CS 111 Assignment 3:

Thread Dispatcher

Overall Task

- **The threads you've been using so far are implemented by Linux ("system threads")**
- **This project: use one system thread to implement any number of "user-level" threads**
- **Very similar to how an OS uses one core to run many system threads**

Thread Class

`Thread(std::function<void()> main)`

- Constructor: runs `main` as the top-level function in the thread

`~Thread()`

- Destructor: invoked only from `Thread::exit()`

`void schedule()`

- Add the associated thread to the back of the ready queue

`void Thread::redispatch()`

- Run a different thread; current thread will block if it hasn't been scheduled.

`void Thread::exit()`

- Terminate current thread

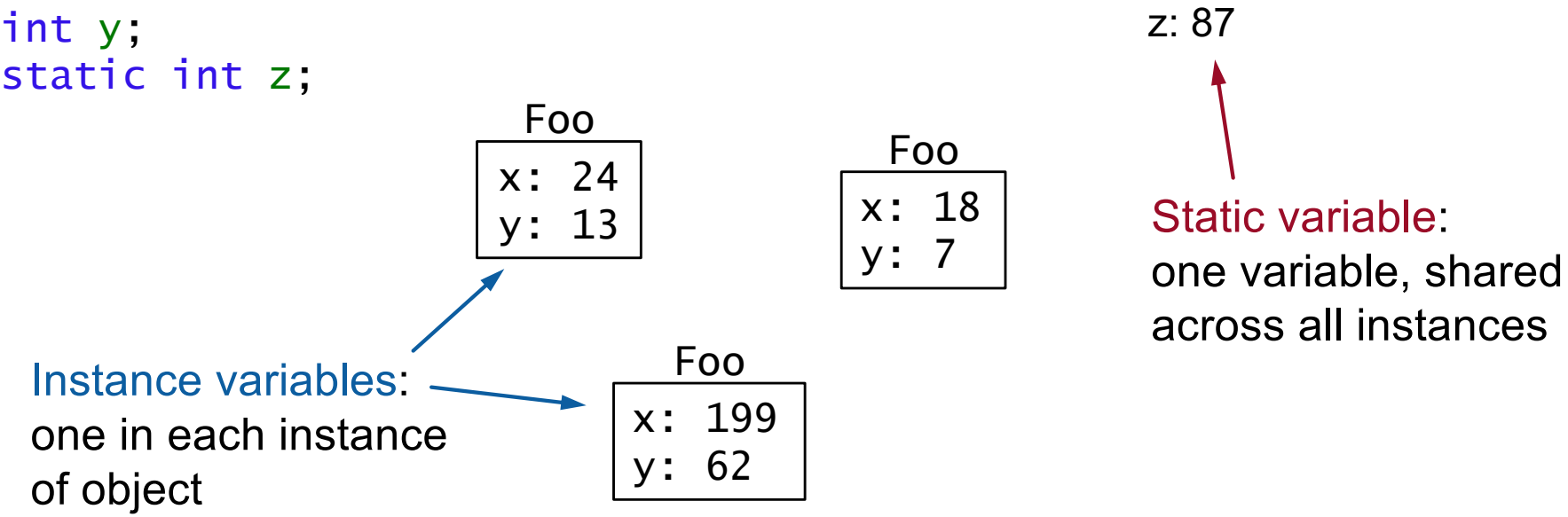
`void Thread::yield()`

- Invoke `schedule()` followed by `redispatch()`; allows other threads to run

`Thread* Thread::current()`

Class Static Variables

```
class Foo {mutex
    int x;
    int y;
    static int z;
}
```



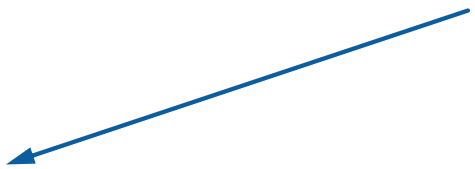
Class Static Methods

```
class Foo {class
public:
    method1(int x);
    static method2(char *s);
}

Foo f1;

f1.method1(14);

Foo::method2("xyzzy");
```



Normal method:

- Invoked on object instance
- Can access instance variables and static variables
- `this` variable refers to current object

Static method:

- Not associated with a particular instance
- No `this` variable accessible in method
- Can access static variables

