CSE 451 Section 6
Synchronization, Project 2

Project 2

- More fun than a barrel of monkeys:
  - Threaded programming
  - Thread-safe programming
  - Task-quere programming
  - Event-driven programming
  - Sprinkled with neat tools.
    - Pthreads, gtk+, valgrind, helgrind,

Project 2

- Teams of up to 3.
- Parts 1,2,3 due Thursday, May 12 at 11:59 PM
- Parts 4,5,6 due Monday, May 23 at 11:59 PM

Hash Tables 2: Electric Boogaloo

- Step 1: Expand a hash table
  - Add a destroy function
    - Includes callback function pointers to destroy keys and values
    - Should ensure all parts of the hash table are destroyed. No memory leaks.
  - Write a test function
  - You don't need to worry about this being threadsafe.
  - Valgrind and gdb will be useful.

Destroy callbacks

- typedef void (*hash_value_destroy_fn_t)(void* value);

// callback to client code to destroy a value, if the hash table isn't empty when it's destroyed

- typedef void (*hash_key_destroy_fn_t)(void* key);

// callback to destroy a key

- extern int hashTable_destroy(hashTable_t t, hash_key_destroy_fn_t   destroyKeyFunc, hash_value_destroy_fn_t destroyValFunc);

Administrivia

- You are not required to use your own hash tables if you don't want to. If you are able to (I haven't actually looked), you may find a working implementation ***so long as it fits or is adapted to our provided header file.***

- I graded your hash tables but it's always possible I missed an error you made. Make sure you thoroughly test your hash table still.

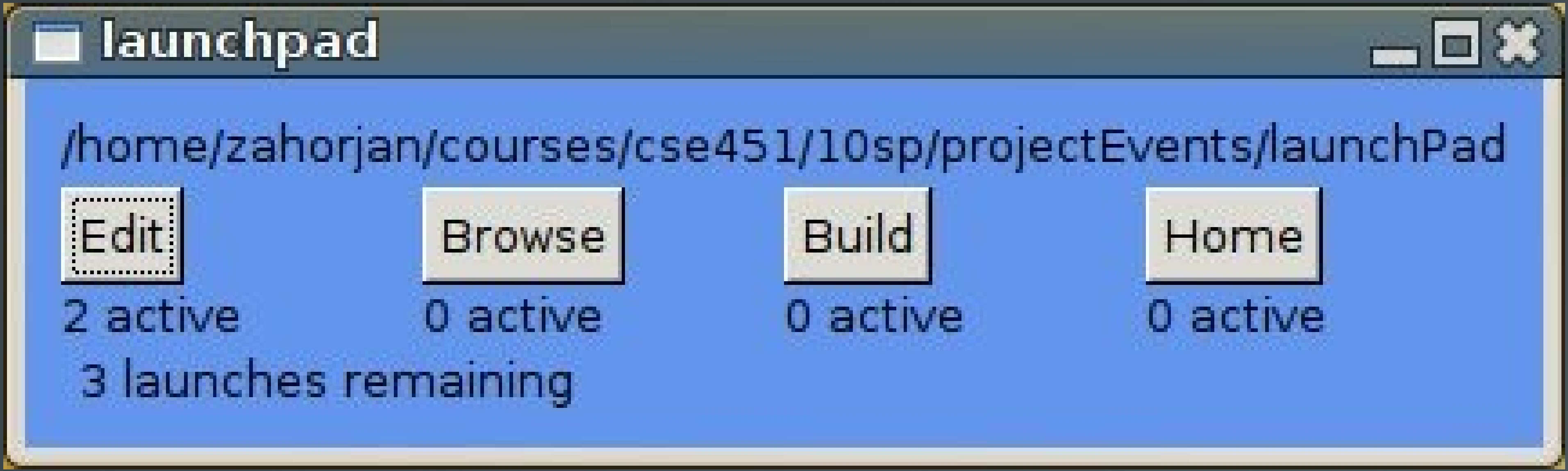
Make Q Omnipotent

- Part 2: make the queue thread safe.
  - Eliminate all “non-benign” race conditions
  - We're using pthreads so use pthreads primitives
    - Actually use our LOCK macros which fill in with the synchronization primitives.
  - Do not turn the queue into a monitor. Use more than more one lock for the queue.
    - But don't worry too much about lock granularity beyond that. Understand the concepts behind lock granularity but don't fret over this one implementation.

Lock trade-offs

- Spin locks vs Blocking locks
  - Spin locks will idle the CPU. Creates “waste”
  - Blocking locks require a context switch. Creates “overhead”
  - Spin locks are generally for multiprocessors (when another CPU may finish critical section) and small critical sections (waste < ctx switch overhead)
- Few locks versus many locks
  - Many locks (smaller critical sections) may allow for more concurrency
  - Few locks have less overhead (less trivial than it sounds)

Part 3



/home/zahorjan/courses/cse451/10sp/projectEvents/launchPad

2 active 0 active 0 active 0 active

3 launches remaining

GUI

- Part 3: Build a hollow GUI.
  - Use gtk+ to set up a window.
  - Buttons do not need to launch any applications (yet)
  - Window should have:
    - Currently working directory printed
    - Two or more buttons
    - Labels for each button saying “N clicks” where N is a counter for how many times the button has been clicked

Parts 4,5,6

- More on this next time.
  - Add a layer on top of your queue to create a task queue
  - Create the event driven system for your GUI
  - Glue all the pieces together
  - Report

Other caveats

- Factory.h facilitates some class-like behavior
  - Has some particular semantics. Check out the sample code for how to use its macros.
- Use LOCK macros (lockDef.h)
  - They allow for easy switching between spin locks and blocking locks.
- Valgrind for checking memory leaks
- Gdb for debugging
- Helgrind for checking race conditions
  - No race detection tool is perfect. You may or may not find helgrind helpful. It will produce false positives, false negatives, and only check for race conditions that appeared during an actual execution.

