Multi-threaded Concurrency

Chris Brown

In This Module ...

Thread concepts

The pthreads API

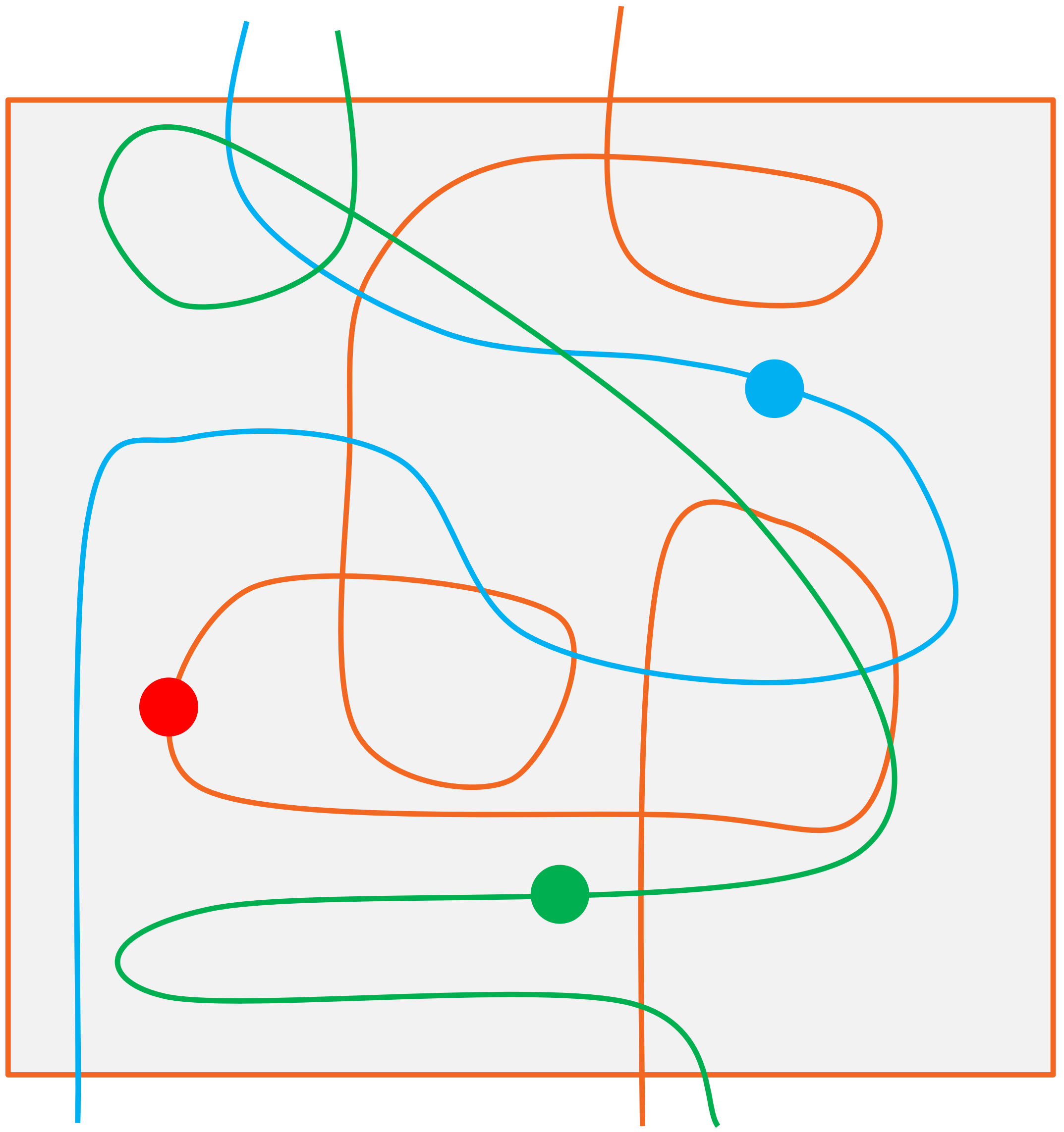
Multi-threaded servers and clients

Processor farms

Demonstration: Counting primes

Writing "thread-safe" code

Multi-threading Illustrated



Reasons for Multi-threading

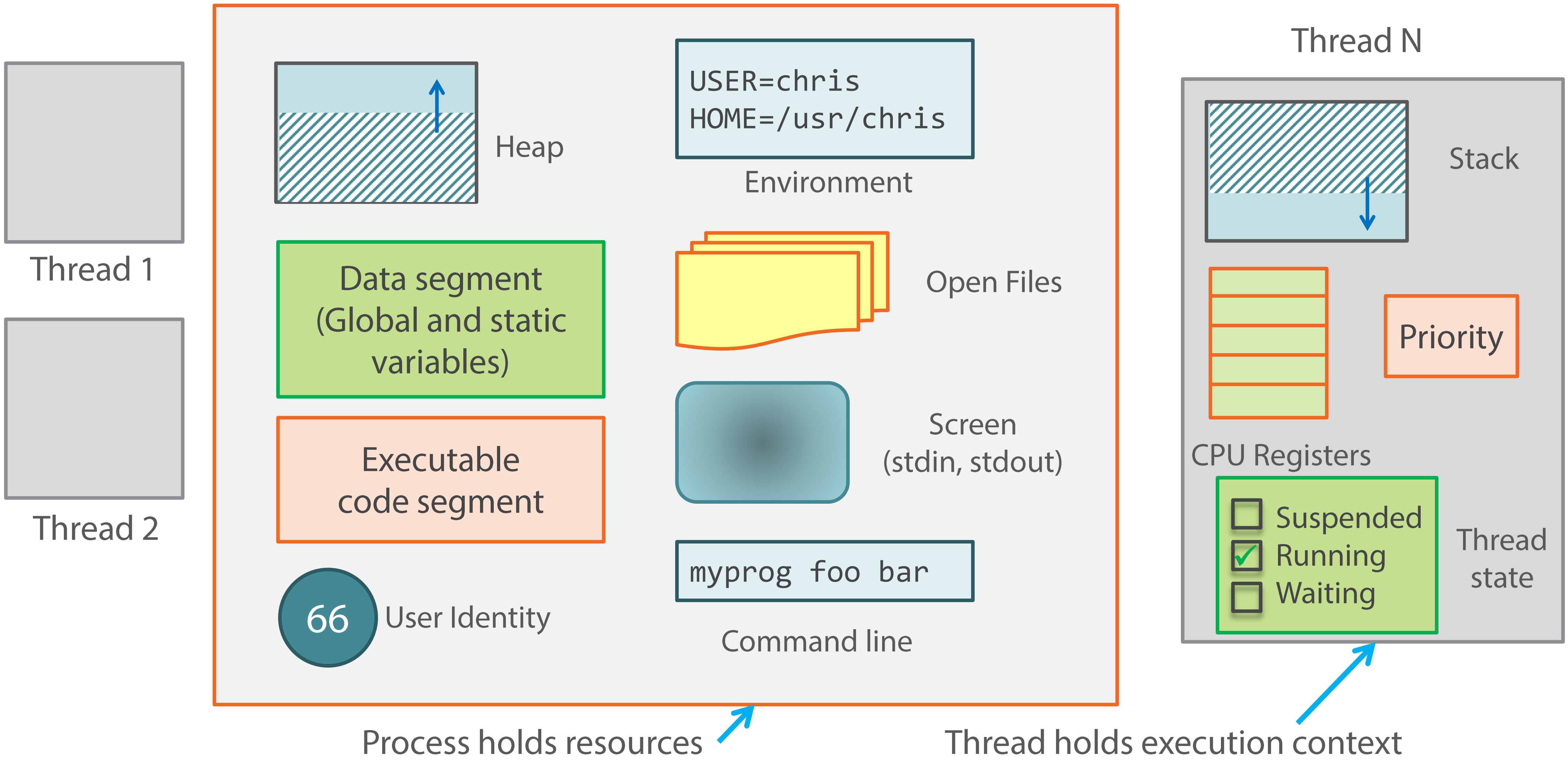
Expressing logical concurrency

Implementing background tasks

Concurrent servers

Exploiting multi-processor hardware

Threads and Processes



Shared and Not Shared

Threads share:

- Code
- Global and static variables
- Open file descriptors

Threads do not share:

- Variables local to functions

Pthreads

IEEE

POSIX 1003.1c

A standardised set of C library routines for thread creation and management

— Upwards of 60 functions

Thread Creation

Thread Attribute Object
(NULL for defaults)

```
pthread_create(&handle, &attr, func, arg)
```

Returns 0 of OK,
Nonzero if error

Thread handle
returned here

```
void *func(void *arg)
{
    //Thread function
}
```

Thread Termination

- A thread can terminate in several ways:
    1. By calling `pthread_exit(exit_status)`
    2. By returning from its top-level function
    3. By some other thread sending a cancellation request:

    ```
    pthread_cancel(handle);
    ```
- Parent can wait for thread to finish and get exit status:
    - `pthread_join(handle, &exit_status);`
- Parent should detach thread if they don't need to join on it:

    ```
    pthread_detach(handle);
    ```

Thread Life Cycle

Parent Thread

Child Thread

```
pthread_create(..., myfunc, ...);
```

```
myfunc()
{
// Normal life of thread

pthread_exit(status);
}
```

```
pthread_join(handle, &status);
```

