Announcements

HW 1

- Due tonight, 6 July, 11:59pm
- Late submissions are NOT accepted

Midterm Exam

- Will be made available on Thursday
- Will be due Monday, 11 July, at 11:59pm
- Today I’ll go over the sample midterm exam posted on the course website
From last time

So that we can formally reason about concurrency programs, we started with a few definitions ... Our goal is to formally understand how/if/when a concurrent program “works”

```plaintext
c0 \[i = 0 \text{ to } n-1, \ j = 0 \text{ to } n-1\] \{ \# all rows and  
c[i,j] = 0.0; \# all columns  
for \[k = 0 \text{ to } n-1\]  
c[i,j] = c[i,j] + a[i,k]*b[k,j];  
\}
```
From last time

So that we can formally reason about concurrency programs, we started with a few definitions ...

Our goal is to formally understand how/if/when a concurrent program “works”

• State :
• Atomic Action :
• History, interleaving, trace :
• Critical Section :
• Property :
  • Safety Property :
  • Liveness Property :

• Partial Correctness :
• Termination :
• Total Correctness :

```c
for (i = 0 to n-1, j = 0 to n-1) { # all rows and columns
    c[i,j] = 0.0;
    for (k = 0 to n-1)
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```
From last time

So that we can formally reason about concurrency programs, we started with a few definitions ...

Our goal is to formally understand how/if/when a concurrent program “works”

- **State**: the values of all the variables at a point in time
- **Atomic Action**: Action which indivisibly examines or changes a state
- **History, interleaving, trace**: Particular sequence of atomic actions
- **Critical Section**: Section that cannot be interleaved with other actions
- **Property**: Something that is true of EVERY possible history
  - **Safety Property**: Program never enters a bad state (mutual exclusion)
  - **Liveness Property**: Program eventually enters good state (enter critical section without deadlock)
- **Partial Correctness**: Final state is correct, assuming the program terminates
- **Termination**: Every loop and procedure call terminates
- **Total Correctness**: Partially correct and termination

```cpp
for [i = 0 to n-1, j = 0 to n-1] { # all rows and
  c[i,j] = 0.0; # all columns
  for [k = 0 to n-1]
    a[i,j] = a[i,j] + a[i,k]*b[k,j];
}
```
From last time

So that we can formally reason about concurrency programs, we started with a few definitions ...
Our goal is to formally understand how/if/when a concurrent program “works”

- **State**: the values of all the variables at a point in time
- **Atomic Action**: Action which indivisibly examines or changes a state
- **History, interleaving, trace**: Particular sequence of atomic actions
- **Critical Section**: Section that cannot be interleaved with other actions
- **Property**: Something that is true of EVERY possible history
  - **Safety Property**: Program never enters a bad state (mutual exclusion)
  - **Liveness Property**: Program eventually enters good state (enter critical section without deadlock)
- **Partial Correctness**: Final state is correct, assuming the program terminates
- **Termination**: Every loop and procedure call terminates
- **Total Correctness**: Partially correct and termination

