Announcements

Homework 2 : Due tomorrow, Thursday
Homework 3 : Due Tuesday (2 book questions, no coding)

Friday, 5 February : Lab (due in 2 weeks)
Monday, 8 February : Lecture
Tuesday, 9 February : Lecture
Wednesday, 10 February : Exam Prep

By Wednesday, you will have received/heard

- HW 1 solution set
- HW 2 solution set
- HW 3 solution set
- Sample midterm solution set (in class)
- Have had a half dozen in-class exercises

Friday, 12 February : Midterm
From last time

```cpp
string buffer;  // contains one line of input
bool done = false;  // used to signal termination

c0  // process 1: find patterns
    string line1;
    while (true) {
        wait for buffer to be full or done to be true;
        if (done) break;
        line1 = buffer;
        signal that buffer is empty;
        look for pattern in line1;
        if (pattern is in line1)
            write line1;
    }

    // process 2: read new lines
    string line2;
    while (true) {
        read next line of input into line2;
        if (EOF) {done = true; break; }
        wait for buffer to be empty;
        buffer = line2;
        signal that buffer is full;
    }
```

From last time

```c
string buffer;  // contains one line of input
bool done = false;  // used to signal termination

do  // process 1: find patterns
    string line1;
    while (true) {
        wait for buffer to be full or done to be true;
        if (done) break;
        line1 = buffer;
        signal that buffer is empty;
        look for pattern in line1;
        if (pattern is in line1)
            write line1;
    }

    // process 2: read new lines
    string line2;
    while (true) {
        read next line of input into line2;
        if (EOF) {done = true; break;}
        wait for buffer to be empty;
        buffer = line2;
        signal that buffer is full;
    }

    oc;

In the coming weeks we will learn which constructs to use and what code to write that accomplishes these tasks...
```
From last time

In the coming weeks we will learn which constructs to use and what code to write that accomplishes these tasks...

which enforces that only one of these critical sections is being executed at any one time

```java
string buffer;  # contains one line of input
bool done = false;  # used to signal termination

// # process 1: find patterns
co  # process 1: find patterns
string line1;
while (true) {
    wait for buffer to be full or done to be true;
    if (done) break;
    line1 = buffer;
    signal that buffer is empty;
    look for pattern in line1;
    if (pattern is in line1)
        write line1;
}

// # process 2: read new lines
string line2;
while (true) {
    read next line of input into line2;
    if (EOF) {done = true; break; }
    wait for buffer to be empty;
    buffer = line2;
    signal that buffer is full;
}
```
From last time

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

Q: What does this code accomplish?

Q: How might it be implemented concurrently?
From last time

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

Q: What does this code accomplish?

Q: How might it be implemented concurrently?

```c
int m = 0;
co [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}
```

Task: Be able to explain WHAT this pseudocode specifies, and why/how/if it is better than the sequential execution.
Assume the following two code statements

\[ S1 : x = x + y; \]
\[ S2 : y = x \times y; \]

And that the variables \( x \) and \( y \) have initial values \( x=3 \) and \( y=4 \). Then, what is the final state (values for \( x \) and \( y \)) for the following program \( P1 \) (assume \( S1 \) and \( S2 \) are executed atomically):

\[ P1 : S1; S2; \]

In class exercise
Further refining concurrent code
Atomic actions
Awaits
Programming Logic
Synchronization

Because all processes might want to write to \( m \) at the same time, the final value of \( m \) will be the value of \( a[i] \) assigned by the last process that is scheduled to update \( m \).

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

Because here we have “too much” parallelization, combine separate actions into a single, atomic one.

```c
int m = 0;
for [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}
```
Synchronization

A possible solution. The `< >` brackets specify that EACH concurrent arm should inspect and write to m as a single action.

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

```c
int m = 0;
cor [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}
```

Q: What are the consequences of making each concurrent thread run its code atomically?
Synchronization

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

A possible solution. The `< >` brackets specify that EACH concurrent arm should inspect and write to `m` as a single action.

```c
int m = 0;
co [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}
```

Each `a[i]` entry is examined in some arbitrary order because the OS schedules each thread to run concurrently.
Synchronization

int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];

A possible solution. The < > brackets specify that EACH concurrent arm should inspect and write to m as a single action.

int m = 0;
co [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}

Each a[i] entry is examined in some arbitrary order because the OS schedules each thread to run concurrently.

