complete on a thread before the yellow loop begins, then the original code (left) might produce an array `a` that is different than the code on the right.

Q: How should the fused loops be executed?

The yellow loop should proceed first, then the green loop, then the blue loop.

Q: Must the entirety of the yellow loop complete prior to the green loop starting?
Code Optimization

In this single for loop approach ...

```java
int a[], b[], c[], d[];
for (i=0; i<600; i++){
    b[i] = c[i] * x + y;
    c[i] = 1/b[i];
}
for (i=0; i<600; i++){
    a[i] = a[i] + b[i];
}
for (i=0; i<600; i++){
    d[i] = Math.sqrt(c[i]);
}
```
In this single for loop approach ...

First a[0] is calculated
In this single for loop approach ...

Then b[0]
In this single for loop approach ...

Followed by c[0] and d[0]
In order for the loop fissioned approach to produce the same a, b, c, and d arrays as the non-threaded version, calculating the \(i\)-th entry for \(a\) must occur before the \(i\)-th entry for \(b\) is calculated, and the \(i\)-th entry for \(c\) must happen after the \(i\)-th entry of \(b\) is calculated. After the green loop completes its calculation for the \(i\)-th position, only then can the \(i\)-th entry of \(d\) be calculated.
In this single for loop approach ...

Q: Are the calculations of the rows of a specific column independent?

Q: Are the calculations (of all the rows) among different columns independent?
Our goal then is to parallelize this code as much as possible, but still make it “safe”

Q: How would you do this?
Q: What factors must you consider?
If your computer has 3n processors, you could safely invoke a unique i for each of the three loops.

If \( n = 600 \), how would you do it?

(on the board discussion)
Code Optimization

If your computer has $3n$ processors, you could safely invoke a unique $i$ for each of the three loops.
If your computer has $3n$ processors, you could safely invoke a unique $i$ for each of the three loops.

Q: Is this a correct approach?
Q: Is this approach time optimized?

Q: If $n=10,000$, how many of us have access to a 30,000 core/CPU computer?
Another approach is the following (because who has access to computer with 3n processors?) ...
Another approach is the following (because who has access to computer with 3n processors?) ...

Perform all of the $i$s for $S2, S3$
Another approach is the following (because who has access to computer with 3n processors?) ...

Q: Does this produce the “correct” results?
Q: Is it time optimal?
Code Optimization

We want something like this ...

```
int a[], b[], c[], d[];
for (i=0; i<600; i++){
    b[i] = c[i] * x + y;
    c[i] = 1/b[i];
}
for (i=0; i<600; i++){
    a[i] = a[i] + b[i];
}
for (i=0; i<600; i++){
    d[i] = Math.sqrt(c[i]);
}
```
Q: Even if the execution of the yellow loop begins before the execution of the green loop, and the green loop begins executing before the blue loop, what issues might arise? And why?
Not only might the speeds differ of the CPUs where the green, yellow, and blue loops are executing, but the calculation being performed by each CPU might vary in complexity.
Even if time $m$ is before time $n$, and $n$ is before time $r$, that does not guarantee that the execution of the green for loop will not overtake the execution of the yellow loop.
We’ve looked closely at semaphores, and threading, both in practice and in theory … that includes mutual exclusion (mutex) and condition synchronization (await)

Before we can tackle this problem …

We will explore critical sections, and how to program these kinds of synchronization (mutex and await) … then on to barriers
The Critical Section Problem

In the critical section problem \( n \) processes repeatedly execute a critical then a non-critical section of code.

Q: What is a critical section? What is a non-critical section?
In the critical section problem \( n \) processes repeatedly execute a critical then a non-critical section of code.

A sequence of statements that access a shared resource (variable, data structure, data on a shared drive)
The Critical Section Problem

In the critical section problem $n$ processes repeatedly execute a critical then a non-critical section of code.

A sequence of statements that access a shared resource (variable, data structure, data on a shared drive)

```java
process CS[i=1 to n]{
    while(true){
        entry protocol;
        critical section;
        exit protocol;
        noncritical section;
    }
}
```
The Critical Section Problem

- Each critical section can be accessed by only one processes at a time

Goal

Q: What safety and liveness properties should the entry (and exit) protocols impose, such that the goal is met?

process CS[i=1 to n]{
    while(true){
        entry protocol;
        critical section;
        exit protocol;
        noncritical section;
    }
}
The Critical Section Problem

- Each critical section can be accessed by only one process at a time

Goal

Q: What safety and liveness properties should the entry (and exit) protocols impose, such that the goal is met?

