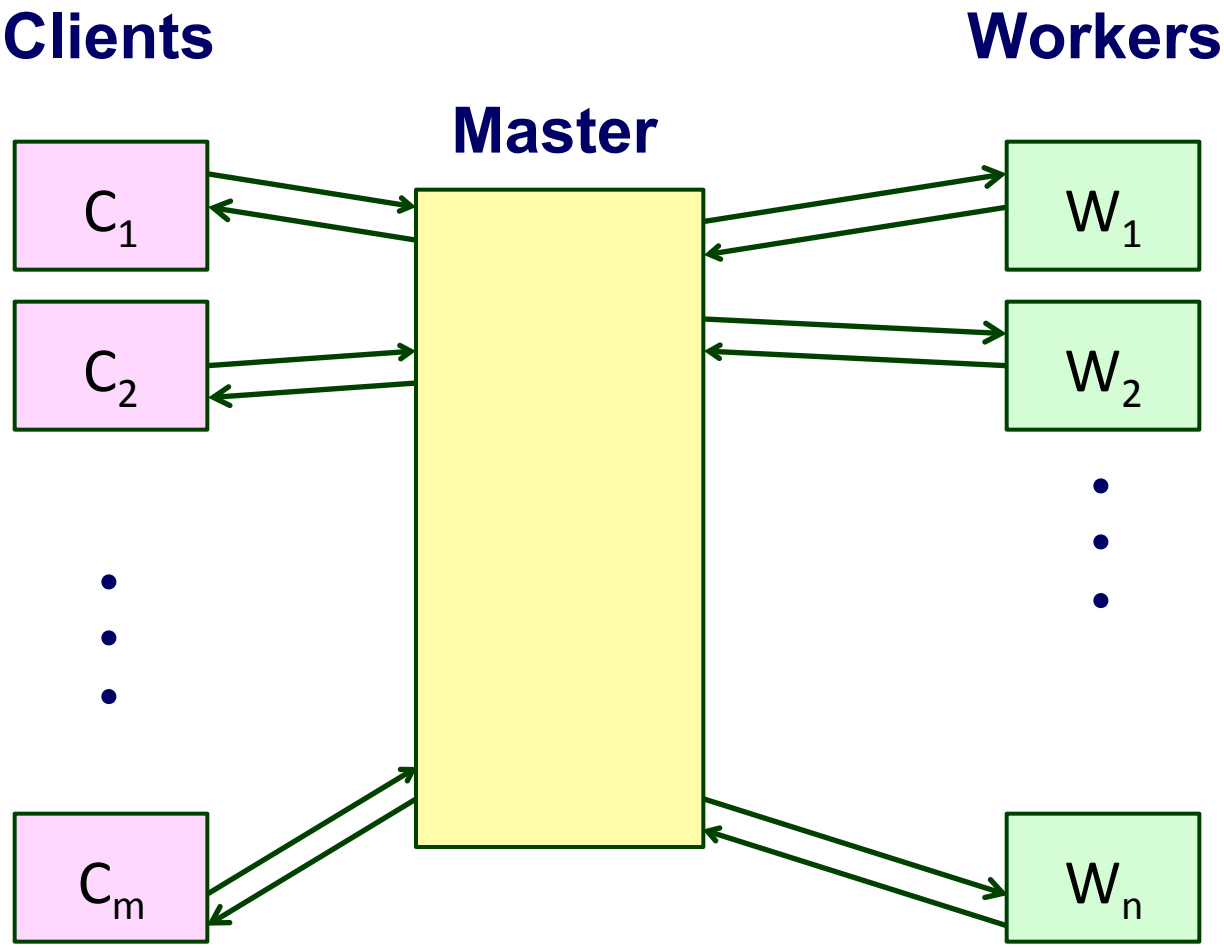
# Preparing for Assignment 4

15-418/618: Parallel Computer Architecture and Programming

Recitation. March 24, 2017

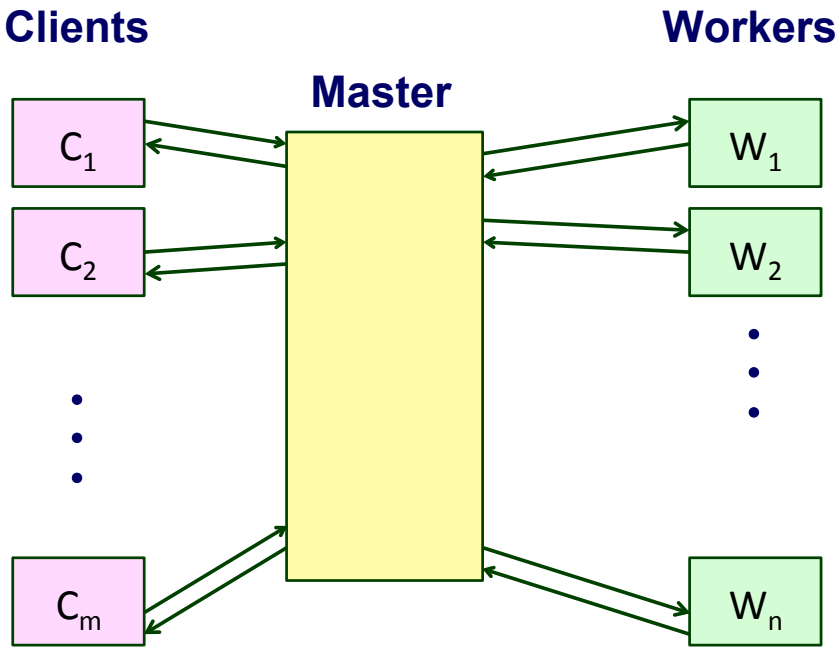
Randy Bryant