Q: Is this any better than the serial solution? (could it be worse? Why or why not?)
Synchronization

int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];

A possible solution. The < > brackets specify that EACH concurrent arm should inspect and write to m as a single action.

int m = 0;
co [i = 0 to n-1] { 
    if (a[i] > m)
        m = a[i];
}

Each a[i] entry is examined in some arbitrary order because the OS schedules each thread to run concurrently

No. It might actually be worse due to the overhead needed in creating separate threads for executing each arm concurrently

Q: How might we then “fix” this poorly efficient concurrent program?
Synchronization

```
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

If the angled brackets are moved, how does that change the program?

Task: Be able to explain how/why these two programs are or are not different
Synchronization

Each arm can inspect its assigned entry in $a$, and then update the value of $m$ atomically, but the architecture already forces that to happen because writing TO a data entry in memory can be performed by one process at a time.

On the board explanation

```c
int m = 0;
for [i = 0 to n-1] {
  if (a[i] > m)
    m = a[i];
}
```

```c
int m = 0;
co [i = 0 to n-1] {
  if (a[i] > m)
    m = a[i];
}
```

Q: What is the optimal solution?
Synchronization

int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];

int m = 0;
co [i = 0 to n-1] {
    if (a[i] > m)
        if (a[i] > m)
            m = a[i];
}

Task: Explain this optimal solution
Synchronization

```c
int m = 0;
for [i = 0 to n-1]
  if (a[i] > m)
    m = a[i];
```

Each process first performs a comparison concurrently; hence MANY processes might be trying to perform the check.
Synchronization

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

If the first comparison is true, perform a second comparison to make sure that ANOTHER process hasn’t updated the value of m in the meantime.

Q: Why is such an approach optimal? It seems counter intuitive ... back-to-back checks?
Synchronization

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
```

If the checks occur somewhat at random (and the scheduler more-or-less guarantees that), then it becomes increasingly likely that a process will not have to do a second check, and after the completion of the program, m is the maximum value.

Q: How is such behavior (atomicity) implemented in software?
Atomicity

**Atomic action**: Makes an *indivisible* state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes.
Atomicity

**Atomic action**: Makes an **indivisible** state transformation. Any intermediate state that might exist in the implementation of the action must **NOT** be visible to other processes.

\[
\text{int } y = 0, z = 0; \\
x = y+z; \quad y = 1; \quad z = 2;
\]

Assignment statements executed sequentially (without concurrency) appear to be atomic because no intermediate state is visible to the program.

If \( x = y+z \) is executed **BEFORE** \( y = 1 \) and \( z=2 \), then the result end state of the variables \( x \) and \( y \) and \( z \) is **ALWAYS** the same, regardless of how many times you run the program.
Atomicity

**Atomic action**: Makes an *indivisible* state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes

```c
int y = 0, z = 0;
co x = y+z;  // y = 1; z = 2; oc;
```

When executed concurrently, however, the assignment statements might be implemented by a sequence of fine-grained machine instructions, and depending on the execution history one of several (potentially many) different results may result. Thus in this case the assignment statement when executed concurrently are NOT atomic actions.

**Q: What are the possible final values of x if the code is run concurrently?**
Atomicity

Atomic action: Makes an **indivisible** state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes.

```c
int y = 0, z = 0;
co x = y+z;  // y = 1; z = 2; oc;
```

When executed concurrently, however, the assignment statements might be implemented by a sequence of fine-grained machine instructions, and depending on the execution history one of several (potentially many) different results may result. Thus in this case the assignment statement when executed concurrently are **NOT** atomic actions.

Q: What are the possible final values of x if the code is run concurrently? 0, 1, 2 or 3
Atomicity

**Atomic action**: Makes an *indivisible* state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes.

```c
int y = 0, z = 0;
co x = y+z; // y = 1; z = 2; oc;
```

Q: How might x=2 be the final state?

Q: What are the possible final values of x if the code is run concurrently? 0, 1, 2 or 3
Atomicity

Atomic action: Makes an **indivisible** state transformation. Any intermediate state that might exist in the implementation of the action must NOT be visible to other processes.

![Code snippet](image)

Keep in mind that $x = y + z$ is in itself a series of instructions:

- $i_1$: fetch $y$
- $i_2$: fetch $z$
- $i_3$: compute $x = y + z$
- $i_4$: write back $x$

**Q:** Assuming the following:

- $i_5$: $y = 1$
- $i_6$: $z = 2$

What is an instruction history such that $x$ has a final value of 2?