Safety (nothing BAD ever happens)
- Final state is correct
- Mutual exclusion
- No deadlock

Liveness (something GOOD eventually happens)
- Program terminates
- Process eventually enters critical section
- A request for service will eventually be honored
- A message will reach its destination

```java
process CS[i=1 to n]{
    while(true){
        entry protocol;
        critical section;
        exit protocol;
        noncritical section;
    }
}
```
The Critical Section Problem

- 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:
  - Absence of Deadlock
  
  - Absence of Unnecessary delay:
  
  - Eventual Entry:

```
process CS[i=1 to n] {
  while (true) {
    entry protocol;
    critical section;
    exit protocol;
    noncritical section;
  }
}
```

You will need to know these (memorized) for the final exam

Task: Define these in the context of critical sections
The Critical Section Problem

- 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

```
process CS[i=1 to n]{
    while(true){
        entry protocol;
        critical section;
        exit protocol;
        noncritical section;
    }
}
```

Assumption: A process that enters its critical section will eventually exit

Q: How do we implement the entry and exit protocols?
The Critical Section Problem

- 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

```java
process CS[i=1 to n]{
    while(true){
        entry protocol;
        critical section;
        exit protocol;
        noncritical section;
    }
}
```

Assumption: a process that enters its critical section will eventually exit

Q: How do we implement the entry and exit protocols?
The Critical Section Problem

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

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

**In class exercise (worksheet) 1**: Add entry and exit protocols to both processes so that their critical sections exhibit mutual exclusion (only one of the critical sections can be executing at any one time).

The while loops of both CS1 and CS2 should be *nearly* identical.

You MAY add additional code (global variables) that are shared among the two processes.
The Critical Section Problem

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

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

Global Variables
in1 = false
in2 = false

Sample solution
The Critical Section Problem

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

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

Global Variables

in1 = false
in2 = false

Q: In order for global variables in1 and in2 to impose mutex (access to the critical section), what condition(s) must be met?
The Critical Section Problem

Global Variables

in1 = false
in2 = false

Q: In order for global variables in1 and in2 to impose mutex (access to the critical section), what condition(s) must be met?

¬(in1 ∧ in2)
The Critical Section Problem

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

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

Global Variables

in1 = false
in2 = false

Task: Identify the entry and exit protocols in the above code
The Critical Section Problem

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

process CS2{
  while (true) {
    while (in1) {
      in1 = true;
      critical section;
      in1 = false;
    }
    noncritical section;
  }
}

Global Variables
in1 = false
in2 = false

Notice: Aside from the 2 processes having different names, and relying on different variables (in1 and in2), they both are running the EXACT same critical section code.
The Critical Section Problem

Q: What is the behavior of this while loop, when \texttt{in2} is false? When \texttt{in2} is true?
Q: What is the behavior of this while loop, when \texttt{in2} is false? When \texttt{in2} is true?

When \texttt{in2} is true, this while loop’s body, which is empty, will loop indefinitely, preventing entry into the critical section of CS1

The \texttt{in2} specifies if CS2 is in its critical section
The Critical Section Problem

Q: What is the behavior of this while loop, when `in2` is false? When `in2` is true?

When `in2` is false, what sequence of actions ensues?
The Critical Section Problem

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

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

Global Variables

- `bool in1 = false`
- `bool in2 = false`

When `in2` is false, what sequence of actions ensues?

- The while terminates
The Critical Section Problem

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

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

Global Variables

```
bool in1 = true
bool in2 = false
```

When in2 is false, what sequence of actions ensues?

- The while terminates
- in1 is set to true ... this prevents CS2 from entering its critical section
The Critical Section Problem

Global Variables

bool in1 = true
bool in2 = false

When \( \text{in2} \) is false, what sequence of actions ensues?

- The while terminates
- \( \text{in1} \) is set to true ... this prevents CS2 from entering its critical section
- CS1 enters its critical section
The Critical Section Problem

Global Variables

```plaintext
bool in1 = false
bool in2 = false
```

When `in2` is false, what sequence of actions ensues?

- The while terminates
- `in1` is set to true ... this prevents CS2 from entering its critical section
- CS1 enters its critical section
- CS1 sets the value of `in1` to false ... which allows CS2 to enter its critical section
The Critical Section Problem

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

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

Global Variables

bool in1 = false
bool in2 = false

When \texttt{in2} is false, what sequence of actions ensues?

