

# Project 2A

CS111 - Dis 1A

02/08/19

# What is this project about?

Update a shared variable

- A simple integer in part 1
- A list in part 2

To demonstrate the existence of race conditions  
...and how to avoid them

# What is this project about?

Update a shared variable

- A simple integer in part 1
- A list in part 2

To demonstrate the existence of race conditions  
...and how to avoid them

```
void add(long long *pointer, long long value) {  
    long long sum = *pointer + value;  
    *pointer = sum;  
}
```

# Multi threaded applications

Initially, your main() program comprises a single thread

- All other threads will have to be explicitly created
  - Done using `pthread_create()`
- Upon exit, wait on all other threads to complete
  - Using `pthread_join`
- Time each run for each thread
  - Using `clock_gettime()`
- Optionally providing protection from race conditions
  - Mutexes, spin locks, compare and swap adds
- Optionally creating conflicts
  - When adding, or when modifying the linked list

# Measuring run times

Success: 0 // Error: errno

**int clock\_gettime(clockid\_t *clk\_id*, struct timespec \**tp*);**

The specified clock:

- **CLOCK\_REALTIME**
- **CLOCK\_MONOTONIC**
- **CLOCK\_PROCESS\_CPUTIME\_ID**
- **CLOCK\_THREAD\_CPUTIME\_ID**

```
struct timespec {  
    time_t    tv_sec;        /* seconds */  
    long      tv_nsec;       /* nanoseconds */  
};
```

Rmk: A clock may be system wide or per-process/per thread. Which do we want?

# Measuring run times

Success: 0 // Error: errno

**int clock\_gettime(clockid\_t *clk\_id*, struct timespec \**tp*);**

The specified clock:

- **CLOCK\_REALTIME**
- **CLOCK\_MONOTONIC**
- **CLOCK\_PROCESS\_CPUTIME\_ID**
- **CLOCK\_THREAD\_CPUTIME\_ID**

```
struct timespec {  
    time_t    tv_sec;        /* seconds */  
    long      tv_nsec;       /* nanoseconds */  
};
```

Rmk: A clock may be system wide or per-process/per thread. Which do we want?

-> A system wide clock

# pthread\_create

Goal: Create a thread

Success: 0 // Error : errno, \*thread left undefined

Opaque identifier returned by the routine

Attribute object, set to 'NULL' for default values. These include:

- Detached or joinable state
- Scheduling inheritance
- Scheduling policy
- Scheduling parameters
- Scheduling contention scope
- Stack size
- Stack address
- Stack guard (overflow) size

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start_routine) (void *), void *arg);
```

The C routine to be executed on creation

A single argument to be passed to start routine:  
Passed by reference as a pointer cast of type void.  
Can be set to "NULL" .

# pthread\_join()

Goal: Join with a terminated thread

Success: 0 // Error: errno

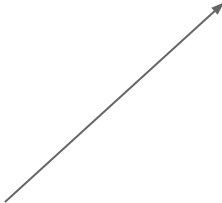


```
int pthread_join(pthread_t thread, void **retval)
```

Thread we're waiting on to terminate



Return value of thread we're waiting on:

- Can be set to 'NULL'
  - If the target thread is cancelled, **PTHREAD\_CANCELED** is placed in
- 



# Initializing a mutex

2 ways:

- Statically , when declared :

```
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
```

- Dynamically, which permits to set attributes:

Success: 0 // Error : errno

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,  
const pthread_mutexattr_t *restrict attr);
```

Mutex object

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Rmk: Attempting to destroy a locked mutex results in a 'busy' error

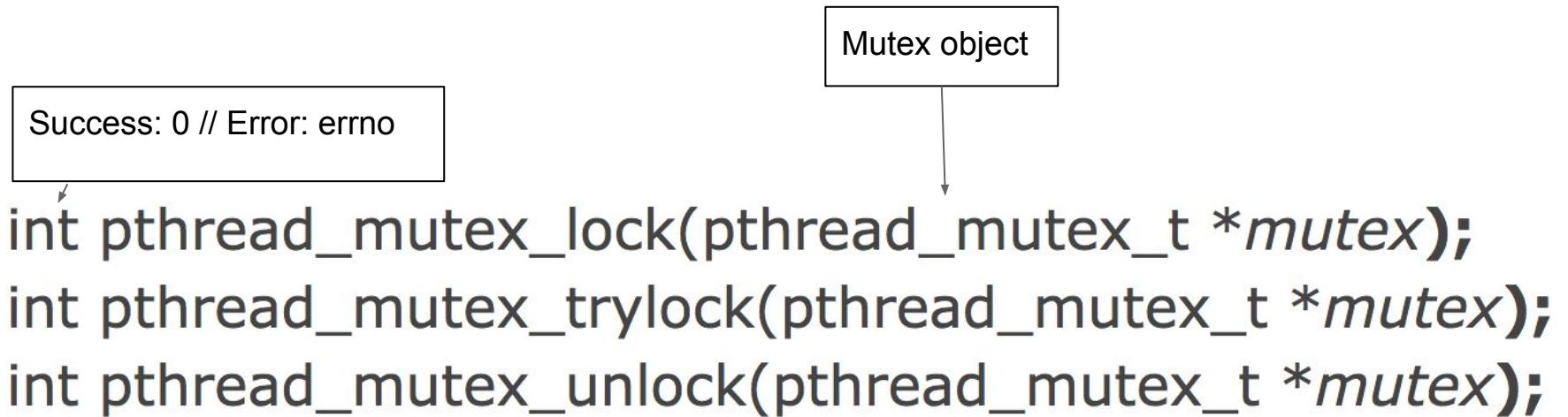
# Setting attributes

Mutex attributes include:

