Interlude: Thread API

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Thread Creation

- **Thread creation interface in POSIX**

```
#include <pthread.h>
int
pthread_create(pthread_t       *thread,
               const pthread_attr_t *attr,
               void            *(*start_routine)(void*),
               void            *arg);
```

Argument	Meaning
pthread_t *thread	To interact with this thread
const pthread_attr_t *attr	To specify any attributes this thread might have (e.g. stack size)
void *(*start_routine)(void *)	To specify which function should this thread start running in (**void pointer type** allow us to pass in any type of argument/result)
void *arg	To specify the argument to be passed to the function

Interface with different data type for argument and return value of the function is available

```
int pthread_create(..., // first two args are the same
                   void *(*start_routine)(int),
                   int  arg);
```

```
int pthread_create(..., // first two args are the same
                   int (*start_routine)(void *),
                   void *arg);
```

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

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Thread Completion

- **Interface to wait for thread completion**

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Argument	Meaning
pthread_t thread	To specify which thread to wait for
void **value_ptr	To specify a pointer to return value you expect to get back

  - Note that not all code that is multi-threaded uses the join routine

```
typedef struct { int a; int b; } myarg_t;
typedef struct { int x; int y; } myret_t;

void *mythread(void *arg) {
    myret_t *rvals = Malloc(sizeof(myret_t));
    rvals->x = 1;
    rvals->y = 2;
    return (void *) rvals;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myret_t *rvals;
    myarg_t args = { 10, 20 };
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &rvals);
    printf("returned %d %d\n", rvals->x, rvals->y);
    free(rvals);
    return 0;
}
```

```
void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    myret_t oops; // ALLOCATED ON STACK: BAD!
    oops.x = 1;
    oops.y = 2;
    return (void *) &oops;
}
```

oops will be automatically de-allocated after return

```
void *mythread(void *arg) {
    long long int value = (long long int) arg;
    printf("%lld\n", value);
    return (void *) (value + 1);
}

int main(int argc, char *argv[]) {
    pthread_t p;
    long long int rvalue;
    Pthread_create(&p, NULL, mythread, (void *) 100);
    Pthread_join(p, (void **) &rvalue);
    printf("returned %lld\n", rvalue);
    return 0;
}
```

passing in a single value

Locks

- **POSIX provides mutual exclusion to a critical section via locks**

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

  - If a code has a critical section, the thread needs to acquire the lock to enter

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

  - There are two ways to initialize locks (static & dynamic)

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

  - A wrapper that checks the error code can be used

```
// Keeps code clean; only use if exit() OK upon failure
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

  - There are two more calls to acquire the lock

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                            struct timespec *abs_timeout);
```

`pthread_mutex_trylock`: return failure if the lock is already held
`pthread_mutex_timedlock`: return after a timeout or after acquiring the lock

Condition Variables

- **Condition variables are useful when some kind of signaling must take place between threads**
  - One has to in addition have a lock associated with this condition

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

`pthread_cond_wait`: puts the calling thread to sleep and waits for other thread to signal it
`pthread_cond_signal`: unblocks at least one thread that is blocked on condition variable

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

  - **`pthread_cond_wait()`** take a lock as argument since it releases the lock when sleep, otherwise, the other thread can't acquire lock and signal it
  - Before returning being woken, **`pthread_cond_wait()`** re-acquires the lock
  - **`while`**, instead of **`if`**, is used to allow the waiting thread re-check the condition
  - Don't ever use this kind of pair due to poor performance and potential bug

```
while (ready == 0)
    ; // spin
```

```
ready = 1;
```