Threads

CS 520
Dept. of Computer Science
Univ. of New Hampshire
thread: a virtual CPU
executes instructions
has registers
uses stack

useful for performance — exploit multiple CPUs

useful for design — e.g., assign thread to wait for GUI events while another thread does main computation
But
memory is shared
threads need to contend for it
threads can interfere with each other
Consider two threads incrementing the same variable.

\[ \frac{T_0}{i++} \quad \frac{T_1}{i++} \]

\[ i = ? \]

First, what is `i++` actually?

- load \( i \)
- add 1
- store \( i \)

*not atomic*
Need to consider interleavings of the low-level instructions:

\[ \text{To} \]
\[ \varnothing \Leftarrow \text{load } i \]
\[ \text{add } 1 \]
\[ \text{store } i \rightarrow 1 \]

\[ \text{To} \]
\[ \text{load } i \rightarrow \varnothing \]
\[ \text{add } 1 \]
\[ \text{store } i \rightarrow 1 \]

\[ i = X + 1 \]

Race condition: possibility of incorrect results due to timing
need mechanism to ensure only one thread at a time is in the critical section:

\[ i = 0 \]

\[ T_0 \]

\[ \text{lock} \]

\[ \text{load } i \leftarrow \varnothing \]

\[ \text{add } 1 \]

\[ \text{store } i \rightarrow 1 \]

\[ \text{unlock} \]

\[ T_1 \]

\[ \text{lock} \]

\[ \ldots \}

\[ \text{thread is blocked} \]

\[ \text{load } i \leftarrow \varnothing \]

\[ \text{add } 1 \]

\[ \text{store } i \rightarrow 2 \]

\[ \text{unlock} \]

\[ i = x \times 2 \]
Also need mechanism to control interaction of threads.

For example, consider a thread producing a series of values to be consumed by another thread.

Consumer thread can't consume until producer produces value

Producer thread can't produce until last value consumed

assuming a buffer of length 1
Shared data

value Present (boolean)
value 2 initialized to False

also shared
mu  - a lock (mutex)
cu  - a condition variable

producer

lock (mu)
while (!value Present) {
    wait (cu, mu)
}
value = newValue;
value Present = true;
signal (cu)
unlock (mu)

consumer

lock (mu)
while (!value Present) {
    wait (cu, mu)
}
newValue = value;
value Present = false;
signal (cu)
unlock (mu)

note: when wait is called, a lock is released (caller must own a single lock)
when signal is called, first waiter is released but newly released waiter must wait to re-acquire lock
bad things can happen:

race condition

deadlock - all threads are blocked

livelock - threads spend so much time responding to each other's actions that they cannot make progress on their work

starvation - some threads dominate access to shared resource so that other threads cannot access it