```c
for (i = 0 to n-1, j = 0 to n-1) {  // all rows and all columns
    c[i,j] = 0.0;
    for (k = 0 to n-1)
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
}
```
Q: How do we demonstrate properties of a concurrent program?
Q: How do we demonstrate properties of a concurrent program?

Test and debug

\[ m \] \text{ threads executing } n \text{ atomic actions} \\
\text{possible histories}

\begin{array}{ccc}
3 & 1 & 6 \\
\end{array}

* Each of the \( m \) threads has \( n \) actions (instructions)
Q: How do we demonstrate properties of a concurrent program?

Test and debug

$m$ threads executing $n$ atomic actions

<table>
<thead>
<tr>
<th>$m$</th>
<th>$n$</th>
<th>possible histories</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>?</td>
</tr>
</tbody>
</table>

For $m=2$ and $n=2$, how many histories are possible, considering each of the atomic actions in threads A are independent from the atomic actions in B?
From last time ...

**Q: How do we demonstrate properties of a concurrent program?**

Test and debug

\[
\begin{array}{ccc}
\text{\(m\)} & \text{\(n\)} & \text{possible histories} \\
3 & 1 & 6 \\
2 & 2 & ? \\
\end{array}
\]

For \(m=2\) and \(n=2\), how many histories are possible, considering each of the atomic actions in threads A are independent from the atomic actions in B?

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>A2</th>
<th>B1</th>
<th>B2</th>
</tr>
</thead>
</table>
Q: How do we demonstrate properties of a concurrent program?

Test and debug

\[
m \text{ threads executing } n \\
\text{atomic actions}
\]

<table>
<thead>
<tr>
<th>(m)</th>
<th>(n)</th>
<th>possible histories</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>?</td>
</tr>
</tbody>
</table>

For \(m=2\) and \(n=2\), how many histories are possible, considering each of the atomic actions in threads A are independent from the atomic actions in B?

A1

A2

B1

B2

Then, identify those that meet the criteria:

A1 A2 B1 B2
A2 A1 B1 B2
B1 A2 A1 B2
B2 A2 B1 A1

A1 A2 B2 B1
A2 A1 B2 B1
B1 A2 B2 A1
B2 A2 A1 B1

A1 B1 B2 A2
A2 B1 B2 A1
B1 A1 B2 A2
B2 B1 A1 A2

A1 B1 A2 B2
A2 B1 A1 B2
B1 A1 A2 B2
B2 B1 A2 A1

A1 B2 A2 B1
A2 B2 A1 B1
B1 B2 A2 A1
B2 A1 A2 B1

A1 B2 B1 A2
A2 B2 B1 A1
B1 B2 A1 A2
B2 A1 B1 A2
From last time ...

**Q: How do we demonstrate properties of a concurrent program?**

Test and debug

$m$ threads executing $n$ atomic actions

<table>
<thead>
<tr>
<th>$m$</th>
<th>$n$</th>
<th>possible histories</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2520</td>
</tr>
</tbody>
</table>

**Q: What is the formula for calculating the count of number of histories?**
Q: How do we demonstrate properties of a concurrent program?

Test and debug

$m$ threads executing $n$ atomic actions

<table>
<thead>
<tr>
<th>$m$</th>
<th>$n$</th>
<th>possible histories</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2520</td>
</tr>
</tbody>
</table>

Q: What is the formula for calculating the count of number of histories?

\[
\frac{(mn)!}{(n!)^m}
\]
Q: How do we demonstrate properties of a concurrent program?

Test and debug

$m$ threads executing $n$ atomic actions

<table>
<thead>
<tr>
<th>$m$</th>
<th>$n$</th>
<th>possible histories</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2520</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>?</td>
</tr>
</tbody>
</table>

Q: What is the formula for calculating the count of number of histories?

$$\frac{(mn)!}{(n!)^m}$$
Q: How do we demonstrate properties of a concurrent program?

Test and debug

$m$ threads executing $n$ atomic actions

<table>
<thead>
<tr>
<th>$m$</th>
<th>$n$</th>
<th>possible histories</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2520</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>472518347558400</td>
</tr>
</tbody>
</table>

Q: What is the formula for calculating the count of number of histories?

\[
\frac{(mn)!}{(n!)^m}
\]
Today

Arms
await
Sample midterm review
Motivation

The state of a concurrent program: ____________

We begin with a few observations.
Motivation

We begin with a few observations:

The state of a concurrent program consists of the values of the program’s variables at some point in time.

Each execution of a concurrent program produces a history, and the number of possible histories is (can be) enormous.
Motivation

We begin with a few observations.

The state of a concurrent program consists of the values of the program’s variables at some point in time.

Each execution of a concurrent program produces a history, and the number of possible histories is (can be) enormous.

Goal: Restrict or constrain the possible histories so that only those histories that are desirable are ever witnessed.

Mutual exclusion and synchronization.
Motivation ...

```c
string line;
read a line of input from stdin into line;
while (!EOF) {   # EOF is end of file
    look for pattern in line;
    if (pattern is in line)
        write line;
    read next line of input;
}
```

Q: What does the above “code” accomplish?
Looking for patterns

```c
string line;
read a line of input from stdin into line;
while (!EOF) {    # EOF is end of file
    look for pattern in line;
    if (pattern is in line)
        write line;
    read next line of input;
}
```

Q: What does the above “code” accomplish?

Similar to grep pattern filename
Looking for patterns

string line;
read a line of input from stdin into line;
while (!EOF) {
    # EOF is end of file
    look for pattern in line;
    if (pattern is in line)
        write line;
    read next line of input;
}

Q: What does the above “code” accomplish?
Q: Can we parallelize it? If so, how?
Looking for patterns

A proposed solution to parallelization ... be sure you “understand” the special symbols being used

```c
string line;
read a line of input from stdin into line;
while (!EOF) {
    co look for pattern in line;
    if (pattern is in line)
        write line;
    // read next line of input into line;
    oc;
}
```
Looking for patterns

We've seen these before; they specify a region of code that should be executed concurrently.