Demonstration

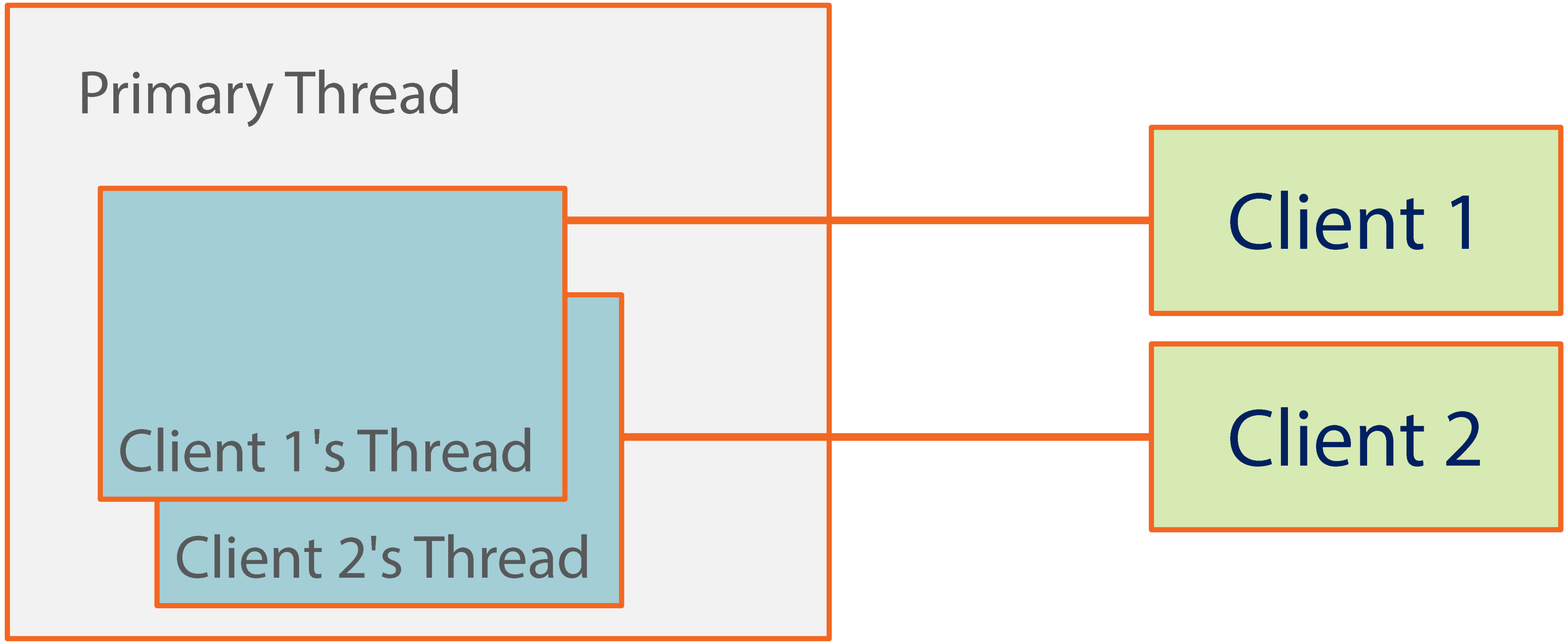
Simple thread example

Thread Example

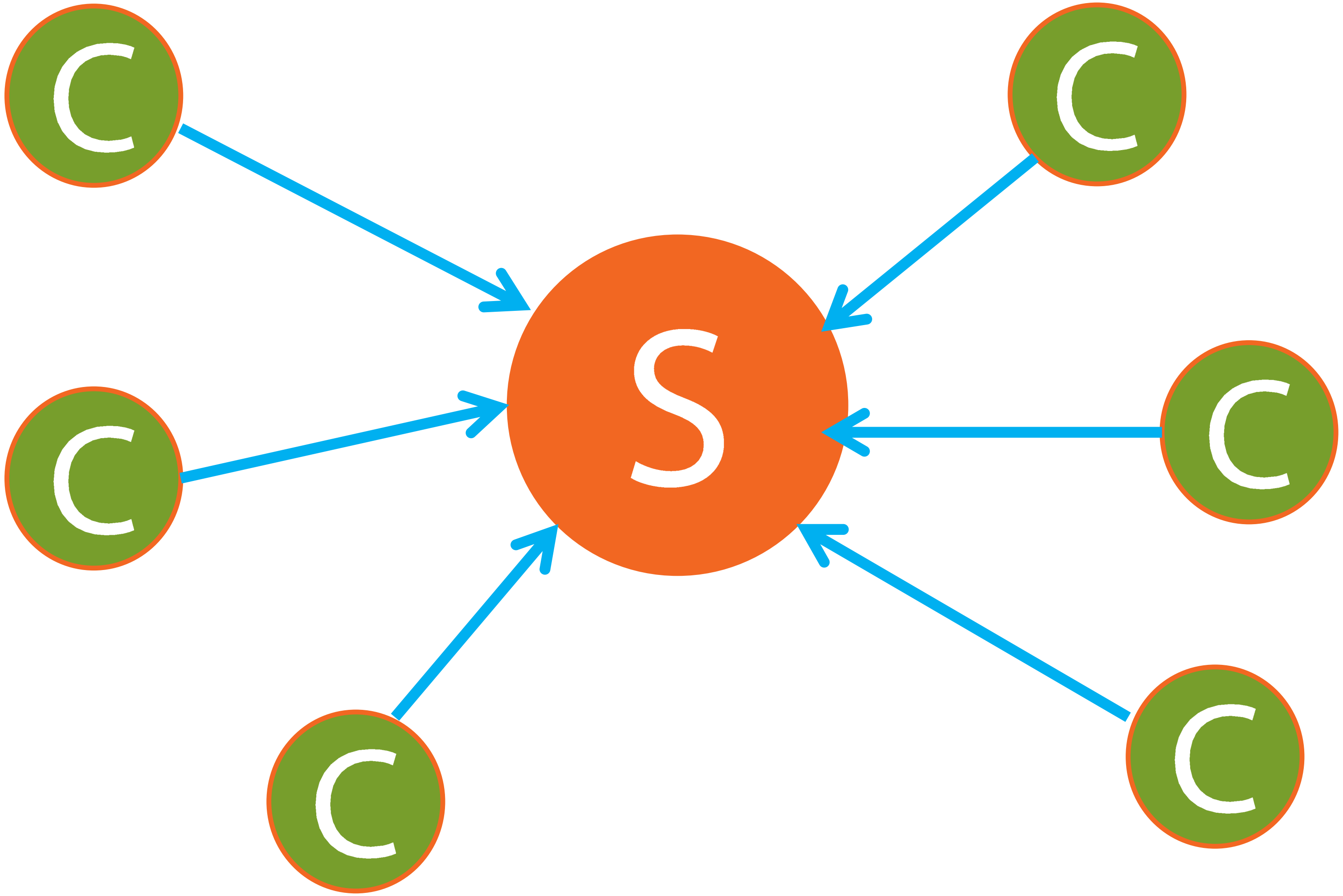
```
#include <pthread.h>
#include <stdio.h>

void *func(void *arg)
{
    printf("child thread says %s\n", (char *)arg);
    pthread_exit((void *)99);
}

int main()
{
    pthread_t handle;
    int exitcode;

    pthread_create(&handle, NULL, func, "hi!");
    printf("primary thread says hello\n");
    pthread_join(handle, (void **)&exitcode);
    printf("exit code %d\n", exitcode);
}
```

