Intro to POSIX Threads with FLTK

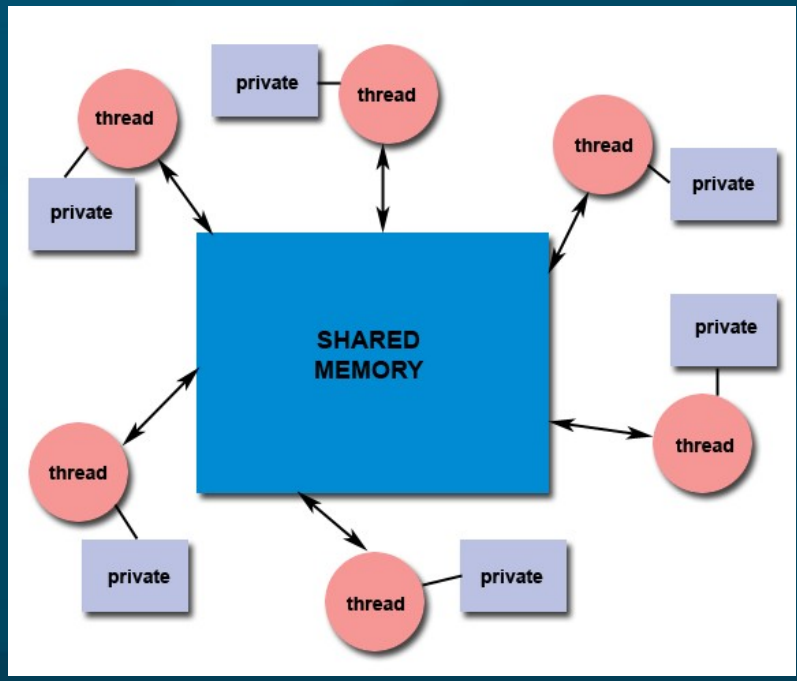
See:
- FlChat/ example code

25 Mar 2009
CMPT166
Dr. Sean Ho
Trinity Western University

TRINITY WESTERN UNIVERSITY

Threads and parallelism

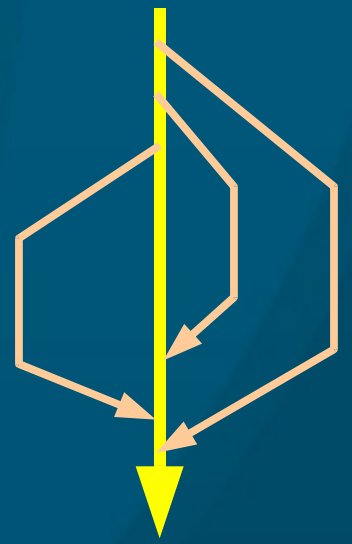
- Threads are lightweight processes
- Threads allow concurrency
  - Make use of multiple processors
  - But still useful even on uniprocessor
- Threads use shared memory
  - Synchronization issues for shared objects
    - Thread-safe code?
  - May also have local (private) variables



Parallel program models

- How to divide work amongst threads?
- Master/worker:
  master thread assigns work to worker threads
  - Master typically handles UI, input
  - Static or dynamic worker pool
- Coworkers: all threads are peers:
  - Main thread participates in doing work
- Pipeline: each thread works on a different part of the task: e.g., automobile assembly line
  - Function parallelism vs. data parallelism

Threads model: PThreads



- POSIX threading: fork/join model
- Start with parent thread (main program)
- Create child thread(s):
  - Specify a callback for the child to execute
  - Optional parameter to callback: (void*)
  - Shared memory access to same data
  - Children may send messages to parent
    - May trigger parent to execute a callback
  - Child threads exit when callback finished

PThreads API

- pthread_create():
  - Parent calls this to start new child
  - Specify start function for child to run
- pthread_exit():
  - Signals this thread is done
  - Implicit at end of child's start function
- pthread_join():
  - Parent calls to wait for result from a child
- pthread_self(): returns my thread ID (pthread_t)

PThreads: get results from child

- Thread callbacks are subroutines declared to return void* and take one void* parameter
  - void* workerThread(void* d) { ... }
- Parent may pass any user data to child via the void* parameter
- Child may pass user data back to parent via the void* return value
- Parent calls pthread_join() to wait for child to finish and fetch child's returned data