# Overall Structure of System



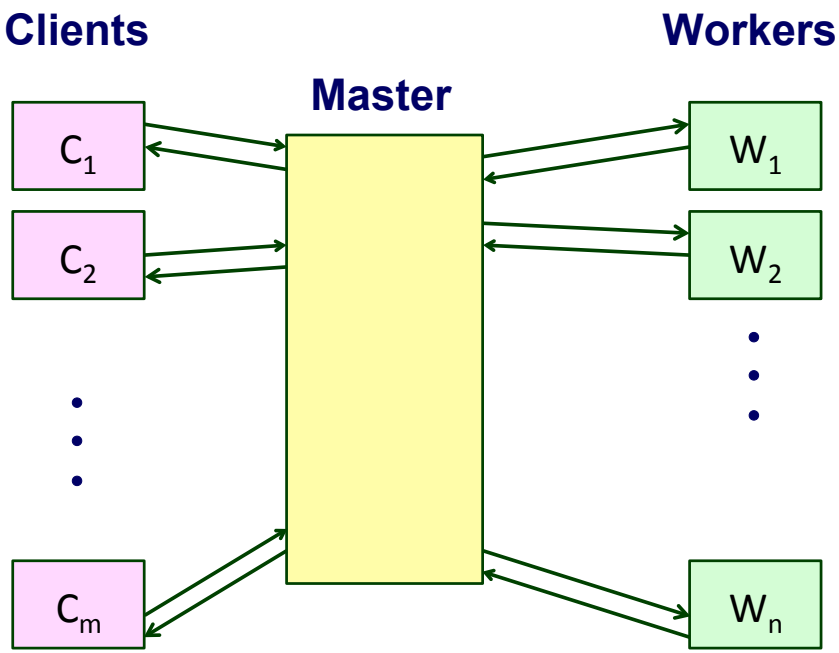
# Overall System Structure

- Clients are remote, requesting services
- Master is single-threaded
- Workers are multi-threaded. They carry out the requests
- All communication is via asynchronous messages