Multi-threaded Server

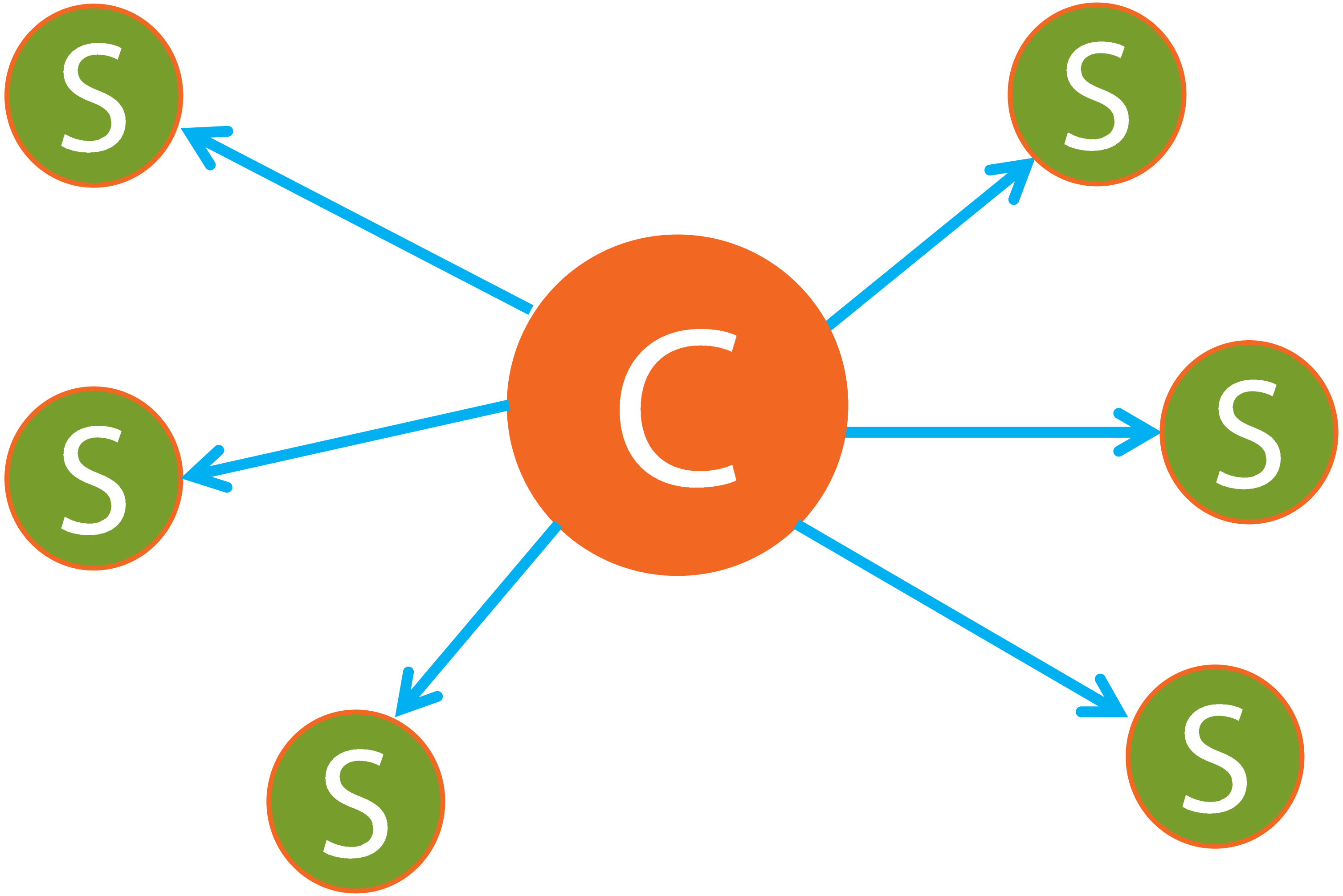
- Thread-per-client is an elegant model for concurrent servers
  - Efficient
  - Easy to keep per-client state (local variables)
  - Easy to share state (global variables)



