CS 476 – Programming Language Design

William Mansky
Questions?
Concurrency

• In a concurrent (parallel) program, multiple sections of code execute at the same time (threads, processes, etc.)

• But what does “at the same time” mean? That’s a question about semantics!

• One common decision: “one thread executes for a while, then stops, then the other thread executes” is an acceptable implementation

• More generally, any possible *interleaving* of the threads is a possible behavior of a concurrent program
Concurrency

• More generally, any possible *interleaving* of the threads is a possible behavior of a concurrent program.

```java
while(x < 100){
    x = x + 1;
}
while(y < 200){
    y = y + 2;
}
```

{x = 0, y = 0}, {x = 1, y = 0}, {x = 2, y = 0}, {x = 2, y = 2}, {x = 2, y = 4}, ...
Concurrency

• More generally, any possible *interleaving* of the threads is a possible behavior of a concurrent program.

```plaintext
while(x < 100){
    x = x + 1;
}

while(y < 200){
    y = y + 2;
}
```

{x = 0, y = 0}, {x = 1, y = 0}, {x = 2, y = 0}, {x = 3, y = 0}, {x = 4, y = 0}, ...
Concurrency: Syntax

\[ E ::= \langle \# \rangle \mid \langle \text{ident} \rangle \mid E + E \mid \ldots \]

\[ C ::= \langle \text{ident} \rangle = E \mid C; C \mid \text{skip} \mid \text{while } E \text{ do } C \mid \ldots \mid (C || C) \]

Examples:

\[ x := 0; y := 0; (x = x + 1 || y = y + 2); z = x + y \]

while (x < 100) do (x = x + 1 || y = y + 2)
Concurrency: Typing and Semantics

\[
\begin{align*}
\Gamma \vdash c_1 : \text{ok} & \quad \Gamma \vdash c_2 : \text{ok} \\
\Gamma \vdash c_1 \parallel c_2 : \text{ok}
\end{align*}
\]

\[
\begin{align*}
(c_1, \sigma) \rightarrow (c'_1, \sigma') & \quad (\text{skip} \parallel \text{skip}, \sigma) \rightarrow (\text{skip}, \sigma') \\
(c_1 \parallel c_2, \sigma) \rightarrow (c'_1 \parallel c_2, \sigma') \\
(c_2, \sigma) \rightarrow (c'_2, \sigma') & \quad (c_1 \parallel c_2, \sigma) \rightarrow (c_1 \parallel c'_2, \sigma')
\end{align*}
\]
Questions?

Top
Concurrency: Shared Variables

\[
\begin{align*}
\text{while}(x < 100) &\text{ do } \quad \text{while}(y < 200) &\text{ do } \\
 & x = x + 1; \quad & y = y + 2; \\
\end{align*}
\]

\{x = 0, y = 0\}, \{x = 1, y = 0\}, \{x = 2, y = 0\}, \{x = 2, y = 2\}, \{x = 2, y = 4\}, \ldots
Concurrency: Shared Variables

while(x < 100){
    x = x + 1;
}
{x = 0}, {x = 1}, {x = 3}, ...

while(x < 200){
    x = x + 2;
}

Concurrency: Shared Variables

\[ x = 0; \]

\[ x = x + 1; \]

Exercise: What are the possible values of \( x \) after this program runs?
Concurrency: Shared Variables

```plaintext
x = 0;

x = x + 1;  
|   
x = x + 1;

{x = 2}
```
Concurrency: Shared Variables

\[
x = 0;
\]

\[
y = x;^1 \quad \| \quad z = x;^2
\]

\[
x = y + 1;^3 \quad \| \quad x = z + 1;^4
\]

\[
\{x = {?}\}
\]
Concurrency: Shared Variables

\[ x = 0; \]

\[ \begin{align*}
  y &= x;^1 \\
  x &= y + 1;^3 \\
  z &= x;^2 \\
  x &= z + 1;^4
\end{align*} \]

\[ \{x = 1\} \quad \frac{(e, \sigma) \Downarrow v}{(x = e; , \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto v])} \]
Concurrency: Shared Variables

x = 0;
acquire(s); acquire(s);
x = x + 1; x = x + 1;
release(s); release(s);