# Master Operation

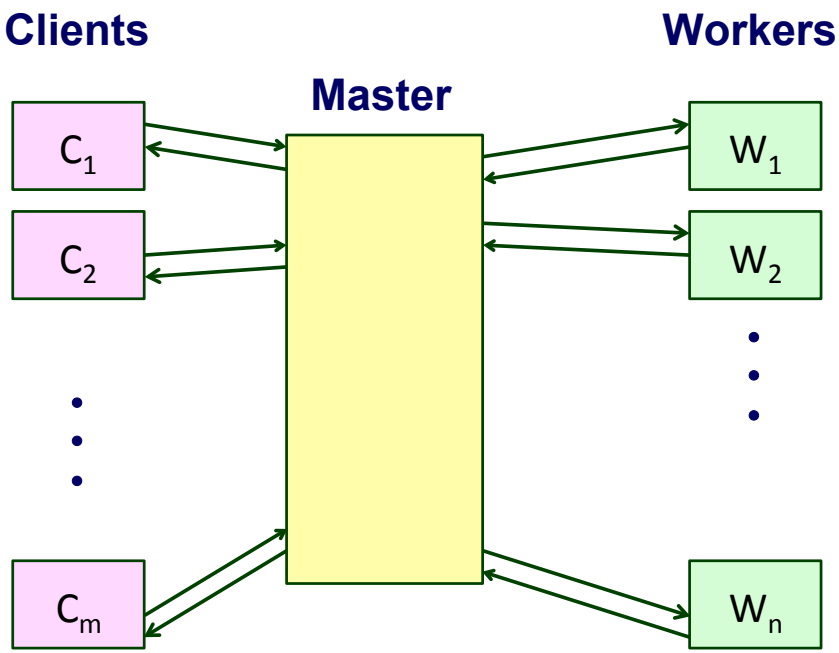
- **Continually responds to *events***
- **Event types**
  - Request from client
  - Response by worker
  - New worker available
  - Period "tick"
- **Master actions**
  - Send request to worker
  - Send response to client
  - Ask for new worker
  - Kill existing worker




- **Supplied code implements event manager. Your job is to handle the different event types**

# Worker Operation

- **Maintains queue of requests from master**
- **Maintains pool of threads**
  - Each thread runs loop
    - Get request from queue
    - Execute
    - Send response to master
- **You must implement most of this**



# Design Features

- **Client concerns**
  - Different request types have different resource requirements
  - Want to provide minimum latency, especially for small jobs
  - Your master must control mapping of requests to workers via its scheduler
- **Resource concerns**
  - Demand varies over time
  - Can add more workers to handle increased demand
    - Takes a while for them to come online
    - Penalized for too much resource use
  - Your master must add and kill workers dynamically

# Request Types

- **418wisdom**
  - Compute intensive
  - Low memory footprint
  - Same each time
- **countprimes**
  - Compute intensive
  - Low memory footprint
  - Varies each time
- **compareprimes**
  - Implemented as 4 countprimes requests
- **tellmenow**
  - Minimal computation or memory
  - Must have quick response
- **projectideas**
  - Fills entire L3 cache

# Benchmarking Study of Request Types

- **All running on GHC machine**
  - 8 cores, each 2x hyperthreaded
  - 20MB L3 cache
  - Latedays will be slightly different
- **Methodology**
  - Code in

/afs/cs.cmu.edu/academic/class/15418-s17/recitations/recw8/recw8-code/asst4_bench

  - Request some combination of different jobs
  - Possibly restrict number of outstanding jobs of given type
- **Measurements**
  - For each job type: Minimum, Average, Maximum elapsed time
  - Overall elapsed time
  - All times in milliseconds

# Benchmark Run Example

```
16 threads
        wisdom  primes  tellme  project
Jobs:   64      0       0       4
Limits: 64      0       0       1
Min:    1056    --      --      3978
avg:    1229    --      --      4335
Max:    1343    --      --      5161
Total time: 17343ms
```

- 16 threads
- Jobs: 64 418wisdom + 4 projectideas
- Limits: None for 418wisdom, 1 at a time for projectideas
- Show min/avg/max for each type
- Overall time

# 418wisdom

- 64 jobs
- Vary number of threads
- No limit on number of threads running requests simultaneously

| Threads | Average | Total |
|---|---|---|
| 8 | 1080 | 8687 |
| 16 | 1243 | 5037 |
| 32 | 2274 | 5142 |
| 64 | 4342 | 5041 |

- Questions (remember 8 cores, each 2x hyperthreaded)
  - Why does average increase with number of threads?
  - Why does overall time improve going from 8 to 16 threads?
  - Why does it stay flat beyond 16 threads?