- The while terminates
- \texttt{in1} is set to true ... this prevents CS2 from entering its critical section
- CS1 enters its critical section
- CS1 sets the value of \texttt{in1} to false ... which allows CS2 to enter its critical section
- CS1 performs its non critical section; proceeds to execute again its interior while
The Critical Section Problem

Global Variables

```cpp
bool in1 = false
bool in2 = false
```

**Task**: Rewrite the code shown in blue using `await`
The Critical Section Problem

```c
process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}
```

```c
process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}
```

Global Variables

```
bool in1 = false
bool in2 = false
```

The `while(in2){}` **spin loop** is the implementation of the **await statement** shown in CS1, and the `while(in1){}` **spin loop** is the implementation of the **await statement** shown in CS2.

**Q:** Does this code satisfy our liveness and safety properties?
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

- **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

Global Variables

bool in1 = false
bool in2 = false
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

- **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

Global Variables

```
bool in1 = false
bool in2 = false
```

Q: Does this code satisfy this property?
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

Global Variables
bool in1 = false
bool in2 = false

- **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
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

Global Variables
bool in1 = false
bool in2 = false

Q: Does this code satisfy this property?

• 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
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

• **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

Global Variables
bool in1 = false
bool in2 = false
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

- **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

Global Variables

bool in1 = false
bool in2 = false

Q: Does this code satisfy this property?
The Critical Section Problem

process CS1{
    while (true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while (true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

- **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

**Global Variables**

```
bool in1 = false
bool in2 = false
```
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

- **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

Global Variables
bool in1 = false
bool in2 = false

Q: Does this code satisfy this property?
The Critical Section Problem

process CS1{
    while(true){
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

- **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

Global Variables
- bool in1 = false
- bool in2 = false

Q: What happens if CS1 is MUCH faster than CS2? In that case, are these conditions still met?
The Critical Section Problem

process CS1{
    while (true) {
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while (true) {
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

Global Variables
bool in1 = false
bool in2 = false

Q: What happens if CS1 is MUCH faster than CS2? In that case, are these conditions still met?

- **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
The Critical Section Problem

process CS1{
    while (true) {
        <await (!in2) in1=true;>
        critical section;
        in1=false;
        noncritical section;
    }
}

process CS2{
    while (true) {
        <await (!in1) in2=true;>
        critical section;
        in2=false;
        noncritical section;
    }
}

Observations / Questions

Q: How many variables are needed when we have 2 processes?
Q: How many variables are needed when we have \( n \) processes?

Assuming \( n \) variables are needed to coordinate the entry/exit of \( n \) critical regions of \( n \) processes, is that practical? Scalable?
The Critical Section Problem

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

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

In class exercise (worksheet) 2: Add an entry and exit protocol to both processes so that their critical sections exhibit mutual exclusion (only one of the critical sections can be executing at any one time).

The while loops of both CS1 and CS2 MUST be identical.

Hint: Use only a SINGLE global variable
The Critical Section Problem

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

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

Global Variable

bool lock = false

Task: explain the behavior of both CS1 and CS2 when

lock = true
lock = false
The Critical Section Problem

Q: What are the await statements that the two interior while loops implement?

Global Variable

```
bool lock = false
```

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

```
process CS2{
    while (true) {
        while (lock) {}  // await lock
        lock = true;
        critical section;
        lock = false;
        noncritical section;
    }
}
```
The Critical Section Problem

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

Global Variable
bool lock = false

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

Q: What are the await statements that the two interior while loops implement?

process CS1{
    while(true){
        <await(!lock) lock=true;>
        critical section;
        lock=false;
        noncritical section;
    }
}

process CS2{
    while(true){
        <await(!lock) lock=true;>
        critical section;
        lock=false;
        noncritical section;
    }
}
The Critical Section Problem

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

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

Although in “theory” this would work ...

Q: What is a scenario when this approach won’t work? (hint : Think about concurrency and the time needed for multiple processes to retrieve the value of a variable)
The Critical Section Problem

Although in “theory” this would work ...

Q: What is a scenario when this approach won’t work? (hint : Think about concurrency and the time needed for multiple processes to retrieve the value of a variable)

Q: What is the solution?
The Critical Section Problem

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

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

One such example is Test and Set

```cpp
bool TS(bool lock) {
    bool initial = lock;
    lock = true;
    return initial;
}
```
The Critical Section Problem

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

One such example is Test and Set

```plaintext
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?
The Critical Section Problem

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

One such example is Test and Set

```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?

Q: Does this solve our issue with concurrent programs trying get the value of a lock variable?

Take-home Task: rewrite the code for CS1 and CS2 to use TS instead of while loops.
Lab (Thursday)