{x = 2}
Concurrency: Locks

\[ E := \langle\#\rangle \mid \langle\text{ident}\rangle \mid E + E \mid \ldots \]

\[ C := \langle\text{ident}\rangle = E \mid C; C \mid \text{skip} \mid \text{while } E \text{ do } C \mid \ldots \mid (C \mid C) \]

\[ \mid \text{acquire}(\langle\text{ident}\rangle) \mid \text{release}(\langle\text{ident}\rangle) \]

\[
\sigma(x) = 0 \\
\frac{\text{(acquire}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 1])}{\sigma(x) = 1} \\
\text{release}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 0])
\]
Concurrency: Locks

{\(x = 0, \ s = 0\)}

\[
\begin{align*}
\text{acquire}(s); & \quad \text{acquire}(s);
\text{x = x + 1;} & \quad \text{x = x + 1;}
\text{release}(s); & \quad \text{release}(s);
\end{align*}
\]

\[
\begin{align*}
\sigma(x) &= 0 \\
(\text{acquire}(x),\sigma) &\rightarrow (\text{skip},\sigma[x \mapsto 1]) \\
\sigma(x) &= 1 \\
(\text{release}(x),\sigma) &\rightarrow (\text{skip},\sigma[x \mapsto 0])
\end{align*}
\]
Concurrency: Locks

\{x = 0, s = 1\}

\[\text{acquire}(s); \quad x = x + 1; \quad \text{release}(s);\]

\[
\begin{align*}
\sigma(x) &= 0 \\
(acquire(x), \sigma) &\rightarrow (\text{skip}, \sigma [x \mapsto 1])
\end{align*}
\]

\[
\begin{align*}
\sigma(x) &= 1 \\
(release(x), \sigma) &\rightarrow (\text{skip}, \sigma [x \mapsto 0])
\end{align*}
\]
Concurrency: Locks

\{x = 1, s = 1\}

\(\text{acquire}(s); \quad x = x + 1; \quad \text{release}(s);\)

\(\sigma(x) = 0\)

\((\text{acquire}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 1])\)

\(\sigma(x) = 1\)

\((\text{release}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 0])\)
Concurrency: Locks

{x = 1, s = 0}

acquire(s);

x = x + 1;

release(s);

\[
\begin{align*}
\sigma(x) &= 0 \\
(acquire(x), \sigma) &\rightarrow (skip, \sigma[x \mapsto 1])
\end{align*}
\]

\[
\begin{align*}
\sigma(x) &= 1 \\
(release(x), \sigma) &\rightarrow (skip, \sigma[x \mapsto 0])
\end{align*}
\]
Questions?
Concurrence: Fork-Join

• Most real languages don’t use the || operator (except Chapel)
  — Beginning and ending parallelism has a cost
  — These languages weren’t designed for concurrency!

• More common model: explicitly create a thread, which runs a function and sends a signal when it terminates
Concurrency: Fork-Join

\[
\begin{align*}
x &= x + 1; \\
| & | \\
x &= x + 1;
\end{align*}
\]

```c
int x = 0;

int main(){
    tid t = fork(thread_fun);
    x = x + 1;
    x = x + 1;
    join(t);
}

void thread_fun(){
    x = x + 1;
    x = x + 1;
}
```
Concurrent Syntax: Fork-Join Syntax

\[ \begin{align*}
E & ::= <#> \mid <\text{ident}> \mid E + E \mid \ldots \\
C & ::= \ldots \mid <\text{ident}> = \text{fork}(<\text{ident}>) \mid \text{join}(E) \\
T & ::= \ldots \mid \text{tid}
\end{align*} \]

\[
\frac{\Gamma(x) = \text{tid} \quad \Gamma(f) = \text{void}()}{\Gamma \vdash x = \text{fork}(f) : \text{ok}} \quad \frac{\Gamma \vdash e : \text{tid}}{\Gamma \vdash \text{join}(e) : \text{ok}}
\]
For single-threaded programs, a configuration is a tuple \((c, \rho, \sigma)\) of command, environment, and store.

