SEMINAR 15

POSIX THREADS. SYNCRONIZATION

THREAD IS THE SMALLEST INDEPENDENTLY MANAGED SEQUENCE OF INSTRUCTIONS

THREADS SHARE ADDRESS SPACE AND FILE DESCRIPTORS

EACH THREAD
HAS ITS OWN STACK
WITH GUARD PAGE

EACH PROCESS HAS AT LEAST ONE THREAD LAUNCHED WITH _START()

UNLIKE PROCESSES.

THREADS DO NOT FORM

HIERARCHY

POSIX THREADS API REQUIRES -PTHREAD FLAG DURING LINKING

UNLIKE MOST POSIX FUNCTIONS.

PTHREAD FUNCTIONS

DON'T USE

ERRNO

WHY USE THREADS?

- MORE LIGHTWEIGHT
- INTER-THREAD COMMUNICATION IS EASIER AND FASTER
- SHARED RESOURCES AND MEMORY SPACE

LET'S HAVE A LOOK AT API!

CREATE AND LAUNCH A THREAD

```
int pthread_create(
  pthread_t *restrict thread, // Handle of thread created
  const pthread_attr_t *restrict attr, // Attributes of new thread
  (void*)(*function)(void*), // Routine to launch
  void *arg // Argument to pass to routine
);
```

NOTE THAT PTHREAD_ATTR_T IS PLATFORM-DEPENDENT! IMPLEMENTATION IS NOT DEFINED IN POSIX

THREAD ATTRIBTUTES

```
int pthread_attr_init(pthread_attr_t *attr); // Constructor
int pthread_attr_destroy(pthread_attr_t *attr); // Destructor
```

THE PARAMETERS ARE MANIPULATED VIA SPECIAL FUNCTIONS

```
int pthread_attr_setstacksize(...); // No less than PTHREAD_STACK_MIN
int pthread_attr_setstackaddr(...);
int pthread_attr_setguardsize(...); // Could be 0
```

THREAD TERMINATION

```
noreturn void pthread_exit(void *retval);

// or just return from thread routine
```

FORCIBLY TERMINATE A THREAD

```
int pthread_cancel(pthread_t thread);
```

WAIT FOR A THREAD TO FINISH

```
int pthread_join(pthread_t thread
                 void **retval); // retval may be NULL;
                                 // if thread was cancelled
                                 // PTHREAD_CANCELED will be placed
```

THINGS GET COMPLICATED
WHEN SHARED RESOURCES ARE INVOLVED

WE NEED SYNCRONIZATION PRIMITIVES

CRITICAL SECTION

IS A PART OF PROGRAM

THAT IMPLIES USE MONOPOLY

WE CAN USE MUTEXES
TO CREATE SUCH SECTIONS

MUTEX HAS TWO STATES:
LOCK OR UNLOCKED

WORKING WITH MUTEXES

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

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

SOMETIMES LOCKS CAN DEGRADE PERFORMANCE

ATOMIC TYPE IS
DATA RACE FREE

C11 INTRODUCED

_ATOMIC QUALIFIER

STDATOMIC.H FEATURES

ALIASES FOR SUPPORTED TYPES

ATOMIC VALUE MANIPULATION

```
void atomic_store(T* object, T value);
T atomic_load(T* object);
T atomic_exchange(T* object, T new_value);
T atomic_compare_exchange_strong(T* object, T* expected, T new_value);
T atomic_compare_exchange_weak(T* object, T* expected, T new_value);
T atomic_fetch_MOD(T* object, T operand); // add, sub, and, or, xor
```

ALL OF THE FUNCTIONS FEATURE _EXPLICIT VERSION THAT ALLOWS TO SPECIFY MEMORY ORDER

MEMORY ORDERS

- MEMORY_ORDER_RELAXED
- MEMORY_ORDER_CONSUME
- MEMORY_ORDER_ACQUIRE
- MEMORY_ORDER_RELEASE
- MEMORY_ORDER_ACQ_REL
- MEMORY_ORDER_SEQ_CST

SOMETIMES. USING ATOMICS
WE CAN CREATE DATA STRUCTURES
THAT CAN BE ACCESSED CONCURRENTLLY
AND DO NOT REQUIRE LOCKING.
THEY ARE CALLED LOCK-FREE