Chapter 27: Interlude - Thread API

Overview

- **Main Idea**: POSIX threads (pthreads) provide interfaces for thread creation, synchronization (locks/condition variables), and management.
- **Key Terms**: `pthread_create`, `pthread_join`, mutex, condition variable, critical section.

---

1. Thread Creation

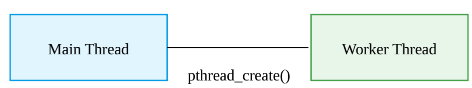
`pthread_create` Signature

```c
#include <pthread.h>
int pthread_create(
    pthread_t *thread,               // Thread identifier
    const pthread_attr_t *attr,      // Thread attributes (NULL for defaults)
    void *(*start_routine)(void *),  // Function pointer to thread's entry point
    void *arg                        // Argument passed to start_routine
);
```

Example: Passing Arguments

```c
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() {
    pthread_t p;
    myarg_t args = { 10, 20 };
    pthread_create(&p, NULL, mythread, &args);  // Pass struct as argument
    pthread_join(p, NULL);
}
```



(Main thread creates worker thread with arguments.)

---

2. Thread Completion

pthread_join Signature

```c
int pthread_join(pthread_t thread, void **retval);  // Waits for thread to finish
```

Example: Returning Values

```c
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;  // Return heap-allocated struct
}

int main() {
    pthread_t p;
    myret_t *rvals;
    pthread_create(&p, NULL, mythread, NULL);
    pthread_join(p, (void **) &rvals);
    printf("Returned %d %d\n", rvals->x, rvals->y);
    free(rvals);
}
```

Warning: Never return stack-allocated data from a thread!

```c
// BAD CODE: Returning stack address
myret_t oops;
return (void *) &oops;  // Crash-prone!
```

---

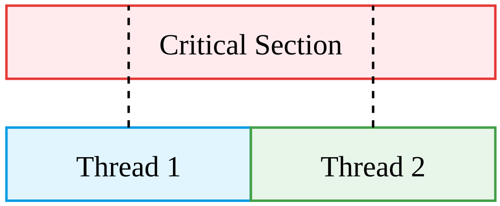
3. Locks (Mutexes)

Initialization

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;  // Static
// OR
pthread_mutex_init(&lock, NULL);                    // Dynamic
```

Usage

```c
pthread_mutex_lock(&lock);
x = x + 1;  // Critical section
pthread_mutex_unlock(&lock);
```



*(Threads serialize access to critical sections.)_

Common Pitfalls

1. **Missing Initialization**:

```c
pthread_mutex_t lock;  // Uninitialized → undefined behavior!
pthread_mutex_lock(&lock);
```

2. **Ignoring Error Codes**:
   Always check return values or use wrappers:

```c
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
    int rc = pthread_mutex_lock(mutex);
    assert(rc == 0);
}
```

---

4. Condition Variables

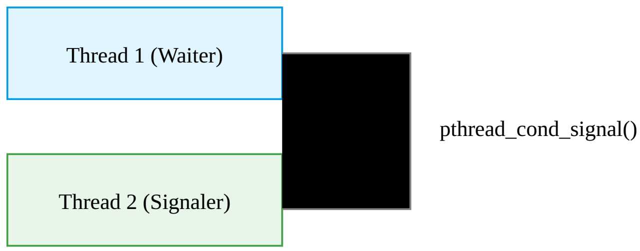
Initialization

```c
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Waiting and Signaling

```c
// Thread 1: Waits for condition
pthread_mutex_lock(&lock);
while (ready == 0) {
    pthread_cond_wait(&cond, &lock);  // Releases lock while waiting
}
pthread_mutex_unlock(&lock);

// Thread 2: Signals condition
pthread_mutex_lock(&lock);
ready = 1;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```



(Signaling thread wakes up waiting thread.)

Key Points:

- Always use a `while` loop (not `if`) for conditions due to spurious wakeups.
- Hold the mutex when modifying shared state (e.g., `ready`).

---

5. Thread API Guidelines

Do	Don't
Initialize locks/condition variables.	Use uninitialized synchronization primitives.
Check return codes.	Ignore error handling.
Pass heap-allocated data to threads.	Return stack addresses from threads.
Use condition variables for signaling.	Spin-wait on flags (`while (flag == 0);`).

Homework Insights

1. **Race Conditions**:

   ```
   valgrind --tool=helgrind main-race  # Detects data races
   ```

2. **Deadlocks**:
   Circular lock acquisition (e.g., T1 holds A, waits for B; T2 holds B, waits for A).

3. **Efficiency**:
   Spin-waiting (`while (done == 0);`) wastes CPU; condition variables are preferred.