Exercise: How would you model the state of a multithreaded program with separate variables but shared memory?
Concurrency: Fork-Join Semantics

• Each thread has its own environment, and a thread id, but all threads share the same store

• A configuration with $n$ threads has the form $\left( \left[ (t_1, c_1, ρ_1); \ldots; (t_n, c_n, ρ_n) \right], σ \right)$

• At each step, we pick one thread to execute:

$$\left( c_i, ρ_i, σ \right) \rightarrow \left( c'_i, ρ'_i, σ' \right)$$

$$\left( \left[ (t_1, c_1, ρ_1 ); \ldots; (t_i, c_i, ρ_i ); \ldots; (t_n, c_n, ρ_n ) \right], σ \right) \rightarrow$$

$$\left( \left[ (t_1, c_1, ρ_1 ); \ldots; (t_i, c'_i, ρ'_i ); \ldots; (t_n, c_n, ρ_n ) \right], σ' \right)$$
Concurrency: Fork-Join Semantics

\[
\sigma(f) = (\text{void } f()\{c\}) \quad t_2 \text{ fresh}
\]

\[
\frac{}{([[t_1, x = \text{fork}(f), \rho); \ldots], \sigma) \rightarrow
([[t_1, \text{skip}, \rho[x \mapsto t_2]); (t_2, c, \{\}); \ldots], \sigma)}
\]

\[
\text{(e, } \rho, \sigma) \downarrow t_2
\]

\[
\frac{}{([[t_1, \text{join}(e), \rho); (t_2, \text{skip}, \rho_2); \ldots], \sigma) \rightarrow
([[t_1, \text{skip}, \rho); \ldots], \sigma)}
\]
Concurrence: Fork-Join

```c
int x, y = 0;

int main(){
    void thread_fun(){
        tid t = fork(thread_fun); (0, ρ₀)
        y = 2;
        x = 1;
    }
    join(t);
}
```

\{x \mapsto 0, y \mapsto 0\}
Concurrency: Fork-Join

```c
int x, y = 0;

int main(){
    tid t = fork(thread_fun);
    x = 1; (0, \rho_0[t \mapsto 1])
    join(t);
}

void thread_fun(){
    y = 2; (1, \rho_0)
}

{x \mapsto 0, y \mapsto 0}
```
Concurrency: Fork-Join

```c
int x, y = 0;

int main(){
    tid t = fork(thread_fun);
    x = 1;
    join(t);
}
```

```c
void thread_fun(){
    y = 2; (1, ρ₀)
}
```

\{x \mapsto 1, y \mapsto 0\}
Concurrency: Fork-Join

```c
int x, y = 0;

int main()
{
    tid t = fork(thread_fun);
    x = 1; // (0, \rho_0[t \mapsto 1])
    join(t);
}

void thread_fun()
{
    y = 2;
    } // (1, \rho_0)
```

{x \mapsto 0, y \mapsto 2}
Concurrency: Fork-Join

```c
int x, y = 0;

int main(){
    void thread_fun(){
        tid t = fork(thread_fun);
        y = 2;
        join(t);
    }

    x = 1;
    join(t);
}
```

{x ↦ 1, y ↦ 2}
Concurrency: Fork-Join

```c
int x, y = 0;

int main(){
    void thread_fun(){
        tid t = fork(thread_fun);
        y = 2;
    }

    x = 1;
    join(t);
}
```

(0, ρ₀[t ↦ 1])

{x ↦ 1, y ↦ 2}
Concurrent: Relaxed Memory

\[
x = 0;
\]

\[
y = x; \\
z = x;
\]

\[
x = y + 1; \\
x = z + 1;
\]