Example: static.cc

```
class Demo {
public:
    Demo();
    ~Demo();
    static int num_live();
private:
    static int live_objects;
};

int Demo::live_objects = 0;

Demo::Demo() {
    live_objects++;
}

Demo::~Demo() {
    live_objects--;
}

int Demo::num_live() {
    return live_objects;
}
```

```
int main(int argc, char **argv)
{
    std::cout << "Initial number of live objects: "
            << Demo::num_live() << std::endl;

    Demo *d1 = new Demo();
    Demo *d2 = new Demo();
    Demo *d3 = new Demo();

    std::cout << "New number of live objects: "
            << Demo::num_live() << std::endl;

    delete d2;
    delete d3;

    std::cout << "Live objects after deleting 2: "
            << Demo::num_live() << std::endl;

    delete d1;
}
```

std::function (function.cc)

```
int add2(int value)
{
    int result = value + 2;
    std::cout << "Added 2, result "
            << result << std::endl;
    return result;
}

int main(int argc, char **argv)
{
    std::vector<std::function<int(int)>> funcs;

    funcs.emplace_back(add2);
    funcs.emplace_back([](int value) -> int {
        int result = value * 4;
        std::cout << "Multiplied by 4, result "
                << result << std::endl;
        return result;
    });
    funcs.emplace_back([](int value) -> int {
        int result = value - 1;
        std::cout << "Subtracted 1, result "
                << result << std::endl;
        return result;
    });
    funcs.emplace_back([](int value) -> int {
        int result = value / 2;
        std::cout << "Divided by 2, result "
                << result << std::endl;
        return result;
    });

    int value = 0;

    for (int i = 1; i < argc; i++) {
        if (i == 1) {
            value = std::stoi(argv[i]);
            std::cout << "Initial value: "
                    << value << std::endl;
        } else {
            value = funcs[std::stoi(argv[i])](
                    value);
        }
    }
}
```

Managing Stacks

- **Stack class created for you to use:**

```
Stack(void(*start)(Thread *), Thread *t));
void stack_switch(Stack *current, Stack *next);
```

- **Stack object holds:**
  - Space for call stack
  - Place to save stack pointer when stack isn't active
- **Constructor takes a function as argument**
  - This function will be invoked the first time the stack is activated via `stack_switch`
- **`stack_switch` does a context switch**
  - Save registers on current stack
  - Save sp in `current`
  - Load sp from `next`
  - Restore registers from new stack
  - Return in new context

Simple Stack Example: two_stacks.cc

```
Stack *stack1;
Stack *stack2;

void func1(Thread *t) {
    for (int i = 0; i < 10; i++) {
        std::cout << "func1: " << i << std::endl;
        if ((i == 2) || (i == 9)) {
            stack_switch(stack1, stack2);
        }
    }
    std::cout << "func1 calling exit" << std::endl;
    exit(0);
}

void func2(Thread *t) {
    for (int i = 0; i < 10; i++) {
        std::cout << "    func2: " << i << std::endl;
        if ((i == 3) || (i == 7)) {
            stack_switch(stack2, stack1);
        }
    }
    std::cout << "    func2 calling exit" << std::endl;
    exit(0);
}
```

```
int main()
{
    stack1 = new Stack(func1, nullptr);
    stack2 = new Stack(func2, nullptr);
    stack_switch(nullptr, stack1);
}
```

Preemption

```
void timer_init(uint64_t usec, std::function<void()> handler);
bool intr_enabled();
void intr_enable(bool on);
class IntrGuard;
```

- **Preemption requires interrupts**
- **`timer_init` causes timer handler to be called periodically**
- **For safety, need to disable interrupts when accessing data shared by multiple threads**
- **`IntrGuard` makes it easy to disable interrupts**
  - Creating an `IntrGuard` object saves current state, disables interrupts
  - Destroying the `IntrGuard` restores interrupts to original state
  - Analogous to `std::unique_lock`

Timer Example: interrupt.cc

```
std::atomic<int64_t> counter(0);

void timer_interrupt()
{
    std::cout << "Timer interrupt with counter "
            << counter << std::endl;
}

int main(int argc, char **argv)
{
    timer_init(500000, timer_interrupt);
    while (1) {
        counter++;
    }
}
```

Disabling Interrupts: interrupt2.cc

```
void timer_interrupt()
{
    std::cout << "Timer interrupt" << std::endl;
}

int main(int argc, char **argv)
{
    std::string line;

    timer_init(500000, timer_interrupt);
    while (std::getline(std::cin, line)) {
        if (intr_enabled()) {
            intr_enable(false);
            std::cout << "Disabled interrupts" << std::endl;
        } else {
            intr_enable(true);
            std::cout << "Enabled interrupts" << std::endl;
        }
    }
}
```

Disabling Interrupts: interrupt3.cc

```
void timer_interrupt()
{
    static int i = 0;
    std::atomic<int64_t> spins;

    std::cout << "Timer interrupt with counter " << counter << std::endl;
    i++;
    if (i == 3) {
        for (spins = 0; spins < 500'000'000; spins++) {
            /* Empty loop body; just waste a lot of time.*/
        }
        i = 0;
    }
}

int main(int argc, char **argv)
{
    timer_init(500000, timer_interrupt);
    while (1) {
        counter++;
    }
}
```