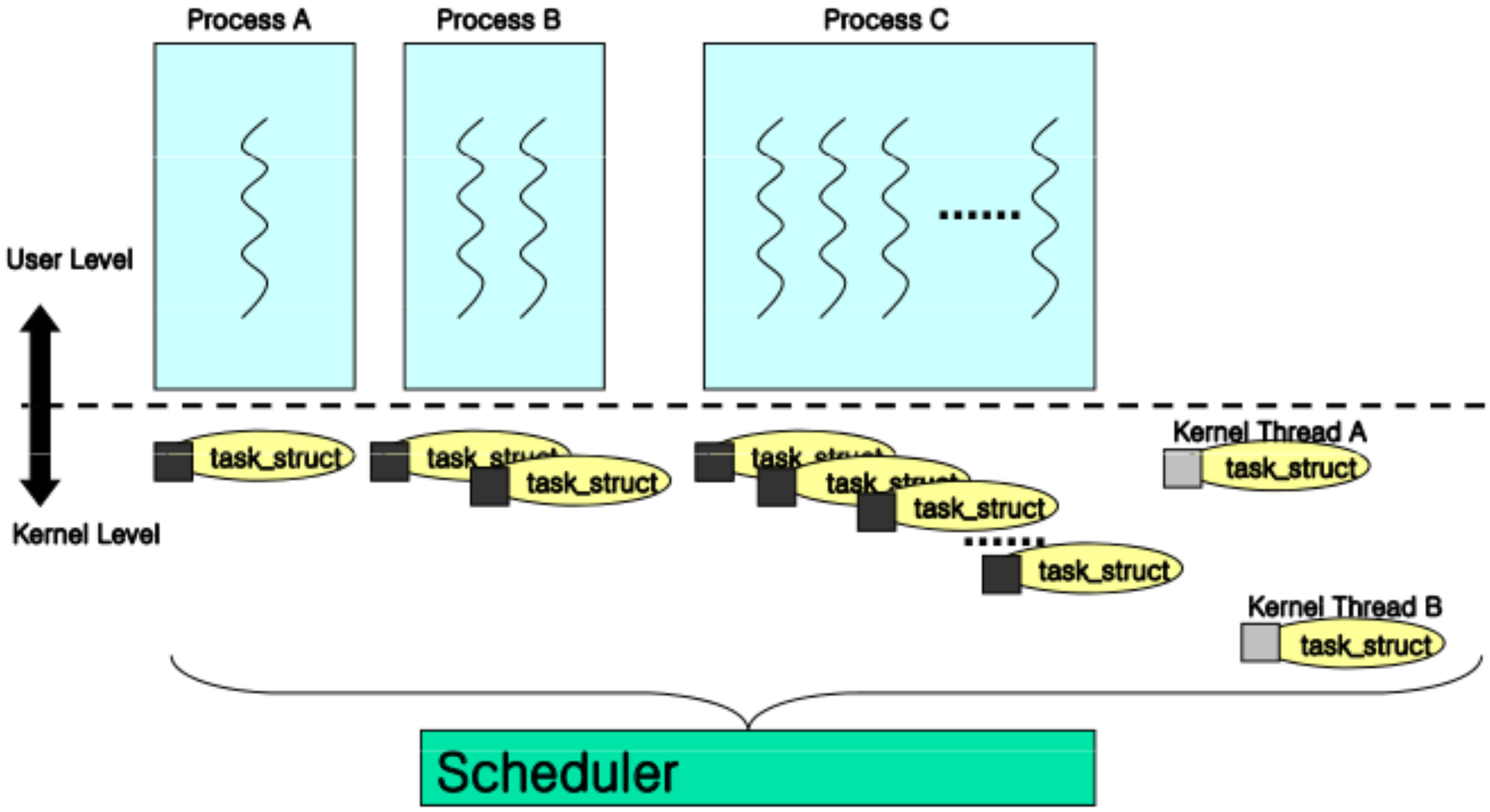
Chapter 11. Threads

朱金辉

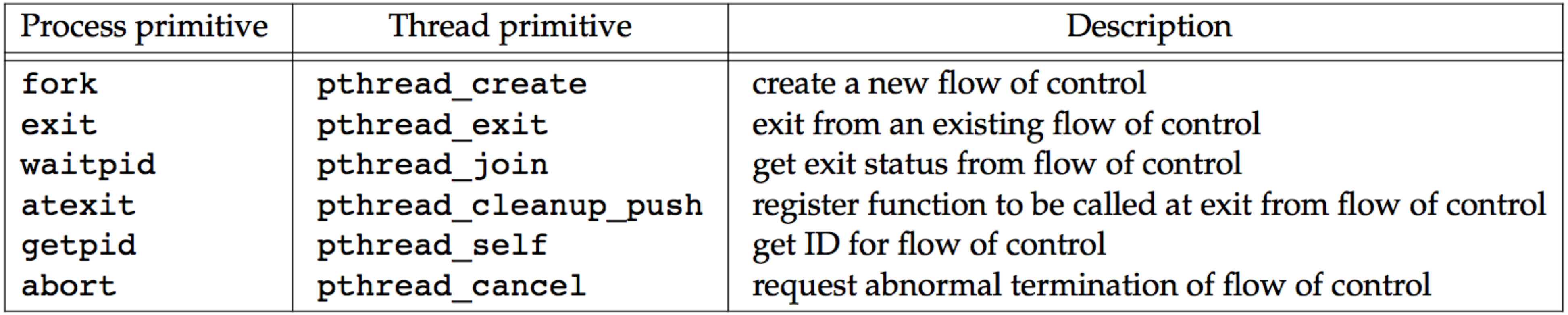
华南理工大学软件学院

1. Introduction

- Internal structure



Comparison of Process and Thread primitives



Process primitive	Thread primitive	Description
`fork`	`pthread_create`	create a new flow of control
`exit`	`pthread_exit`	exit from an existing flow of control
`waitpid`	`pthread_join`	get exit status from flow of control
`atexit`	`pthread_cleanup_push`	register function to be called at exit from flow of control
`getpid`	`pthread_self`	get ID for flow of control