# Compute Intensive Requests

- 64 jobs
- 16 threads

| | 418wisdom | countprimes | tellmenow |
|---|---|---|---|
| Minimum | 1207 | 2 | < 1 |
| Average | 1245 | 1254 | < 1 |
| Maximum | 1344 | 2718 | < 1 |

- Questions:
  - What do these numbers imply about scheduling?

# projectideas

- 4 jobs
- 16 threads
- Limit number of simultaneous jobs

| Limit | Average | Total |
|---|---|---|
| 1 | 4028 | 16116 |
| 2 | 10473 | 21077 |
| 4 | 14293 | 15027 |

- Questions:
  - Why does both the average and the total increase when going from 1 to 2 running simultaneously?
  - Why does the total drop when run 4 simultaneously?
  - What would be a good scheduling policy?

# Mixing 418wisdom and projectideas

- Combine 418wisdom jobs + projectideas jobs
- 16 threads
- Limit of 1 simultaneous projectideas job

| projectideas jobs | 418wisdom jobs | 418wisdom average | projectideas average | Total |
|---|---|---|---|---|
| 0 | 64 | 1243 | — | 5037 |
| 4 | 0 | — | 4028 | 16116 |
| 4 | 64 | 1229 | 4151 | 16951 |

- Questions:
  - Why is the average time for 418wisdom unaffected by projectideas?
  - Why is the average time for projectideas unaffected by 418wisdom?
  - What does this imply about scheduling policies?

# Useful Features of C++

- Reference parameters
- C++ strings
- Class basics
- Templates
- Memory management

/afs/cs.cmu.edu/academic/class/15418-s17/recitations/recw8/recw8-code/cpp

# Reference Parameters (eg1-refparam)

```
// Demonstration of reference parameters
#include <iostream>

// C style
int pincr(int *xp) {
    int r = *xp;
    (*xp)++;
    return r;
}

// C++ style
int rincr(int &x) {
    int r = x;
    x++;
    return r;
}

int main(int argc, char *argv[]) {
    int x = 1;
    int y = pincr(&x);
    int z = rincr(x);
    std::cout << "x=" << x << ", y=" << y
              << ", z=" << z << std::endl;
    return 0;
}
```

- Function rincr uses call-by-reference
- What are resulting values of x, y, and z?

# Implementation of Reference Parameters

## C Pointer Code

```
// C style
int pincr(int *xp) {
    int r = *xp;
    (*xp)++;
    return r;
}
```

## C++ Reference Code

```
// C++ style
int rincr(int &x) {
    int r = x;
    x++;
    return r;
}
```

```
_Z5pincrPi:
   movl     (%rdi), %eax
   leal     1(%rax), %edx
   movl     %edx, (%rdi)
   ret
```

```
_Z5rincrRi:
   movl     (%rdi), %eax
   leal     1(%rax), %edx
   movl     %edx, (%rdi)
   ret
```

# C++ strings (eg2-string)

```
// Demonstration of C++ strings
#include <string>
#include <iostream>
#include <stdio.h>

std::string stringify(char *s) {
    std::string ss = s;
    return ss;
}

int main(int arg, char *argv[]) {
    char buf[10];
    std::string strings[5];
    for (int i = 0; i < 5; i++) {
        sprintf(buf, "i=%d", i);
        strings[i] = stringify(buf);
    }
    for (int i = 0; i < 5; i++) {
        std::cout << "String " << i << ":"
                  << strings[i] << std::endl;
    }
    return 0;
}
```

- C++ provides managed strings
- Automatic garbage collection via reference counting
- How is this possible, given rest of language requires explicit memory management?