\{x = ?\}
Concurrency: Relaxed Memory

\[ x = 0; \]

\[ y = x; \]
\[ z = x; \]
\[ x = y + 1; \]
\[ x = z + 1; \]
\[ \{ x = 1 \} \]
Concurrency: Relaxed Memory

\[
x = 0;
\]

\[
y = x; \\
x = y + 1; \\
x = z + 1; \\
\{x = 2\}
\]

\[
z = x; \\
x = z + 1; 
\]
Concurrency: Relaxed Memory

\[ x = 0; \quad y = 0; \]

\[ \begin{align*}
  x &= 1; \\
  a &= y; \\
  y &= 1; \\
  b &= x;
\end{align*} \]

\[ \{x = 1, \quad y = 1, \quad a = 0, \quad b = 1\} \]

Can we end up with \( a = 0 \) and \( b = 0 \)?
Concurrency: Relaxed Memory

\[ x = 0; \ y = 0; \]

\[ x = 1; \quad 1 \]
\[ y = 1; \quad 2 \]
\[ a = y; \quad 3 \]
\[ b = x; \quad 4 \]

\{x = 1, y = 1, a = 1, b = 1\}

Can we end up with \( a = 0 \) and \( b = 0 \)?
Concurrency: Relaxed Memory

x = 0; y = 0;

x = 1;  
\[ a = y; \]
\[ b = x; \]

\{x = 1, y = 1, a = 1, b = 0\}

Can we end up with a = 0 and b = 0?

Logically, no: one of the reads is going to be last, and so will happen after both writes
Concurrency: Relaxed Memory

Thread 1 (with local variables $\rho_1$) 

$\star x = 1$

Thread 2 (with local variables $\rho_2$) 

$a = \star x$

$\{x \mapsto 1\}$
Concurrency: Relaxed Memory

Thread 1 (with local variables $\rho_1$)

\[
\downarrow *x = 1
\]

Cache 1

\{x \mapsto 1\}

Thread 2 (with local variables $\rho_2$)

\[
\downarrow a = *x
\]

Cache 2

\{x \mapsto 0\}

Memory ($\sigma$)

\{x \mapsto 0\}
Concurrency: Relaxed Memory

Thread 1 (with local variables $\rho_1$)

Thread 2 (with local variables $\rho_2$)

Cache 1

$\{x \mapsto 1\}$

Memory ($\sigma$)

$\{x \mapsto 1\}$

Cache 2

$\{x \mapsto 0\}$
Concurrency: Relaxed Memory

Thread 1 (with local variables $\rho_1$)

Cache 1

{x $\mapsto$ 1}

Thread 2 (with local variables $\rho_2$)

Cache 2

{x $\mapsto$ 1}

Memory ($\sigma$)

{x $\mapsto$ 1}
Concurrency: Relaxed Memory

\*x = 0; \*y = 0;

\*x = 1; \*y = 1;

\text{a = \*y;} \quad \| \quad \text{b = \*x;}

Can we end up with a = 0 and b = 0?
Yes, if the writes propagate slowly to other threads!
Concurrenti: Relaxed Memory

```
*x = 0; *y = 0;

*x = 1;   *y = 1;
fence;   fence;
a = *y;   b = *x;
```

Can we end up with $a = 0$ and $b = 0$?
Yes, if the writes propagate slowly to other threads!
Concurrency: Relaxed Memory

\*x = 0; \*y = 0;

\*x = 1 (strict); \*y = 1 (relaxed);

a = \*y; \hspace{1cm} b = \*x;

Can we end up with \(a = 0\) and \(b = 0\)?

Yes, if the writes propagate slowly to other threads!
Questions?

Top
Concurrency: Summary

• In a concurrent (parallel) program, multiple sections of code execute at the same time (threads, processes, etc.)
  — We can declare sections to run in parallel, or use a fork-join model

• What does “at the same time” mean? That’s a question about semantics!
  — Sometimes (esp. when threads don’t directly interact), interleaving is good enough
  — We might want to require synchronization (like lock acquire/release) when threads share variables/memory
  — Relaxed memory models can produce behavior that doesn’t come from any interleaving!