CSCI 322
Principles of Concurrent Programming

Filip Jagodzinski
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
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

A. Submit a **one paragraph** project idea
B. Submit proposal report, along with code (appendix), and required sections.

• Done individually or in groups of 2.
• Check Canvas for due dates.
For last Friday’s lab (due this week), you are asked to implement 4 different runs of the ridiculousCalculation, such that:

- Not optimized, Not threaded, slow
- Optimized, Not threaded, slow
- Not optimized, threaded, slow
- Optimized, threaded, fast
From last time

**Control Dependence**: an instruction is control dependent on a preceding instruction if the output of the latter can determine if the former should be executed.

![Code Example]

S7 is control dependent on S6
Removing redundant code may increase the number of dependencies, because now S3 is dependent on S1 AND S0, while in the previous case (code on the left), S3 was dependent on ONLY S0.

The use of waits, semaphores, etc. to impose these constraints might consume resources that will negate any improvements that you might expect by using threads.
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
From last time

Q: What is an algorithm to determine how fission should (can?) be performed on the following statements (S1 through S4)?

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

The “solution” may not be unique (here the order of execution for S4 or S1 can be interchanged), but at least we know that the code in green MUST precede both S1 and S4.
Today

Locks and Barriers
(Chapter 3 of textbook)
Locks and Barriers

We’ve looked closely at semaphores, and threading, both in practice and in theory ... that includes mutual exclusion (mutex) and condition synchronization (await)

In this part of the course, we will explore critical sections and barriers, and how to program these kinds of synchronization (mutex and await)
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?
Q: What is a non-critical section?
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)
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 process 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

```plaintext
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:

```plaintext
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.
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

```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

- 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

```
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

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

Global Variables

\[\text{in1} = \text{false}\]
\[\text{in2} = \text{false}\]

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

\begin{align*}
\text{in1} &= \text{false} \\
\text{in2} &= \text{false}
\end{align*}

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

\[ \neg (\text{in1} \land \text{in2}) \]
The Critical Section Problem

Task: Identify the entry and exit protocols in the above code

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
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

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 in2 is false? When in2 is true?
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;
    }
}

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

Global Variables

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

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

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

The in2 specifies if CS2 is in its critical section
The Critical Section Problem

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

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

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

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

Global Variables

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

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

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

Global Variables

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

When \( \text{in2} \) 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

Q: How does this affect the behavior of CS2?
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 \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
The Critical Section Problem

Global Variables

bool in1 = true
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
The Critical Section Problem

```cpp
process CS1{
    while (true) {
        while (in2) {} // cs2 enters its critical section
        in1=true;
        critical section;
        in1=false;
        noncritical section;
    }
}
```

```cpp
process CS2{
    while (true) {
        while (in1) {} // cs1 enters its critical section
        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
- `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 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
• CS1 performs its non critical section; proceeds to execute again its interior while
The Critical Section Problem

Global Variables

bool in1 = false
bool in2 = false

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

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.

Does this code satisfy the liveness and safety properties we want imposed?
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

**Does this code satisfy the liveness and safety properties we want imposed?**
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

boolean in1 = false
boolean in2 = false

Does this code satisfy the liveness and safety properties we want imposed?
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

Does this code satisfy the liveness and safety properties we want imposed?

• **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

• **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

Does this code satisfy the liveness and safety properties we want imposed?
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

Does this code satisfy the liveness and safety properties we want imposed?
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

Does this code satisfy the liveness and safety properties we want imposed?
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

Does this code satisfy the liveness and safety properties we want imposed?
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

Does this code satisfy the liveness and safety properties we want imposed?
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;
  }
}

**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

```java
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

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?

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
The Critical Section Problem

Global Variable

bool lock = false

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

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;
    }
}

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

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

```java
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)

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;
}
```

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

```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?

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.
Up Next

Locks and Barriers