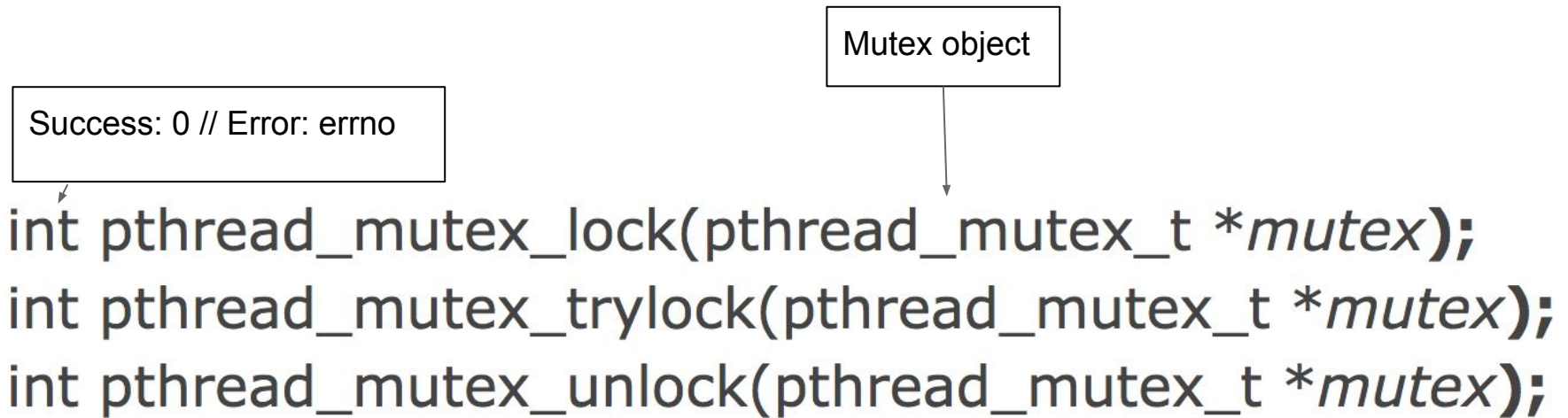
- The type (deadlocking, deadlock-detecting, recursive...)
- The robustness (if you acquire a mutex and original owner died while processing it)
- Process-shared attributes (sharing a mutex across process boundaries)
- Protocol (how a thread behaves when higher priority thread wants the mutex)
- Priority ceiling (of the critical section, can prevent priority inversion)

Rmk: you need to call init/destroy to create the 'attributes' object

# Locking and Unlocking a Mutex



# Locking and Unlocking a Mutex



- Lock: locks object referenced by mutex if available, otherwise blocks until it becomes available
  - Returns with the mutex in a locked state with the calling thread as its owner
- Trylock : like lock, but returns immediately if mutex is already locked
  - Returns a 'busy' error code
- Unlock : release object referenced by mutex
  - Scheduling policy determines which thread will acquire the mutex

# Sync lock test and set

Goal: Implement a spin lock

# Sync lock test and set

Goal: Implement a spin lock

Writes 'value' into \*ptr, and returns the previous contents of \*ptr

*type* `__sync_lock_test_and_set` (*type* \*ptr, *type* value, ...)

`__sync_lock_release` (*type* \*ptr, ...)

Releases lock pointed at by ptr

# Compare and swap

Goal: Atomic compare and swap, if the current value of `*ptr` is `oldval`, then write `newval` into `*ptr`

# Compare and swap

Goal: Atomic compare and swap, if the current value of *\*ptr* is *oldval*, then write *newval* into *\*ptr*

True if comparison is successful and *newval* is written

```
bool __sync_bool_compare_and_swap (type *ptr, type oldval type newval, ...)
type __sync_val_compare_and_swap (type *ptr, type oldval type newval, ...)
```

Contents of *\*ptr* before the operation

Variable to modify



# 2.1.1

## **QUESTION 2.1.1 - causing conflicts:**

**Why does it take many iterations before errors are seen?**

**Why does a significantly smaller number of iterations so seldom fail?**

# 2.1.2

## **QUESTION 2.1.2 - cost of yielding:**

**Why are the --yield runs so much slower?**

**Where is the additional time going?**

**Is it possible to get valid per-operation timings if we are using the --yield option?**

**If so, explain how. If not, explain why not.**

## 2.1.3

Why does the average cost per operation drop with increasing operations?

If the cost per iteration is a function of the number of iterations, how do we know how many iterations to run (or what the 'correct' cost is)?

## 2.1.4

### **QUESTION 2.1.4 - costs of serialization:**

**Why do all of the options perform similarly for low numbers of threads?**

**Why do the three protected operations slow down as the number of threads rises?**

## 2.2.1

Compare the variation in time per mutex-protected operation vs number of threads in Part1 and Part2.

Comment on general shapes of the curves, and explain why they have this shape.

Comment on the relative rates of increase and differences in the shape of the curves, and offer an explanation of these differences.

## 2.2.2

Compare the variation time per protected operation vs the number of threads for list operations protected by Mutex vs Spin locks. Comment on the general shapes of the curves, and explain why they have this shape.

Comment on the relative rates of increase and differences in the shapes of the curves, and offer an explanation for these differences.