# Comparing to C strings (eg2a-string)

```
// Comparison with C strings
// Does this code work?
#include <iostream>
#include <stdio.h>
#include <string.h>

char *stringify(char *s) {
    char buf[10];
    strcpy(buf, s);
    return buf;
}

int main(int arg, char *argv[]) {
    char buf[10];
    char *strings[5];
    for (int i = 0; i < 5; i++) {
        sprintf(buf, "i=%d", i);
        strings[i] = stringify(buf);
    }
    for (int i = 0; i < 5; i++) {
        std::cout << "String " << i << ":"
        << strings[i] << std::endl;
    }
    return 0;
}
```

- C strings are simply arrays of characters
- No management
- Programmer must explicitly manage storage

# Implementation of string-returning function

```
// Nominal version
std::string stringify(char *s) {
    std::string ss = s;
    return ss;
}

// True implementation
void rstringify(char *s, std::string &ss) {
    ss = s;
}
```

## Returning struct in C

- Caller allocates space on stack
- Passes pointer to the callee
- Callee fills in fields

## Returning object in C++

- Same strategy

# Class Example (eg3-class)

```
// Implementation of linked list of integers
#include <iostream>

class ListEle {
private:
    int val;
    ListEle *next;

public:
    ListEle(int v, ListEle *np)
        { val = v; next = np; }

    ListEle *getNext() { return next; }

    int getValue() { return val; }
};
```

- C++ class is a variant of a struct
  - Reference with "." or "->" operator
- Can make some fields inaccessible outside of class code
- Can write one or more constructor functions

# Class Example (cont)

```
class List {
private:
    ListEle *head;

public:
    List() { head = NULL; }

    void insert(int v) { head = new ListEle(v, head); }

    int front() {
        if (head) return head->getValue();
        else      return -1;
    }

    void pop() {
        if (head) head = head->getNext();
    }

    bool isEmpty() { return head == NULL; }
};
```

- Allocate element with *new* operation
- Can you find the memory leak?
- How would you fix it? (see eg3a-class.cpp)

# Using Class

```
int main(int arg, char *argv[]) {
    List ls;
    for (int i = 0; i < 5; i++) {
        ls.insert(i);
    }

    while (!ls.isEmpty()) {
        int v = ls.front();
        std::cout << "Popped value " << v << std::endl;
        ls.pop();
    }
    return 0;
}
```

- List head allocated on stack
- List elements allocated on heap

# Template Example (eg4-template)

```
// Implementation of linked list of integers
#include <iostream>

template <class T>
class ListEle {
private:
    T val;
    ListEle *next;
public:

    ListEle(T v, ListEle *np)
    { val = v; next = np; }

    ListEle *getNext() { return next; }

    T getValue() { return val; }
};
```

- Template allows generic code that is later instantiated for one or more types

# Templates (eg4-template)

```
template <class T>
class List {
private:
    ListEle<T> *head;
public:
    List() { head = NULL; }

    void insert(T v) { head = new ListEle<T>(v, head); }

    T front() { return head->getValue(); }

    void pop() {
        if (head) {
            ListEle<T> *save = head;
            head = head->getNext();
            delete save;
        }
    }

    bool isEmpty() { return head == NULL; }
};
```

- Use T in places where previously had int
- Note use of *delete* to avoid memory leak

# Template instantiation (eg4-template)

```
int main(int arg, char *argv[]) {
    List<int> ls;
    for (int i = 0; i < 5; i++) {
        ls.insert(i);
    }

    while (!ls.isEmpty()) {
        int v = ls.front();
        std::cout << "Popped value " << v << std::endl;
        ls.pop();
    }
    return 0;
}
```

- Creates list of integers
- But, can use for list of any object type

# Freeing storage with destructors (eg5-deallocate)

## Destructor for ListEle

```
~ListEle() {
     std::cout << "Destroying element with value " << val << std::endl;
     if (next)
         delete next;
```

