CS 476 – Programming Language Design

William Mansky
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

```
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

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

```c
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

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

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

Examples:

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

\[ \text{while } (x < 100) \text{ do } (x = x + 1 \mid\mid y = y + 2) \]
Concurrency: Typing and Semantics

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

\[
\begin{align*}
(c_1, \sigma) & \to (c'_1, \sigma') \\
(c_1 \parallel c_2, \sigma) & \to (c'_1 \parallel c_2, \sigma') \\
(c_2, \sigma) & \to (c'_2, \sigma') \\
(c_1 \parallel c_2, \sigma) & \to (c_1 \parallel c'_2, \sigma') \\
\text{(skip} \parallel \text{skip}, \sigma) & \to (\text{skip}, \sigma')
\end{align*}
\]
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}, ...
```
Concurrency: Shared Variables

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

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

\[ x = 0; \]
\[ y = x; \]
\[ x = y + 1; \]
\[ z = x; \]
\[ x = z + 1; \]
\[ \{x = ?\} \]
Concurrency: Shared Variables

\[ x = 0; \]

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

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

\[
x = 0;
\]

acquire(s);  acquire(s);
\[
x = x + 1;  x = x + 1;
\]

release(s); release(s);

\{x = 2\}
Concurrency: Locks

\[ E := \texttt{<#> | <ident> | } E + E | ... \]

\[ C := \texttt{<ident> = E | } C; C | \texttt{skip | while } E \texttt{do } C | ... | (C \| C) \]

\[ \quad | \texttt{acquire(<ident>) | release(<ident>)} \]

\[ \sigma(x) = 0 \]

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

\[ \sigma(x) = 1 \]

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

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

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

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

\( \{ x = 0, \, s = 1 \} \)

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

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

\{x = 1, s = 1\}

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

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

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

\{x = 1, s = 0\}

\[
\begin{align*}
\text{acquire}(s); \quad \sigma(x) = 0 \\
x = x + 1; \quad (\text{acquire}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 1]) \\
\text{release}(s); \quad \sigma(x) = 1 \\
\text{release}(x), \sigma) \rightarrow (\text{skip}, \sigma[x \mapsto 0])
\end{align*}
\]
Concurrency: Fork-Join

- Most real languages don’t use the $\parallel$ 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

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

int x = 0;

void thread_fun()
{
    tid t = fork(thread_fun);
    x = x + 1;
    x = x + 1;
    join(t);
}
Concurrency: Fork-Join Syntax

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

\[ C ::= \ldots \mid \langle\text{ident}\rangle = \text{fork}(\langle\text{ident}\rangle); \mid \text{join}(E); \]

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

\[
\begin{align*}
\Gamma(x) &= \text{tid} & \Gamma(f) &= \text{void}() \\
\Gamma \vdash x &= \text{fork}(f); : \text{ok}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \text{tid} \\
\Gamma \vdash \text{join}(e); : \text{ok}
\end{align*}
\]
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 
\[
\left[ \left( i_1, c_1, \rho_1, k_1 \right); \ldots; \left( i_n, c_n, \rho_n, k_n \right) \right], \sigma
\]

• At each step, we pick one thread to execute:
\[
\begin{align*}
(c_j, \rho_j, k_j, \sigma) &\rightarrow (c'_j, \rho'_j, k'_j, \sigma') \\
\left[ \left( i_1, c_1, \rho_1, k_1 \right); \ldots; \left( i_j, c_j, \rho_j, k_j \right); \ldots; \left( i_n, c_n, \rho_n, k_n \right) \right], \sigma &\rightarrow \\
\left[ \left( i_1, c_1, \rho_1, k_1 \right); \ldots; \left( i_j, c'_i, \rho'_i, k'_i \right); \ldots; \left( i_n, c_n, \rho_n, k_n \right) \right], \sigma'
\end{align*}
\]
Concurrency: Fork-Join Semantics

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

\[
\begin{array}{c}
\frac{((i, x = \text{fork}(f), \rho, k); \ldots], \sigma) \rightarrow ((i, \text{skip}, \rho[x \mapsto i'], k); (i', c, \{\}, \text{nil}); \ldots], \sigma)}{(e, \rho, \sigma) \downarrow j}
\end{array}
\]

\[
\frac{(i, \text{join}(e), \rho, k); (j, \text{skip}, \rho_j, \text{nil}); \ldots], \sigma) \rightarrow ((i, \text{skip}, \rho, k); \ldots], \sigma)}
\]