```cpp
string line;
read a line of input from stdin into line;
while (!EOF) {
    co look for pattern in line;
    if (pattern is in line)
        write line;
    // read next line of input into line;
    oc;
}
```

This is the code enclosed by co and oc, which is being parallelized.
Looking for patterns

This specifies an arm of the program ... think of a program with multiple arms as being able to do several things at once (as many arms as there are available)

There are “two” arms, each of which is executed concurrently

Notice that the “first” arm is a sequence of statements, but the second arm is a single statement
Looking for patterns

```c
string line;
read a line of input from stdin into line;
while (!EOF) {
    if (pattern is in line) {
        write line;
    }
    // read next line of input into line;
    oc;
}
```

Q: Are the two arms (or processes) independent? Why or why not? What are their read and write sets? Are they disjoint?
Looking for patterns

No, because they are sharing a variable, `line`, and the two processes read from and write to it potentially at the same time ... and maybe at different speeds.

Q: If this is not a good approach, how do we fix it?
Looking for patterns

Q: Is this a good solution?

```c++
string line1, line2;
read a line of input from stdin into line1;
while (!EOF) {
    if (pattern is in line1)
        write line1;
    // read next line of input into line2;
    oc;
}
```

Task: Explain what this code accomplishes
Looking for patterns

Q: Is this a good solution?

```c
string line1, line2;
read a line of input from stdin into line1;
while (!EOF) {
    co look for pattern in line1;
    if (pattern is in line1)
        write line1;
    // read next line of input into line2;
    oc;
}
```

The two processes are working on different “lines” stored in variables line1 and line2

Q: What does this code accomplish?
Looking for patterns

Q: Is this a good solution?

```cpp
string line1, line2;
read a line of input from stdin into line1;
while (!EOF) {
    // look for pattern in line1;
    if (pattern is in line1)
        write line1;
    // read next line of input into line2;
    oc;
}
```

The first process repeatedly looks at line1, and the second process repeatedly reads into (writes to) line 2 from standard input. We say that these two processes are disjoint.

Q: How do we fix this program? (in class exercise)
Looking for patterns

```c
string line1, line2;
read a line of input from stdin into line1;
while (! EOF) {
    co look for pattern in line1;
    if (pattern is in line1)
        write line1;
    // read next line of input into line2;
    oc;
    line1 = line2;
}
```

Q: What have we changed, and how does this make the concurrent program “work”?
Looking for patterns

```c
string line1, line2;
read a line of input from stdin into line1;
while (! EOF) {
    // look for pattern in line1;
    if (pattern is in line1)
        write line1;
    // read next line of input into line2;
    line2 = line2;
}
```

At the end of each loop iteration, and after each arm has finished, copy the contents of line2 into line1. Be sure that you can “see” what this is doing.
Looking for patterns

```cpp
string line1, line2;
read a line of input from `stdin` into `line1`;
while (! EOF) {
    co look for pattern in `line1`;
    if (pattern is in `line1`) write `line1`;
    // read next line of input into `line2`;
    oc;
    line1 = line2;
}
```

At the end of each loop iteration, and after each arm has finished, copy the contents of `line2` into `line1`. Be sure that you can “see” what this is doing.
Looking for patterns

```cpp
string line1, line2;
read a line of input from stdin into line1;
while (! EOF) {
    look for pattern in line1;
    if (pattern is in line1)
        write line1;
    // read next line of input into line2;
    line1 = line2;
}
```

At the end of each loop iteration, and after each arm has finished, copy the contents of line2 into line1. Be sure that you can “see” what this is doing.

Reading from stdin, and saving into line2

Looking for the pattern

CPU_0
Arm 1

CPU_1
Arm 2
Looking for patterns

```cpp
string line1, line2;
read a line of input from stdin into line1;
while (! EOF) {
  co look for pattern in line1;
  if (pattern is in line1)
    write line1;
  // read next line of input into line2;
  oc;
  line1 = line2;
}
```

Done. Right? Or are we?

Q: Why is this solution inefficient?
Looking for patterns

```cpp
string line1, line2;
read a line of input from stdin into linel;
while (! EOF) {
    co look for pattern in linel;
    if (pattern is in linel)
        write linel;
    // read next line of input into line2;
oc;
    line1 = line2;
}
```

At each iteration of the while loop, how many concurrent processes are created, run to completion, and are destroyed?
Looking for patterns

At each iteration of the while loop, how many concurrent processes are created, run to completion, and are destroyed?

This approach would create and destroy (when done) MANY! processes especially if stdin is reading from a HUGE text file)

```c
string line1, line2;
read a line of input from stdin into line1;
while (! EOF) {
    co look for pattern in line1;
    if (pattern is in line1)
        write line1;
    // read next line of input into line2;
    oc;
    line1 = line2;
}
```
Looking for patterns

Looking for patterns

At each iteration of the while loop, how many concurrent processes are created, run to completion, and are destroyed?

Q: What is the “solution”?

This approach would create and destroy (when done) MANY! processes especially if stdin is reading from a HUGE text file)
Looking for patterns

