Concurrency

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

• “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

```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 = 2, y = 2}, {x = 2, y = 4}, ...
```
HW8 Questions
CS 473: Compiler Design

• We’ve learned how to mathematically describe the behavior of different kinds of languages

• CS 473 asks: how do you implement a language design?
  — From program text to AST (lexing, parsing)
  — From AST to efficient internal representation
  — From internal representation to assembly code
  — Optimizations to make code run faster and in less space

• Website: https://www.cs.uic.edu/~mansky/teaching/cs473/sp20/
Concurrent

• 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}, ...
```
Concurrent 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

\[ \Gamma \vdash c_1 : \text{ok} \quad \Gamma \vdash c_2 : \text{ok} \]
\[ \Gamma \vdash c_1 \parallel c_2 : \text{ok} \]

\[ (c_1, \sigma) \rightarrow (c'_1, \sigma') \]
\[ (c_1 \parallel c_2, \sigma) \rightarrow (c'_1 \parallel c_2, \sigma') \]

\[ (c_2, \sigma) \rightarrow (c'_2, \sigma') \]
\[ (c_1 \parallel c_2, \sigma) \rightarrow (c_1 \parallel c'_2, \sigma') \]

\[ (\text{skip} \parallel \text{skip}, \sigma) \rightarrow (\text{skip}, \sigma') \]
Concurrency: Shared Variables

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

{x = 0}, {x = 1}, {x = 3}, ...

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

{x = 0}, {x = 1}, {x = 3}, ...
Concurrency: Shared Variables

\[
x = 0;
\]

\[
x = x + 1; \quad \|
\quad x = x + 1;
\]

\[
\{x = 2\}
\]
Concurrency: Shared Variables

x = 0;

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

\{x = ?\}
Concurrency: Shared Variables

\[
\begin{align*}
x &= 0; \\
y &= x;^1 & z &= x;^2 \\
x &= y + 1;^3 & x &= z + 1;^4 \\
\{x = 1\}
\end{align*}
\]

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

\[ x = 0; \]
\[ \text{acquire}(s); \]
\[ x = x + 1; \]
\[ \text{release}(s); \]
\[ \{ x = 2 \} \]

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

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

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

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

\[
\sigma(x) = 0 \\
(\text{acquire}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 1])
\]

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

\{x = 0, s = 0\}

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

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

\{ x = 0, s = 1 \}

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

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

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

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

\[\sigma(x) = 0 \quad \text{(acquire}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 1])\]

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

\[ x = x + 1; \quad \| \quad x = x + 1; \]

int \ x = 0;

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

void thread_fun(){
    x = x + 1;
}


Concurrency: Fork-Join Syntax

\[ E ::= \# | \text{id} | E + E | \ldots \]

\[ C ::= \ldots | \text{id} = \text{fork}(\text{id}); | \text{join}(E); \]

\[ T ::= \ldots | \text{tid} \]

\[
\frac{\Gamma(x) = \text{tid} \quad \Gamma(f) = \text{void}()} {\Gamma \vdash x = \text{fork}(f); : \text{ok}}
\]

\[
\frac{\Gamma \vdash e : \text{tid}} {\Gamma \vdash \text{join}(e); : \text{ok}}
\]
Concurrency: Fork-Join Semantics

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

• A configuration with \( n \) threads is of the form

\[
\left[ \left[ (i_1, c_1, \rho_1, k_1); \ldots; (i_n, c_n, \rho_n, k_n) \right], \sigma \right]
\]

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

\[
\frac{(c_j, \rho_j, k_j, \sigma) \rightarrow (c'_j, \rho'_j, k'_j, \sigma')}{\left[ \left[ (i_1, c_1, \rho_1, k_1); \ldots; (i_j, c_j, \rho_j, k_j); \ldots; (i_n, c_n, \rho_n, k_n) \right], \sigma \right] \rightarrow \left[ \left[ (i_1, c_1, \rho_1, k_1); \ldots; (i'_j, c'_j, \rho'_j, k'_j); \ldots; (i_n, c_n, \rho_n, k_n) \right], \sigma' \right]}
\]
Concurrency: Fork-Join Semantics

\[ \sigma(f) = (\cdot, c) \quad i' \text{ fresh} \]

\[
\frac{\sigma(f) = (\cdot, c) \quad i' \text{ fresh}}{\text{[[} (i, x = \text{fork}(f), \rho, k); \ldots \text{]}, \sigma) \rightarrow \text{[[} (i, \text{skip}, \rho[x \mapsto i'], k); (i', c, \{\}, \text{nil}); \ldots \text{]}, \sigma)}
\]

\[ (e, \rho, \sigma) \downarrow j \]

\[
\frac{\sigma(f) = (\cdot, c) \quad i' \text{ fresh}}{\text{[[} (i, \text{join}(e), \rho, k); (j, \text{skip}, \rho_j, \text{nil}); \ldots \text{]}, \sigma) \rightarrow \text{[[} (i, \text{skip}, \rho, k); \ldots \text{]}, \sigma)}
\]
Concurrent fork-join example:

```c
int x, y = 0;

int main()
{
    void thread_fun()
    {
        tid t = fork(thread_fun);  // (0, ρ_0)
        y = 2;
        x = 1;
        join(t);
    }

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

```c
int x, y = 0;

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

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

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

int x, y = 0;

int main() {
    void thread_fun() {
        tid t = fork(thread_fun);
        y = 2;  // (1, ρ₀)
    }

    void thread_fun() {
        tid t = fork(thread_fun);
        x = 1;
        join(t);  // (0, ρ₀[t → 1])
    }

    join(t);  // (0, ρ₀[t → 1])
}

{x ← 1, y ← 0}
Concurrency: Fork-Join

int x, y = 0;

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

{x \mapsto 0, y \mapsto 2}

{y = 2; (1, \rho_0)}
Concurreny: Fork-Join

```c
int x, y = 0;

int main(){
    tid t = fork(thread_fun);
    x = 1;
    join(t); // (0, ρ₀[t ↦ 1])
}

void thread_fun(){
    y = 2;
}
```

{x ↦ 1, y ↦ 2}
Concurrent Fork-Join Example

```c
int x, y = 0;

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

void thread_fun(){
    y = 2;
}
```

Initial state: $(0, \rho_0[t \mapsto 1])$

Final state: $\{x \mapsto 1, y \mapsto 2\}$