**Q:** How might $x = 2$ be the final state?

- 0, 1, **2** or 3
Assume the following two code statements

\[
\begin{align*}
S1 & : x = x + y; \\
S2 & : y = x \times y;
\end{align*}
\]

And that the variables \(x\) and \(y\) have initial values \(x=3\) and \(y=4\). Then, what is the final state (values for \(x\) and \(y\)) for the following program \(P2\) (assume \(S1\) and \(S2\) are executed atomically):

\[
P2 : \text{co } S1; \; // \; S2; \; \text{oc}
\]

In class exercise
Atomicity

When discussing concurrent programs, assume ...

- Values of basic types are stored in **memory elements** that are read and written as atomic actions
- Values are manipulated by **loading** into registers, **operating** on them, and **storing** the results back into memory
When discussing concurrent programs, assume ...

- Values of basic types are stored in **memory elements** that are read and written as atomic actions.
- Values are manipulated by **loading** into registers, **operating** on them, and **storing** the results back into memory.
- Each **process** has its **own set** of registers.
- Intermediate results when a complex expression is evaluated are stored in registers or private memory (private stack).
Atomicity

When discussing concurrent programs, assume ...

- Values of basic types are stored in memory elements that are read and written as atomic actions
- Values are manipulated by loading into registers, operating on them, and storing the results back into memory
- Each process has its own set of registers
- Intermediate results when a complex expression is evaluated are stored in registers or private memory (private stack)

Appearance of atomicity

- If evaluating expression e, one process does not reference a variable altered by another process
  - None of the values on which e depends could change during evaluation
  - No other process can see any temporary values that might be created
Atomicity

When discussing concurrent programs, assume ...

- Values of basic types are stored in **memory elements** that are read and written as atomic actions.
- Values are manipulated by **loading** into registers, **operating** on them, and **storing** the results back into memory.
- **Each process** has its own set of registers.
- Intermediate results when a complex expression is evaluated are stored in registers or private memory (private stack).

**Appearance of atomicity**

- If assignment $x = e$ does not reference any variable altered by another process then the assignment will appear atomic.

**Q:** So is there any hope in writing concurrent code in which variables are shared?
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process
- Assume a critical reference is to a simple variable (updating a value)
At-Most-Once Property

- A critical reference in an expression is a reference to a variable that is changed by another process.
- Assume a critical reference is to a simple variable (updating a value).
- An assignment $x = e$ is at-most-once if either
  - $e$ contains at MOST one critical reference and $x$ is not read by another process.
  - $e$ contains no critical references.
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process.
- Assume a critical reference is to a simple variable (updating a value).
- An assignment \( x = e \) is **at-most-once** if either
  - \( e \) contains at MOST one critical reference and \( x \) is not read by another process.
  - \( e \) contains no critical references.

```c
int x = 0, y = 0;
co x = x+1; // y = y+1; oc;

int x = 0, y = 0;
co x = y+1; // y = y+1; oc;

int x = 0, y = 0;
co x = y+1; // y = x+1; oc;
```

Q: Which of these are good concurrent programs, and why?
(determine which assignments are at-most-once)
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process.
- Assume a critical reference is to a simple variable (updating a value).
- An assignment `x = e` is **at-most-once** if either
  - `e` contains at most one critical reference and `x` is not read by another process.
  - `e` contains no critical references.

```
int x = 0, y = 0;
co x = x+1; // y = y+1; oc;
```

No critical references

**Q:** Does it matter which arm executes first?

```
int x = 0, y = 0;
co x = y+1; // y = y+1; oc;
```

```
int x = 0, y = 0;
co x = y+1; // y = x+1; oc;
```
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process.
- Assume a critical reference is to a simple variable (updating a value).
- An assignment \( x = e \) is **at-most-once** if either
  - \( e \) contains at most one critical reference and \( x \) is not read by another process.
  - \( e \) contains no critical references.

```
int x = 0, y = 0;
co x = x+1; // y = y+1; oc;
```

No critical references

Q: Does it matter which arm executes first?