## Destructor for List

```
~List() { if (head) delete head; }
```

- Called when object deallocated
- Stack-allocated objects deallocated when exit scope of declaration
- Heap-allocated objects only deallocated by *delete*

# Unintended deallocation (eg5a-deallocate)

```
int main(int arg, char *argv[]) {
    List ls;
    for (int i = 0; i < 5; i++) {
        ls.insert(i);
    }

    while (!ls.isEmpty()) {
        int v = ls.front();
         std::cout << "Popped value " << v << std::endl;
        ls.pop();
    }
    return 0;
}
```

- What happens when call `ls.pop()`?
- Why doesn't this happen in garbage-collected languages?

```
void pop() {
    if (head) {
        ListEle *save = head;
        head = head->getNext();
        delete save;
    }
}
```

# Controlling deallocation (eg5b-deallocate)

```
void pop() {
    if (head) {
        ListEle *save = head;
        head = head->getNext();
        save->unlink();
        delete save;
    }
}
```

- Set next pointer to null before deleting

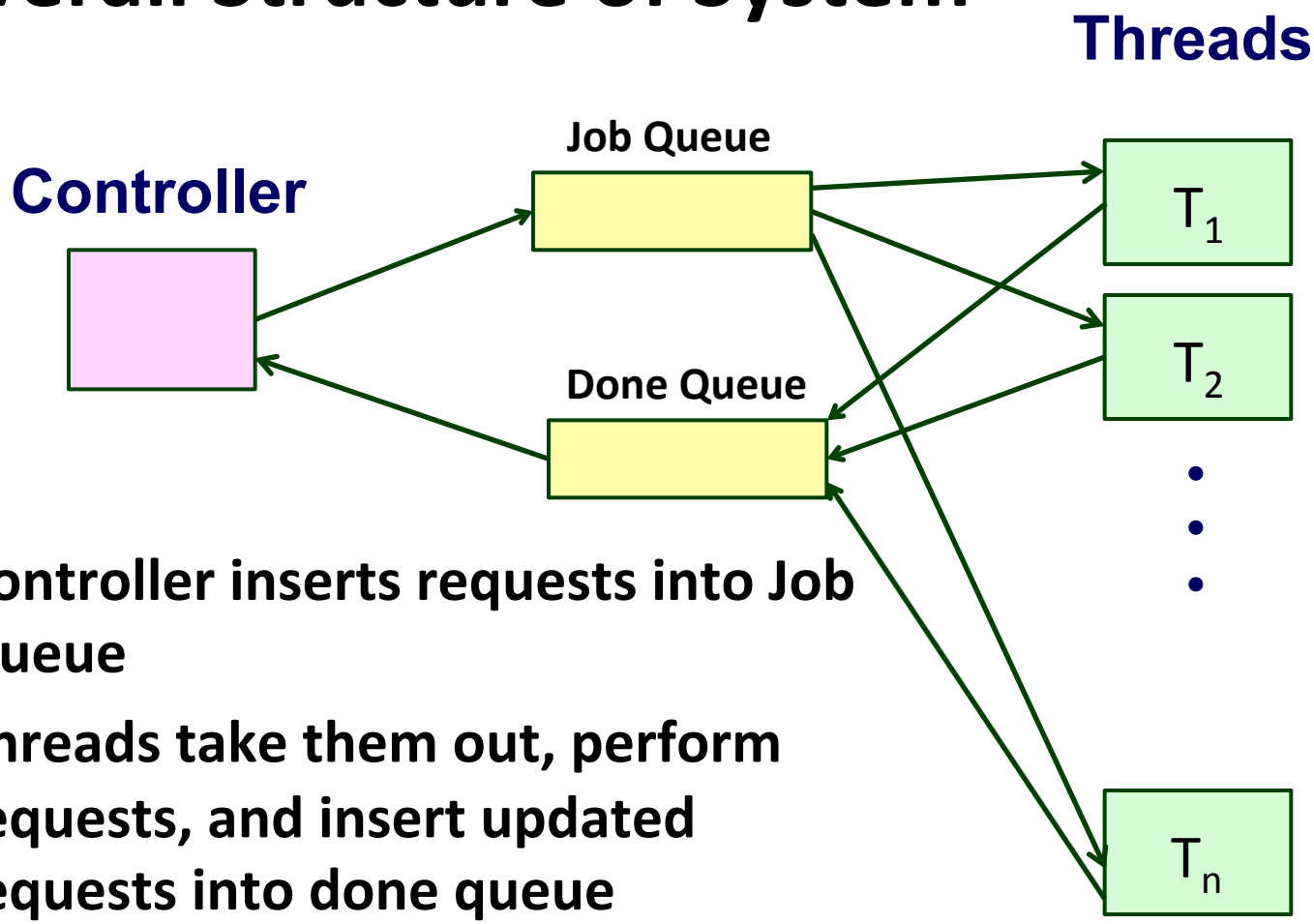
# Program Example

- **The benchmarking framework described earlier**

/afs/cs.cmu.edu/academic/class/15418-s17/recitations/recw8/recw8-code/asst4_bench

- **Elements similar to those required for assignment worker**
  - But, much simpler

# Overall Structure of System




- Controller inserts requests into Job Queue
- Threads take them out, perform requests, and insert updated requests into done queue
- Controller reads completed requests
- All synchronization through queues

# Request format (run.cpp)

```
// Record to track single job.
// Continues from creation to execution to completion
class Job {
    static int nextjid;  // Use for generating unique ids

 public:
    int benchmark;  // Which benchmark type
    int id;         // Unique ID for job
    int msecs;      // How many milliseconds did it require

    Job(int b) {
        id = nextjid++;
        benchmark = b;
        msecs = 0;
    }
};

int Job::nextjid;  // Initializes to 0
```

- Integer benchmark indicates job type
- msecs will be filled in by worker
- Variable nextjid part of class, not object
  - Use to generate unique IDs

# Queues

```
template <class T>
class WorkQueue {
private:
  ...
public:

  WorkQueue() {
    ...
  }

  T get_work() {
    ...
  }

  void put_work(const T& item) {
    ...
};
```

## Code Samples

```
// Maintain two job queues
typedef WorkQueue<Job> JobQueue;

// Jobs waiting to be executed
JobQueue *newJobQueue = NULL;
// Jobs that have completed
JobQueue *doneQueue = NULL;
```