General Tips (same as usual)

- Don't make mistakes.
  - ...at least try to make as few mistakes on your first pass. Think before you code.
  - Know where your critical sections are and how to protect them.
- Do not leave this to the last minute.
  - Often, the ability and act of walking away from code with help you digest the task.
  - You lose this ability the night before a due date.

Synchronization

High-level
- Monitors
- Java synchronized method

OS-level support
- Special variables – mutex, semaphor, condition var
- Message passing primitives

Low-level support
- Disable/enable interrupts
- Atomic instructions (test_and_set)

Disabling/Enabling Interrupts

```
Thread A:
   disable_irq()
   critical_section()
   enable_irq()
```

```
Thread B:
   disable_irq()
   critical_section()
   enable_irq()
```

- Prevents context-switches during execution of critical sections
- Sometimes necessary
  - E.g. to prevent further interrupts during interrupt handling
- Many problems

Semaphore review

Semaphore = a special *variable*

Manipulated atomically via two operations:

- P (wait)
- V (signal)

Has a counter = number of available resources

P decrements it

V increments it

Has a queue of waiting threads

If execute wait() and semaphore is free, continue

If not, block on that waiting queue

signal() unblocks a thread if it's waiting

Mutex is a binary semaphore (capacity 1)

Condition Variable

A “place” to let threads wait for a certain event to occur while holding a lock

It has:

Wait queue

Three functions: *wait*, *signal*, and *broadcast*

- *wait* – sleep until the event happens
- *signal* – event/condition has occurred. If wait queue nonempty, wake up *one* thread, otherwise *do nothing*
  - Do not run the woken up thread right away
  - FIFO determines who wakes up
- *broadcast* – just like *signal*, except wake up all threads

Typically associated with some logical condition in program

Condition Variable (2)

`cond_wait(sthread_cond_t cond, sthread_mutex_t lock)`

Should do the following atomically:

- Release the lock (to allow someone else to get in)
- Add current thread to the waiters for `cond`
- Block thread until awoken

Read man page for **`pthread_cond_[wait|signal|broadcast]`**

Must be called while holding **`lock`**! -- Why?

Semaphores vs. CVs

Semaphores

- Used in apps
- wait() does not always block the caller
- signal() either releases a blocked thread, if any, or increases sem. counter.

Condition variables

- Typically used in monitors
- Wait() always blocks caller
- Signal() either releases blocked thread(s), if any, or the signal is lost forever.

Sample synchronization problem

Late-Night Pizza

- A group of students study for cse451 exam
- Can only study while eating pizza
- Each student thread executes the following:

  -
    ```
    while (must_study) {
     pick up a piece of pizza;
     study while eating the pizza;
     }
    ```
- If a student finds pizza is gone, the student goes to sleep until another pizza arrives
- First student to discover pizza is gone orders a new one.
- Each pizza has S slices.

Late-Night Pizza

- Synchronize student threads and pizza delivery thread
- Avoid deadlock
- When out of pizza, order it exactly once
- No piece of pizza may be consumed by more than one student

Semaphore / mutex solution

```
shared data:
    semaphore_t pizza;  (counting sema, init to 0, represent
    number of available pizza resources)
    semaphore_t deliver; (init to 1)
    int num_slices = 0;
    mutex_t mutex; (init to 1) // guard updating of num_slices
```

```
Student {
   while (must_study) {
    P(pizza);
    acquire(mutex);
    num_slices--;
    if (num_slices==0)
      // took last slice
      V(deliver);
    release(mutex);
    study();
    }
}
```

```
DeliveryGuy {
   while (employed) {
      P(deliver);
      make_pizza();
      acquire(mutex);
      num_slices=S;
      release(mutex);
      for (i=0; i < S; i++)
        V(pizza);

   }
}
```

Condition Variable Solution

```
int slices=0;
Condition order, deliver;
Lock mutex;
bool has_been_ordered = false;
```

```
Student() {
  while(diligent) {
    mutex.lock();
    if( slices > 0 ) {
      slices--;
    }
    else {
      if( !has_been_ordered ) {
        order.signal(mutex);
        has_been_ordered = true;
      }
      deliver.wait(mutex);
    }
    mutex.unlock();
    Study();
  }
}
```

```
DeliveryGuy() {
  while(employed) {
    mutex.lock();
    order.wait(mutex);
    makePizza();
    slices = S;
    has_been_ordered = false;
    mutex.unlock();
    deliver.broadcast();
  }
}
```