No. If the left arm goes first, then the right arm, then the final value of \( x \) is 1 and \( y = 1 \). If the right arm goes first, then the left arm, then the final value of \( x \) is 1 and \( y \) is 1.

```
int x = 0, y = 0;
co x = y+1; // y = y+1; oc;
```

```
int x = 0, y = 0;
co x = y+1; // y = x+1; oc;
```

Thus both the left and right arm at **at-most-once**
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by **another** process.
- Assume a critical reference is to a simple variable (updating a value).
- An assignment \( x = e \) is **at-most-once** if either:
  - \( e \) contains at MOST one critical reference and \( x \) is not read by another process.
  - \( e \) contains no critical references.

```c
int x = 0, y = 0;
co x = x+1; // y = y+1; oc;
```

Q: How many critical references?

```c
int x = 0, y = 0;
co x = y+1; // y = y+1; oc;
```

```c
int x = 0, y = 0;
co x = y+1; // y = x+1; oc;
```
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process.
- Assume a critical reference is to a simple variable (updating a value).
- An assignment \( x = e \) is **at-most-once** if either
  - \( e \) contains at MOST one critical reference and \( x \) is not read by another process.
  - \( e \) contains no critical references.

```c
int x = 0, y = 0;
c o x = x+1; // y = y+1; oc;
```

Q: **How many critical references?** One
One LHS is ready by another
Q: **What are the possible values of** \( x \) **and** \( y \)?

```c
int x = 0, y = 0;
c o x = y+1; // y = y+1; oc;
```
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process
- Assume a critical reference is to a simple variable (updating a value)
- An assignment \( x = e \) is **at-most-once** if either
  - \( e \) contains at MOST one critical reference and \( x \) is not read by another process
  - \( e \) contains no critical references

```c
int x = 0, y = 0;
co x = x+1; // y = y+1; oc;
```

Q: How many critical references? One
One LHS is ready by another
Q: What are the possible values of \( x \) and \( y \)?
- \( x \) is 1 or 2, and \( y \) is 1

Thus both the left and right arm at **at-most-once**
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by **another** process
- Assume a critical reference is to a simple variable (updating a value)
- An assignment \( x = e \) is **at-most-once** if either
  - \( e \) contains at MOST one critical reference and \( x \) is not read by another process
  - \( e \) contains no critical references

```
int x = 0, y = 0;
c o x = x+1; // y = y+1; oc;
```

```
int x = 0, y = 0;
c o x = y+1; // y = y+1; oc;
```

Q: Does either arm satisfy the at-most-once property?
Q: What are values of \( x \) and \( y \)?
At-Most-Once Property

- A **critical reference** in an expression is a reference to a variable that is changed by another process.
- Assume a critical reference is to a simple variable (updating a value).
- An assignment \( x = e \) is **at-most-once** if either
  - \( e \) contains at MOST one critical reference and \( x \) is not read by another process.
  - \( e \) contains no critical references.

```
int x = 0, y = 0;
c o x = x+1;  // y = y+1; oc;
```

```
int x = 0, y = 0;
c o x = y+1;  // y = y+1; oc;
```

No. Both arms have an “e” that is a critical reference, AND the “x” portion of each arm is ready by another process.

```
int x = 0, y = 0;
c o x = y+1;  // y = x+1; oc;
```

Q: Does either arm satisfy the at-most-once property?

Q: What are values of \( x \) and \( y \)?
If an expression or assignment does not satisfy the at-most-once property, we need to have it executed atomically. This does not mean that a program that has arms that satisfy the at-most-once property will have the same “results” for all histories.

Thus we need to synchronize two or more processes if at least one of them is NOT at-most-once.
Await

If an expression or assignment does not satisfy the at-most-once property, we need to have it executed atomically. This does not mean that a program that has arms that satisfy the at-most-once property will have the same “results” for all histories.

Thus we need to synchronize two or more processes if at least one of them is NOT at-most-once.

**Atomic** actions are specified by use of angle brackets, < and >

Atomic actions in pseudocode can be enforced via use of await statements, which can be implemented using while loops.
Wait

- `< await (B) S; >`
- `B` specifies a Delay condition
- `S` is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
Await

• \(< \text{await} \ (B) \ S; \ >\)
• \(B\) specifies a Delay condition
• \(S\) is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
• \(<>\) specify atomic action
• Therefore \(B\) is guaranteed to be true when execution of \(S\) begins
• No internal state of \(S\) is visible to other processes

\(< \text{await} \ (s > 0) \ s = s - 1 \ >\)

Q: What does the above await statement indicate?
Await

• < await (B) S; >
• B specifies a Delay condition
• S is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
• <> specify atomic action
• Therefore B is guaranteed to be true when execution of S begins
• No internal state of S is visible to other processes