PThreads vs Windows threads

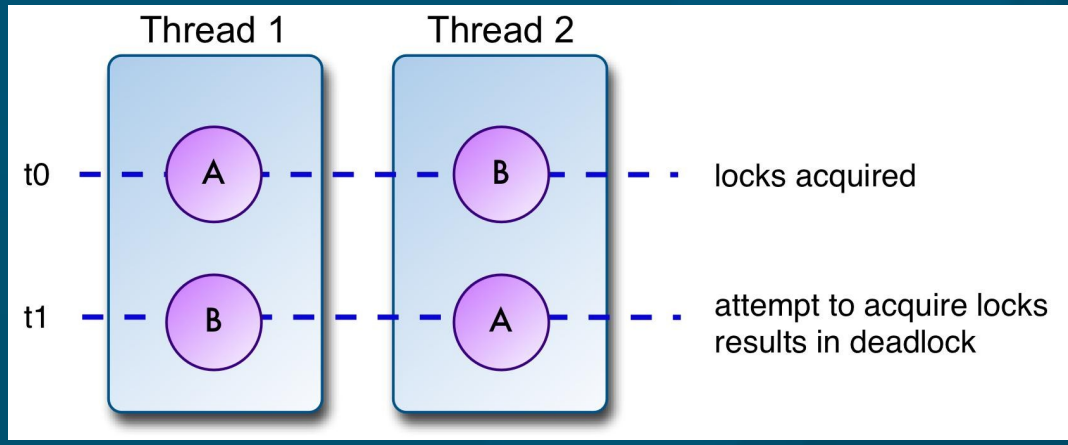
- PThreads (POSIX threads) are a standard:
  - #include <pthread.h>
- But Windows has its own threading library:
  - #include <process.h>
- FLTK provides a small wrapper around both:
  - #include “fl_threads.h”
  - fl_create_thread(*tid*, *callback*, *userdata*);
    - Fl_Thread tid lets parent track child
    - callback is function for child to execute
    - userdata (optional) is passed to callback

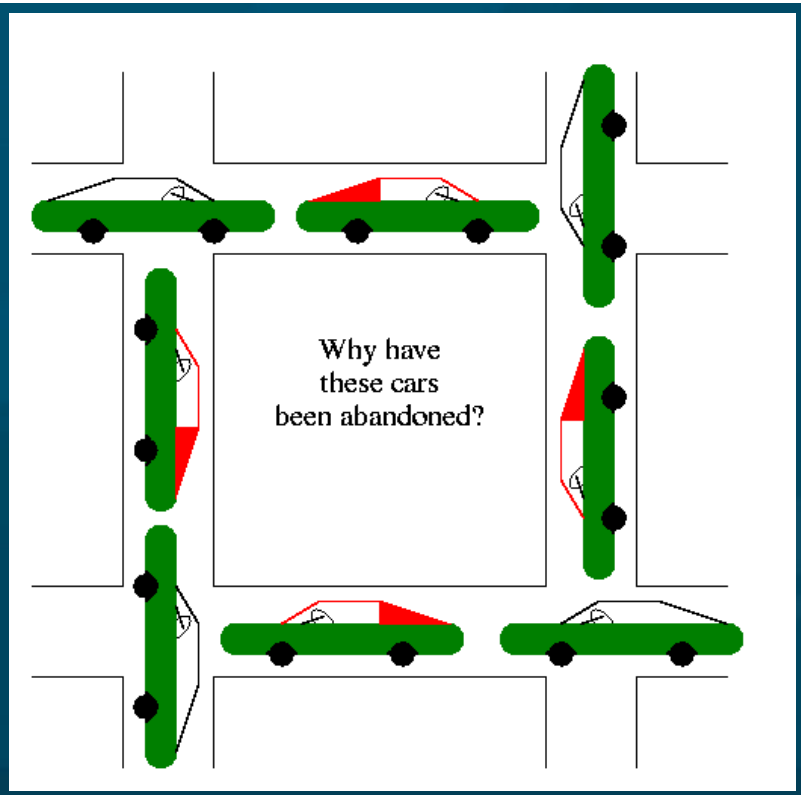
Issues with threads: locking

- Big problem whenever we have concurrent threads accessing shared data: data corruption
  - e.g., threads == children playing; shared resources == toys/blocks
- Mutual exclusion (mutex): only one thread accesses shared object at a time
- Locks: a way to implement mutex
  - Thread asks for lock before modifying object
  - If it gets the lock, it can modify
  - If not, wait (block) until the lock is freed
  - Free the lock when done modifying

Problem with locks: deadlock

- Shared resources A, B each have own lock
- Thread 1 locks A, then asks to lock B
- Thread 2 locks B, then asks to lock A
- → both threads hang forever! Deadlocked
- → be careful with locks; only hold lock for minimum time needed





Locking in FLTK

- FLTK provides one global lock so that multiple threads won't change the GUI simultaneously
- First call Fl::lock() in main() to enable threads
- Then before a thread modifies any shared object:
  - Fl::lock();
  - myWindow->show(); // or other shared
  - Fl::unlock();
- See FLTK doc ch10.

When to use Fl::lock()

- Before a shared resource is modified
  - e.g., both parent and child want to write to a string buffer
- Before any FLTK windowing operation
  - show()/hide()
  - timers (fl_add_timeout())
  - changing window decorations
  - In general, only the parent thread should do these

Fl::awake()

- Children can send messages to the parent:
  - void* msgToParent;
  - Fl::awake(msgToParent);
- Parent (doing main FL event loop) checks for messages with
  - void* msgFrChild = Fl::thread_message();
- Message may be pointer to any object (void*)
- Child may also ask parent to run a callback:
  - void* runMe(void* u) { ... }
  - Fl::awake(runMe, userdata);

Example code: FlChat

- Each of client and server has its own UI and main class: Client, ClientUI, Server, ServerUI
- ServerUI: creates Server object and initializes
  - Splits off a thread to wait and listen
- ClientUI: creates Client object
  - Upon connect, splits thread to receive
- This is still a serial server:
  - Can't handle multiple simultaneous clients
  - Extension: use threads to do switchboard