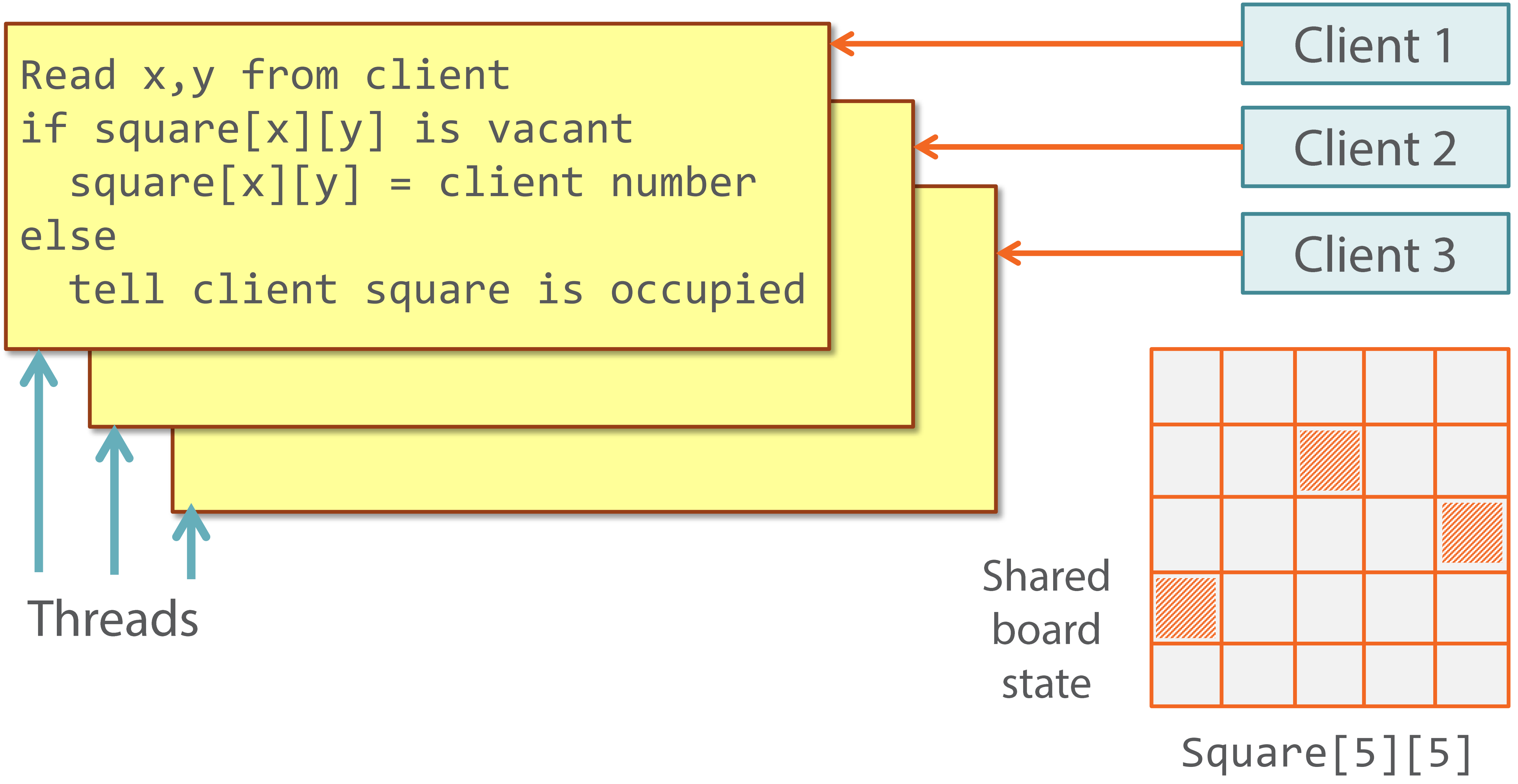
Concurrent Server



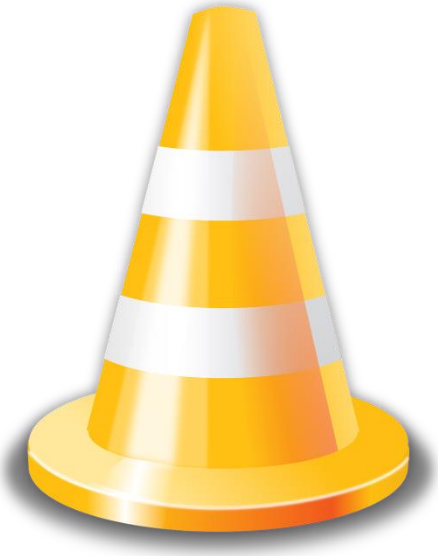
Processor Farm



Sharing Client State



Thread Safety



Problems can arise when multiple asynchronous threads access shared data

Mutual exclusion locks ("mutexes") control entry to code sections that update or access shared state

"Thread-safe" code

Library functions you call must also be thread-safe

Pthread Mutexes

```
#include <pthread.h>

