OSTEP Chapter 27

ECE 3600, Fall 2022

Table of Contents

1. Thread Creation
2. Simple Args
3. Return Args
4. Locks
5. Conditions
6. Exercises
7. Spinning vs. Yielding

1. Thread Creation

if *man pthread_create* does not work: **sudo apt install -y manpages-posix manpages-posix-dev**

pthread_create(), pthread_join()

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

threads-api/thread_create.c

```
#include <assert.h>
#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);
    assert(rc == 0);
    (void) pthread_join(p, NULL);
    printf("done\n");
    return 0;
}
```

```
$ gcc -Wall -o thread_create thread_create.c -pthread
$ ./thread_create
10 20
done
$
```

2. Simple Args

threads-api/thread_create_simple_args.c

```
 1  #include <stdio.h>
 2  #include <pthread.h>
 3  #include "common_threads.h"
 4
 5  void *mythread(void *arg) {
 6      long long int value = (long long int) arg;
 7      printf("%lld\n", value);
 8      return (void *) (value + 1);
 9  }
10
11  int main(int argc, char *argv[]) {
12      pthread_t p;
13      long long int rvalue;
14      Pthread_create(&p, NULL, mythread, (void *) 100);
15      Pthread_join(p, (void **) &rvalue);
16      printf("returned %lld\n", rvalue);
17      return 0;
18  }
```

```
$ gcc -Wall -I../include -o thread_create_simple_args \
   thread_create_simple_args.c -pthread
$ ./thread_create_simple_args
100
returned 101
$
```

3. Return Args

threads-api/thread_create_with_return_args.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include "common_threads.h"

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

typedef struct {
    int x;
    int y;
} myret_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("args %d %d\n", args->a, args->b);
    myret_t *rvals = malloc(sizeof(myret_t));
    assert(rvals != NULL);
    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;
}
```

```c
// bad alternative:
void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("args %d %d\n", args->a, args->b);
    myret_t oops = { 1, 2};
    return &oops; // mistake, local variable deallocated on return
}
```

```
$ gcc -Wall -I../include -o thread_create_with_return_args \
   thread_create_with_return_args.c -pthread
$ ./thread_create_with_return_args
args 10 20
returned 1 2
$
```

4. Locks

pthread_mutex_init(), pthread_mutex_lock() (and unlock)

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Example:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
// or: pthread_mutex_t lock; assert( pthread_mutex_init(&lock,NULL) == 0);
...
pthread_mutex_lock(&lock);
  x = x + 1; // or whatever your critical section is
  ...
pthread_mutex_unlock(&lock);
```

5. Conditions

pthread_cond_init(), pthread_cond_signal(), pthread_cond_wait()

```
int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);
```

Example:

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

// some thread, waiting:
Pthread_mutex_lock(&lock);
  while (ready == 0) Pthread_cond_wait(&cond, &lock);
  // condition occurred and we have the lock here
  ... do stuff ...
Pthread_mutex_unlock(&lock);

// some other thread, signaling:
Pthread_mutex_lock(&lock);
  ... do stuff ...
  ready = 1;
  Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

6. Exercises

See the book for exercises using helgrind, i.e. `valgrind --tool=helgrind`

Example: Q1,2 main-race.c

```
 1  #include <stdio.h>
 2
 3  #include "mythreads.h"
 4
 5  int balance = 0;
 6
 7  void* worker(void* arg) {
 8      balance++; // unprotected access
 9      return NULL;
10  }
11
12  int main(int argc, char *argv[]) {
13      pthread_t p;
14      Pthread_create(&p, NULL, worker, NULL);
15      balance++; // unprotected access
16      Pthread_join(p, NULL);
17      return 0;
18  }
```

```
$ valgrind --tool=helgrind ./main-race
...
==25147== Possible data race during read of size 4 at 0x30A040 by thread #1
==25147== Locks held: none
...
==25147== This conflicts with a previous write of size 4 by thread #2
==25147== Locks held: none
...
```

Fix using a lock.

7. Spinning vs. Yielding

Example: Q6,7 main-signal.c

```
 1  #include <stdio.h>
 2
 3  #include "mythreads.h"
 4
 5  int done = 0;
 6
 7  void* worker(void* arg) {
 8      printf("this should print first\n");
 9      done = 1;
10      return NULL;
11  }
12
13  int main(int argc, char *argv[]) {
14      pthread_t p;
15      Pthread_create(&p, NULL, worker, NULL);
16      while (done == 0)
17          ;
18      printf("this should print last\n");
19      return 0;
20  }
```

valgrind warns about a possible data race, but the warning is spurious and the code is correct.

Improve using sched_yield()