

Project 2A

CS111 - Dis 1A

05/03/19

A high level picture

- Multi-threaded applications are ubiquitous
 - You should learn how to program one
 - A new problem arises : synchronization
- If done poorly, things can go seriously wrong
 - Race conditions can make the output of your program unpredictable
- As a demonstration, you will modify a shared variable
 - An integer in part 1 (initialized to 0)
 - A doubly-linked list in part 2 (initially empty)

What is this project about?

- In part 1, each thread will add 1, then subtract 1
- In part 2, each thread will insert a node, then delete a node
- For both parts, the user can specify how many times a single thread executes these operations through the 'iterations' flag
- Regardless, we expect:
 - A 0 value after all threads have run
 - An empty list in part 2

Are threads like process children?

- How is this situation different from the shell you spawned in project 1A?
- Why are we talking about race conditions now?
- How will a race condition manifest itself using:

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

The main steps

- (The logic is the same for part 1 and 2)
- How will you notice that a race condition occurred?

The main steps

- (The logic is the same for part 1 and 2). For part 1:
- How will you notice that a race condition occurred?
 - The final value of the shared variable will be nonzero
- Now that you witnessed a race condition, you will be asked to create some
 - How will you modify the add to ensure that things go wrong?

The main steps

- (The logic is the same for part 1 and 2). For part 1:
- How will you notice that a race condition occurred?
 - The final value of the shared variable will be nonzero
- Now that you witnessed a race condition, you will be asked to create as many as possible
 - How will you modify the add() to ensure that things go wrong?
 - Force the thread to yield at the worst possible time
 - Your final value should be further away from 0
- The final step is to counter the race conditions you created, using 3 different mechanisms:
 - Compare-and-Swap (CAS) add
 - Spin-locks
 - Mutexes

Monitoring performance

- You are using 3 methods to do the same thing
- How does their performance compare?
 - Do they all work equally well in avoiding race conditions?
 - Do they all guarantee that the final counter is set to 0?
 - Is there an impact on time/operation?
 - You have to evaluate time/operation
 - Is there an impact on CPU usage?
 - More on that in part 2B ...

Multi threaded applications : API

Initially, your main() 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, CAS adds
- Optionally creating conflicts
 - When adding, or when modifying the linked list

All of this is done with variations of the number of threads and iterations/thread : a lot of output!

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, to be able to compare all threads to the same reference

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_create

Goal: Create a thread

Note that unlike `fork()`, you are giving the thread a routine to run on creation - as if `fork()` and `exec()` were combined. The reason that `fork()` and `exec()` are split is to allow for redirection before running a command, this is useless for a thread.

```
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: Wait for a thread to exit

Success: 0 // Error: errno

A box containing the text "Success: 0 // Error: errno" with an arrow pointing to the first parameter of the pthread_join() function signature.

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

Two boxes with arrows pointing to the parameters of the pthread_join() function. One box points to the 'thread' parameter, and another points to the '**retval' parameter.

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:

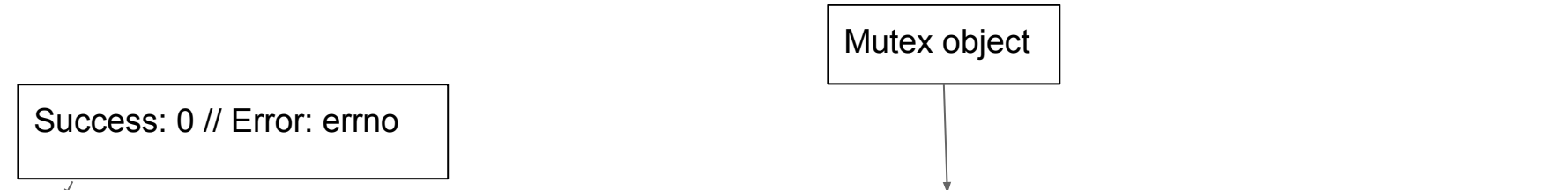
- The type (deadlocking, deadlock-detecting, recursive...)
- The robustness (if you acquire a mutex and original owner died while processing it)
- Process-shared attributes (sharing a mutex across process boundaries)
- Protocol (how a thread behaves when higher priority thread wants the mutex)
- Priority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

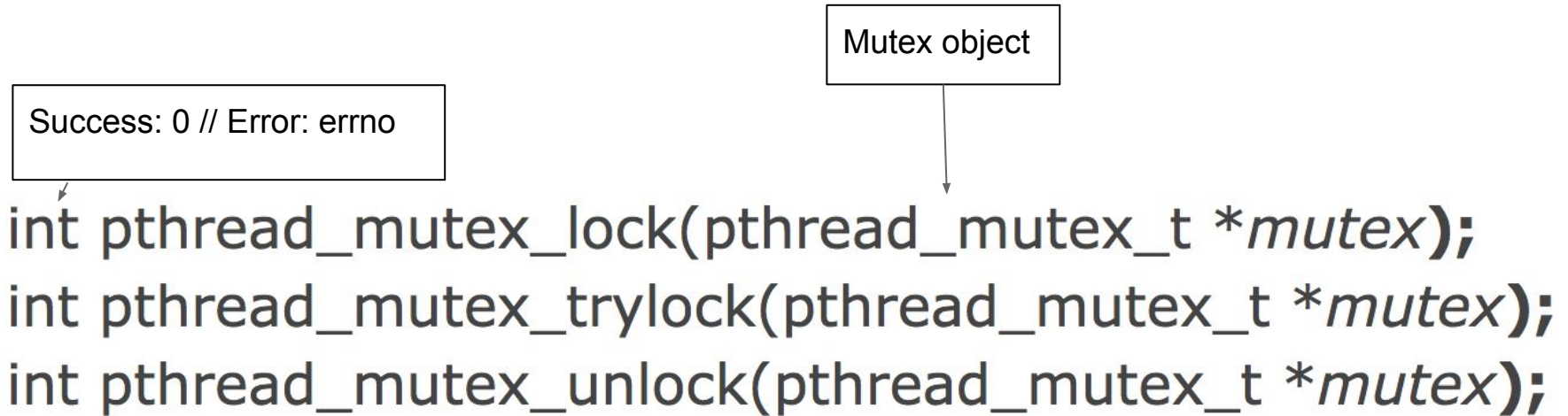
Success: 0 // Error: errno

Mutex object



```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_trylock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *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

How would you use this function to implement a spin lock?

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.