static int shared_data;
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;

func( ... )
{
        pthread_mutex_lock(&mylock);
        // Update or access shared_data here
        pthread_mutex_unlock(&mylock);
}
```

Demonstration

Why do we need mutexes?

Processor Farms

A common model for decomposing computationally-intensive tasks

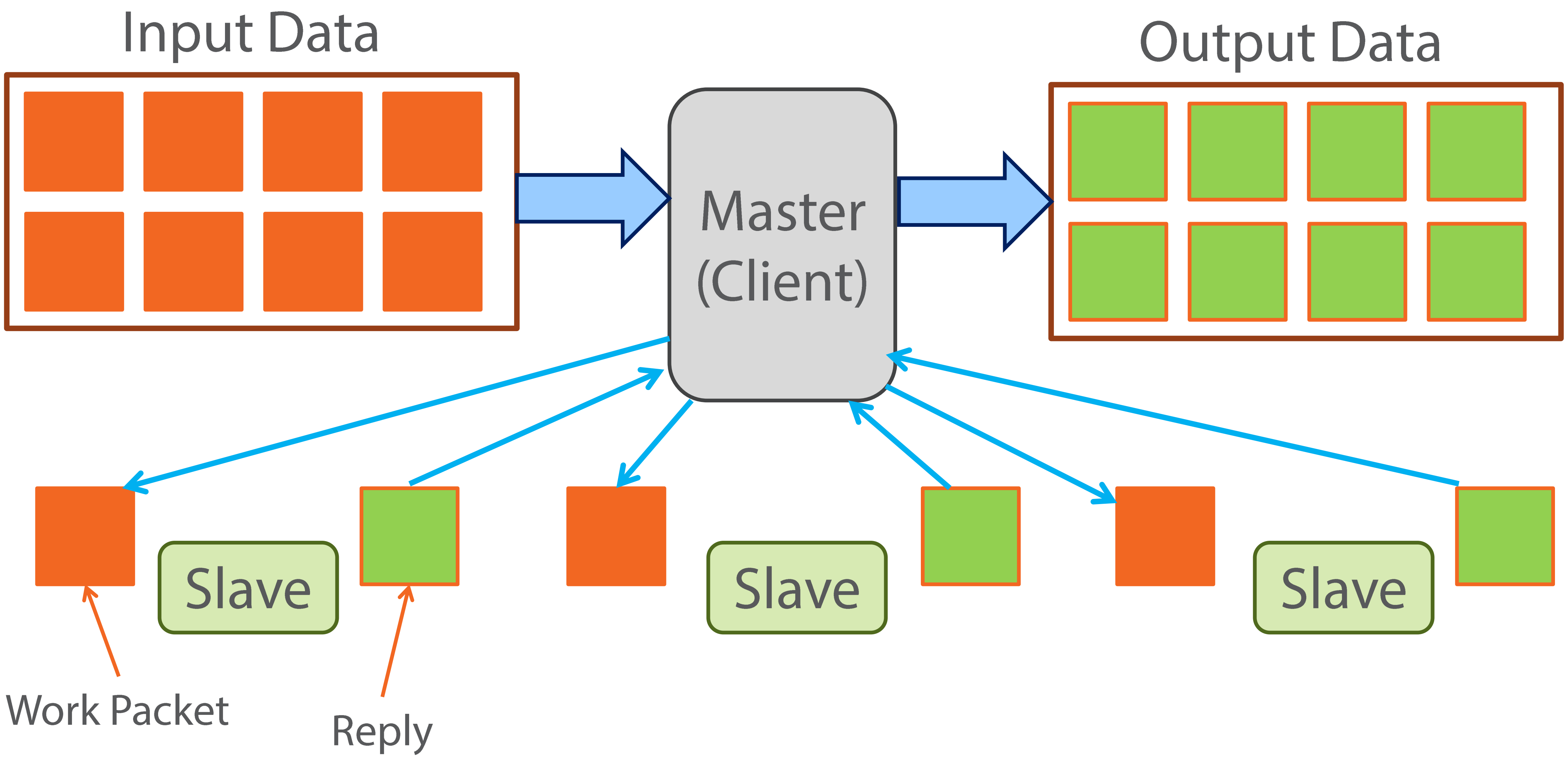
Input data set broken down into "work packets"
— each processed independently of the others