• < await (s > 0) s = s − 1 >

Decrement s when s is greater than 0
The value of s is guaranteed to be positive before x is decremented
Await

- `< await (B) S; >`
- B specifies a Delay condition
- S is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
- `<>` specify atomic action
- Therefore B is guaranteed to be true when execution of S begins
- No internal state of S is visible to other processes

```plaintext
< await (s > 0) s = s - 1 >
```
Wait for
Await

• \(< \text{await } (B) \; S; \; >\)
• \(B\) specifies a Delay condition
• \(S\) is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
• \(<\>\) specify atomic action
• Therefore \(B\) is guaranteed to be true when execution of \(S\) begins
• No internal state of \(S\) is visible to other processes

\(< \text{await } (s > 0) \; s = s - 1 \; >\)

Wait for This condition
Await

- `< await (B) S; >`
- `B` specifies a Delay condition
- `S` is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
- `<>` specify atomic action
- Therefore `B` is guaranteed to be true when execution of `S` begins
- No internal state of `S` is visible to other processes

`: await (s > 0) s = s - 1`

Wait for This condition And only then decrement `s`

Q: Does `await` imply mutex, conditional synchronization, neither, or both?
Await

- \(< \text{await (B) } S; >\)
- B specifies a Delay condition
- S is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
- <> specify atomic action
- Therefore B is guaranteed to be true when execution of S begins
- No internal state of S is visible to other processes

\(< \text{await (s > 0) } s = s - 1 >\)

Await specifies mutual exclusion AND conditional synchronization

Q: How would you specify ONLY mutex?
Await

- `< await (B) S; >`
- B specifies a Delay condition
- S is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
- `<>` specify atomic action
- Therefore B is guaranteed to be true when execution of S begins
- No internal state of S is visible to other processes

```
< await (s > 0) s = s - 1 >
```

await specifies mutual exclusion AND conditional synchronization

**Q:** How would you specify ONLY mutex? `<S;>`
Await

- `< await (B) S; >`
- B specifies a Delay condition
- S is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
- `<>` specify atomic action
- Therefore B is guaranteed to be true when execution of S begins
- No internal state of S is visible to other processes

`< await (s > 0) s = s - 1 >`

**await** specifies mutual exclusion AND conditional synchronization

Q: How would you specify ONLY condition synchronization?
Await

- `< await (B) S; >`
- B specifies a Delay condition
- S is a sequence of statements Guaranteed to terminate (a sequence of assignment operations)
- `< >` specify atomic action
- Therefore B is guaranteed to be true when execution of S begins
- No internal state of S is visible to other processes

`< await (s > 0) s = s - 1 >`

Await specifies mutual exclusion AND conditional synchronization

Q: How would you specify ONLY condition synchronization? `<await (B);>`
Q: Task: Write “code” for delaying the execution process until \( \texttt{count} > 0 \)

Q: How do “stall” a program in python, C++, Java, etc.?

(on the board suggestions)
Q: Task: Write “code” for delaying the execution process until \(\text{count} > 0\)

\[\texttt{<Await (count > 0);>}\]

Q: How do we implement in code an await statement?
Q: Task: Write “code” for delaying the execution process until \( \text{count} > 0 \)

\[
<\text{await} \ (\text{count} > 0);> 
\]

Q: How do we implement in code an await statement?

\[
\text{while (not B);} 
\]
Await

Q: Task: Write “code” for delaying the execution process until `count > 0`

```c
<await (count > 0);>
```

Q: How do we implement in code an `await` statement?

```c
while (not B);
```

And in this case `<await (count > 0);>` is
Await

Q: Task: Write “code” for delaying the execution process until count > 0

```csharp
<await (count > 0);>
```

Q: How do we implement in code an await statement?

```csharp
while (not B);
```

And in this case `<await (count > 0);>` is

```csharp
while (not count > 0);
```
Assume the following two code statements

\[
\begin{align*}
S1 & : \ x = x + y; \\
S2 & : \ y = x * y;
\end{align*}
\]

And that the variables \(x\) and \(y\) have initial values \(x=3\) and \(y=4\). Then, what is the final state (values for \(x\) and \(y\)) for the following program \(P3\) (assume \(S1\) and \(S2\) are executed atomically):

\[
P3 : \ co \ await \ (y < x) \ S1; \ // \ S2; \ oc
\]

In class exercise
Up Next

Formal Logic
Logic Axioms
Conditional Proofs
Indirect Proofs