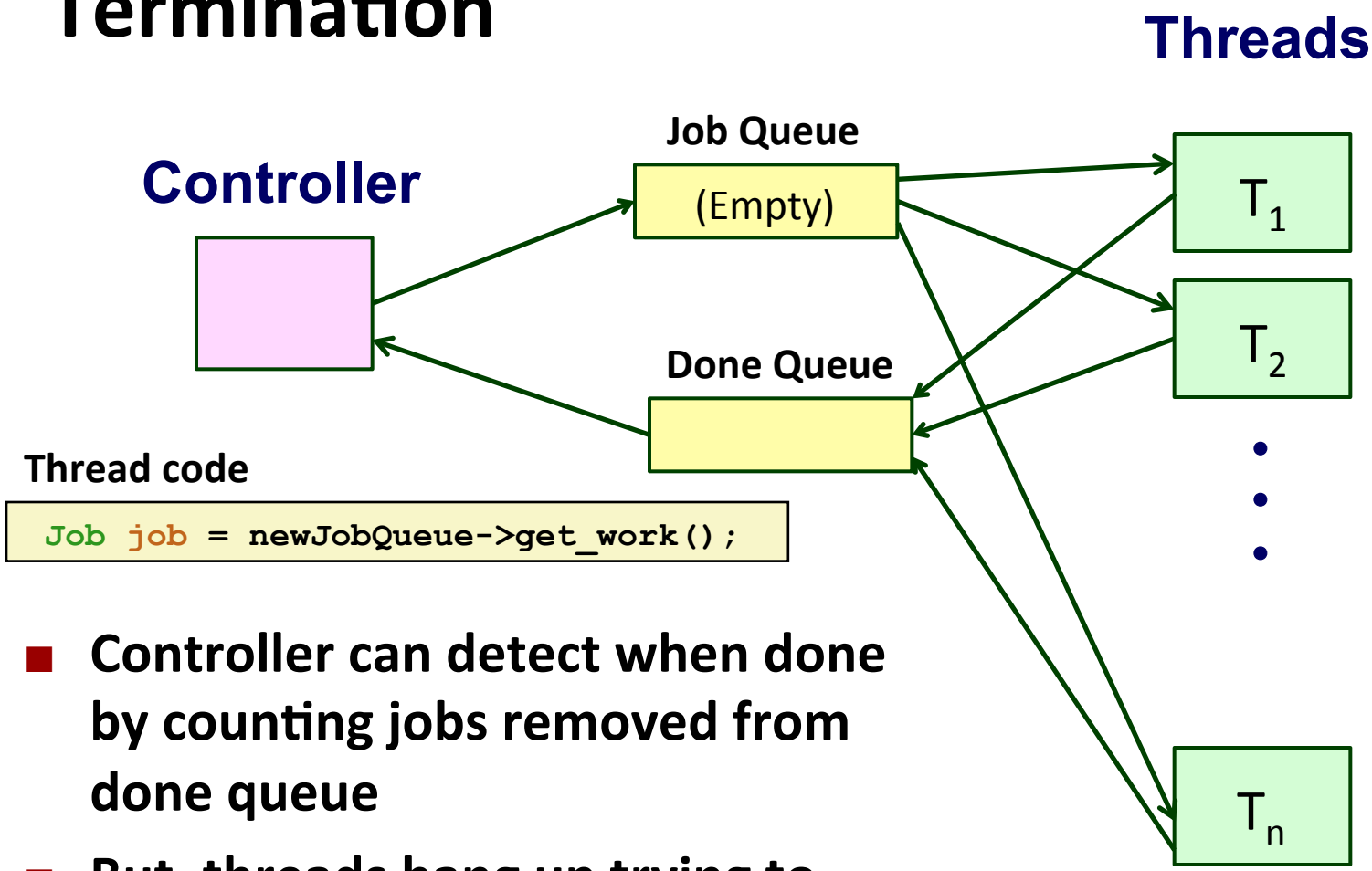
```
newJobQueue = new JobQueue;
doneQueue = new JobQueue;
```

```
Job job = Job(b);
newJobQueue->put_work(job);
```

```
Job job = doneQueue->get_work();
int b = job.benchmark;
report_job(b, job.msecs);
```

- Make use of code provided in work_queue.h
- Uses mutex and condition variable to ensure safe insert/remove

# Termination




- Controller can detect when done by counting jobs removed from done queue
- But, threads hang up trying to remove from job queue
- How can they be notified?

# Logging

- Google logging code included as part of assignment code

```
DLOG(INFO) << "Thread #" << tid << " working on request [" << request_tag
           << ":" << request_string << "]\n";
```

- Logging information accumulated in separate log file for each run
- Tip: Make up useful prefixes to allow extraction of interesting lines with grep

```
DLOG(INFO) << "TCK\tTIME " << mstate.num_seconds
           << ". Queued requests:\n";
```

# Some Advice

- **Lots of design thinking required**
  - Data structures used by master
  - How to manage jobs with different priorities
  - Implementing elasticity
  - Priority scheduler for worker
    - May need to extend provided queue code
    - But, it's tricky stuff
- **Expectations (Lines of code)**
  - master: 132 (provided) → 601 (My solution)
  - worker: 94 (provided) → 235 (My solution)
  - Provided code is mostly comments
- **Harsh reality**
  - The latedays machines will be horrendously overloaded as approach deadline