Need ≈ 10x as many work packets as processors

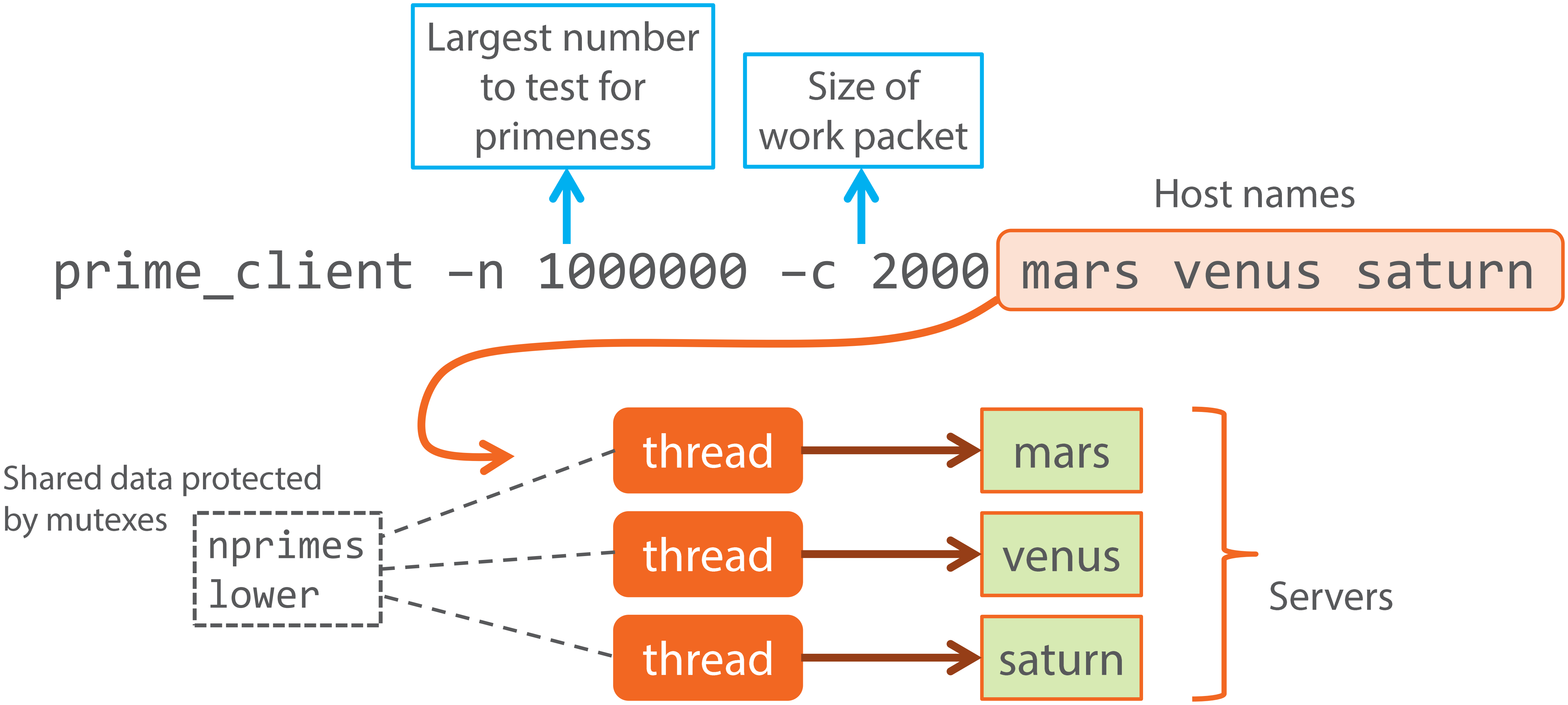
— Automatic load balancing

Goal: Linear speedup

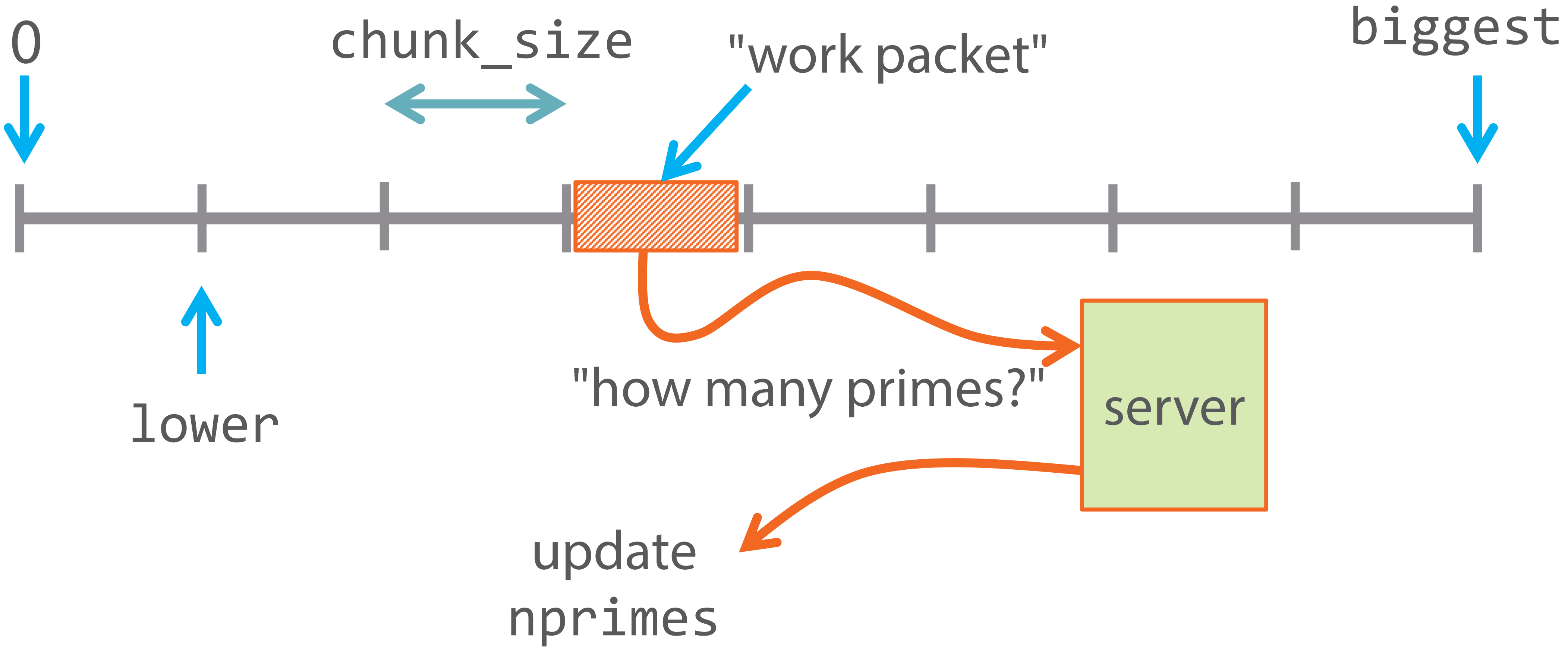
Processor Farm Illustrated



A Processor Farm to Count Prime Numbers



Dividing up the Work



Demonstration

Processor Farm

Module Summary

In this module:

- Threads compared to processes
- The pthreads API
- Thread-safe code: accessing shared data
- The processor farm model
- A processor farm to count prime numbers

Course Summary

Congratulations!

In this course:

- The characteristics of TCP and UDP protocols
- Writing TCP-based servers
- Writing TCP-based clients
- UDP servers and clients
- Writing concurrent servers
- Writing concurrent clients using threads