CSCI 322
Principles of Concurrent Programming

Filip Jagodzinski
From last time

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

Our goal is to understand and “hopefully” automate reasoning about 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
co [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 understand and “hopefully” automate reasoning about 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
co [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]
    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 understand and “hopefully” automate reasoning about 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
co [i = 0 to n-1, j = 0 to n-1] { # all rows and
    c[i,j] = 0.0;
    for [k = 0 to n-1]
        c[i,j] = c[i,j] + a[i,k]*b[k,j];
} 
```
Today

Synchronization
Atomic actions
Awaits
Motivation

We begin with a few observations

The state of a concurrent program: ____________
Motivation

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 enormous.

We begin with a few observations.
Motivation

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 enormous.

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

Mutual exclusion and synchronization.

Ultimate goal: Prove that “bad” states cannot happen.
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?
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.

```
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

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

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).

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

```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

```
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;
}
```

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

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

No, because they are sharing a variable, `line`, and the two processes read from and write to it potentially at the same time or 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) {
    co look for pattern in line1;
    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: but does this code (the arms when executing concurrently) do anything?
Looking for patterns

Q: Is this a good solution?

No. The first process just 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: What’s the solution? (in class exercise)
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;
}
```

Q: What have we changed, and how does this make the concurrent program “work”?
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) {
    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.

Reading from stdin, and saving into line2

Looking for the pattern

CPU_0
Arm 1

CPU_1
Arm 2
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;
}
```

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

```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;
}
```

Done. Right? Or are we?

Q: Why is this solution inefficient?
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;
}
```

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?

Q: What is the “solution”? Instead of co statements inside a while loop …
Looking for patterns

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;
    }
}
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;
}
oc;
```

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)
Looking for patterns

```
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;
    }
```

Task: Draw the process, global variable diagram, and explain how these two processes efficiently search for a pattern in a file.

On the board explanation
Synchronization

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

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.

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

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

```java
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”?
Synchronization

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

```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 ...

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

In class exercise
Synchronization

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

```c
int m = 0;
for [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
Q: What is the “sequential” solution to this problem?

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

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 \( m \)

On the board explanation

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

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

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

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.
Synchronization

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

```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.

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

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

```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.

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

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

```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.

```
int m = 0;
cor [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

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

```java
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.

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

Each a[i] is still examined one at a time due to synchronization, JUST like in the sequential version.

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

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

```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.

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

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

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

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.

```c
int m = 0;
c0 [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}
```

On the board explanation

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

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

Task: Explain this code

```java
int m = 0;
for [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}
```
Q: What is the “sequential” solution to this problem?

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

Each process first performs a comparison
Q: What is the “sequential” solution to this problem?

```c
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.
Synchronization

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

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.

Q: How is such behavior (atomicity) implemented in software?
Up Next ...

Await Statement
Formal Logic
Proof Strategies