```c
string buffer;  // contains one line of input
define 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;
  }
```

Q: How do the processes communicate?
Looking for patterns

```c
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;
    }
c0;
```

Pros

Cons

Looking for patterns

Pros

- Processes are created only once

Cons

- Code is more complex
- There is the sharing of a variable (buffer) that must be synchronized (using semaphores and enforcing mutual exclusion)

```cpp
string buffer;  // contains one line of input
bool done = false;  // used to signal termination
co  // process 1: find patterns
    string linel;
    while (true) {
        wait for buffer to be full or done to be true;
        if (done) break;
        linel = buffer;
        signal that buffer is empty;
        look for pattern in linel;
        if (pattern is in linel)
            write linel;
    }
    // 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;
    }
```
Looking for patterns

We need to accomplish these tasks ... how? ...

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

// # 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;
}
```

Looking for patterns

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

We need to accomplish these tasks ... how? ...

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

In the previous example, synchronization was done by sharing a single variable (buffer). But what if two or more arms (processes) must coordinate among themselves?

Task: Give an example of a problem that can be decomposed into many smaller problems, and where each of the “solutions” to the small problems need to communicate with each other.
In the previous example, synchronization was done by sharing a single variable (buffer). But what if two or more arms (processes) must coordinate among themselves?

**Task:** Give an example of a problem that can be decomposed into many smaller problems, and where each of the “solutions” to the small problems need to communicate with each other.

Q: If you have 4 processors, how do you find the largest entry in this array?
In the previous example, synchronization was done by sharing a single variable (buffer). But what if two or more arms (processes) must coordinate.

Task: Give an example of a problem that can be decomposed into many smaller problems, and where each of the “solutions” to the small problems need to communicate with each other.
Synchronization

Q: What is the “sequential” solution to this problem?

In class exercise
Synchronization

```c
int m = 0;
for [i = 0 to n-1]
    if (a[i] > m)
        m = a[i];
Q: Does this employ parallelization?
Q: Is this “fast”?
```
Because this solution is “slow”, let’s parallelize it ...

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

Q: What is a possible divide-and-conquer approach? (can you break the problem space into two, three, or four regions?)

In class exercise
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)
        m = a[i];
}

Explain what this code does.

Does this solution work? Why or why not?

On-the board explanation
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 \( m \).

Q: How do we fix this code so that the solution is efficient and correct?

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

On the board explanation

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

Q: How do we fix this code so that the solution is efficient and correct?
A possible solution. The `< >` brackets specify that EACH concurrent arm should inspect and write to `m` as a single action.

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

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.

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.

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

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

```c
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.

Q: What is the optimal solution?

```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];
}
```
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) 
        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

```java
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];
}
```

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 increasing likely that a process will not have to do a second check, and after the completion of the program, m is the maximum value.

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

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.

```c
int y = 0, z = 0;
x = y+z; y = 1; 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`?
**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?  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 $x$? 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;
c x = y+z; // y = 1; z = 2; oc;
```

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

- i1: fetch y
- i2: fetch z
- i3: compute \( x = y + z \)
- i4: write back \( x \)

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

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;
c x = y + z; // y = 1; z = 2; oc;
```

Keep in mind that \(x = y + z\) is 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 \(i_1 - i_6\)

**Q:** How might \(x = 2\) be the final state?

What is an instruction history such that \(x\) has a final value of 2?

0, 1, **2** or 3
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 \( P1 \) (\textbf{assume \( S1 \) and \( S2 \) are executed atomically}):

\[ P1 : \text{co } S1; // S2; \text{ oc} \]

In class exercise
How do we **synchronize** two or more processes if we cannot guarantee that those actions are atomic?
Await

How do we synchronize two or more processes if we cannot guarantee that those actions are atomic?

**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.
Await

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

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

- `< 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
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
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 >
```

This 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 >`

This **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 >`

This *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 >
```

This await specifies mutual exclusion AND conditional synchronization

Q: How would you specify ONLY condition synchronization? `<await (B);>`
Await

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

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

(on the board suggestions)
Await

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?
Await

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

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

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

while (not B);
Await

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?

while (not B);

And in this case \(<\text{await} \ (\text{count} > 0);>\) is
Await

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

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

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

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

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

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

S1 : x = x + y;
S2 : y = x * 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 P2 (assume S1 and S2 are executed atomically):

P2 : co await (y < x) S1; // S2; oc

In class exercise
Solutions to sample midterm exam

Provided in-class
Lab (Thursday)