`abort`	`pthread_cancel`	request abnormal termination of flow of control

- The threads interfaces, also known as "pthreads" for "POSIX threads"
- gcc hello.c -lpthread

Thread Creation

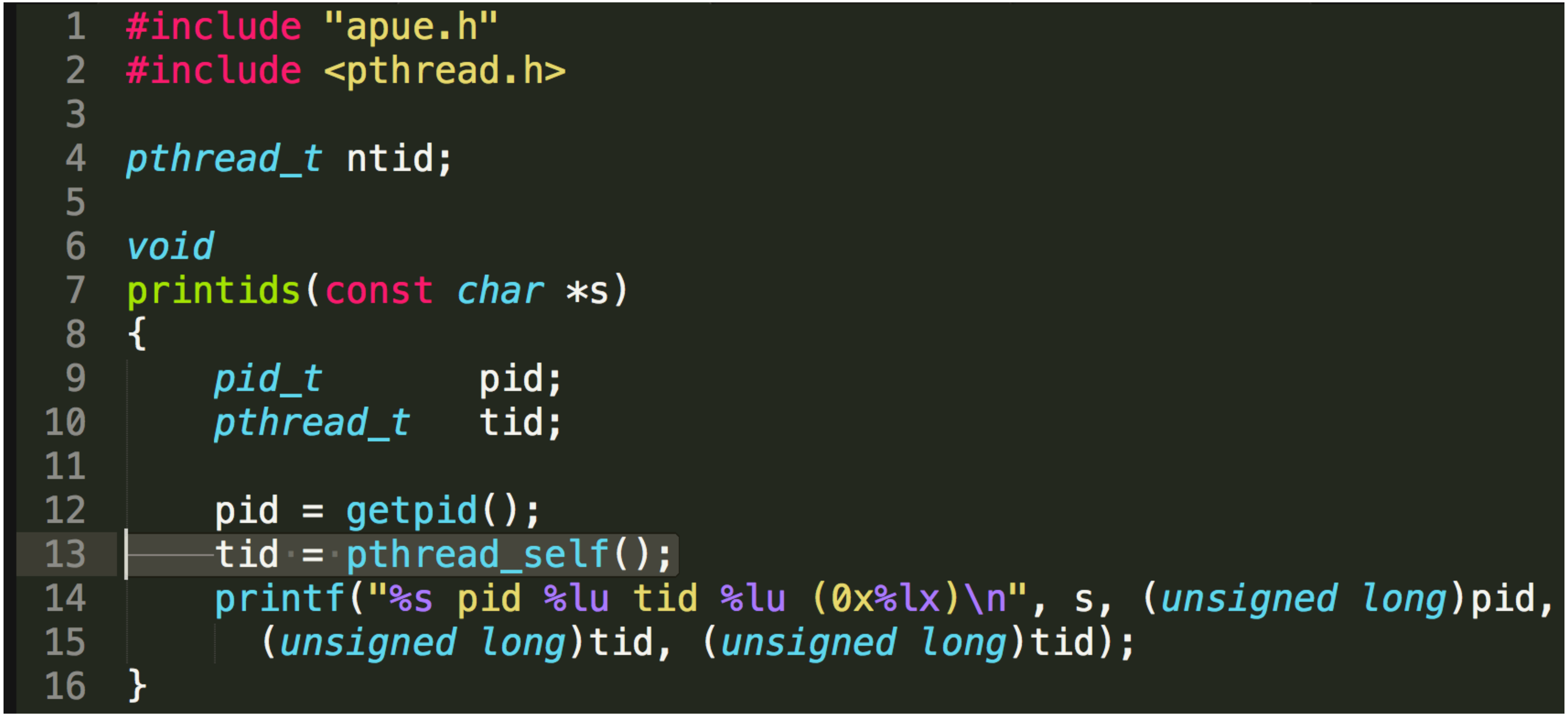
- #include <pthread.h>
- int pthread_create(

  pthread_t *restrict *tidp*,
  const pthread_attr_t *restrict *attr*,
  void *(**start_rtn*)(void *),
  void *restrict *arg*);
- Returns: 0 if OK, error number on failure

Thread Identification

- #include <pthread.h>
- pthread_t pthread_self(void);
- Returns: the thread ID of the calling thread

Figure 11.2 Printing thread IDs

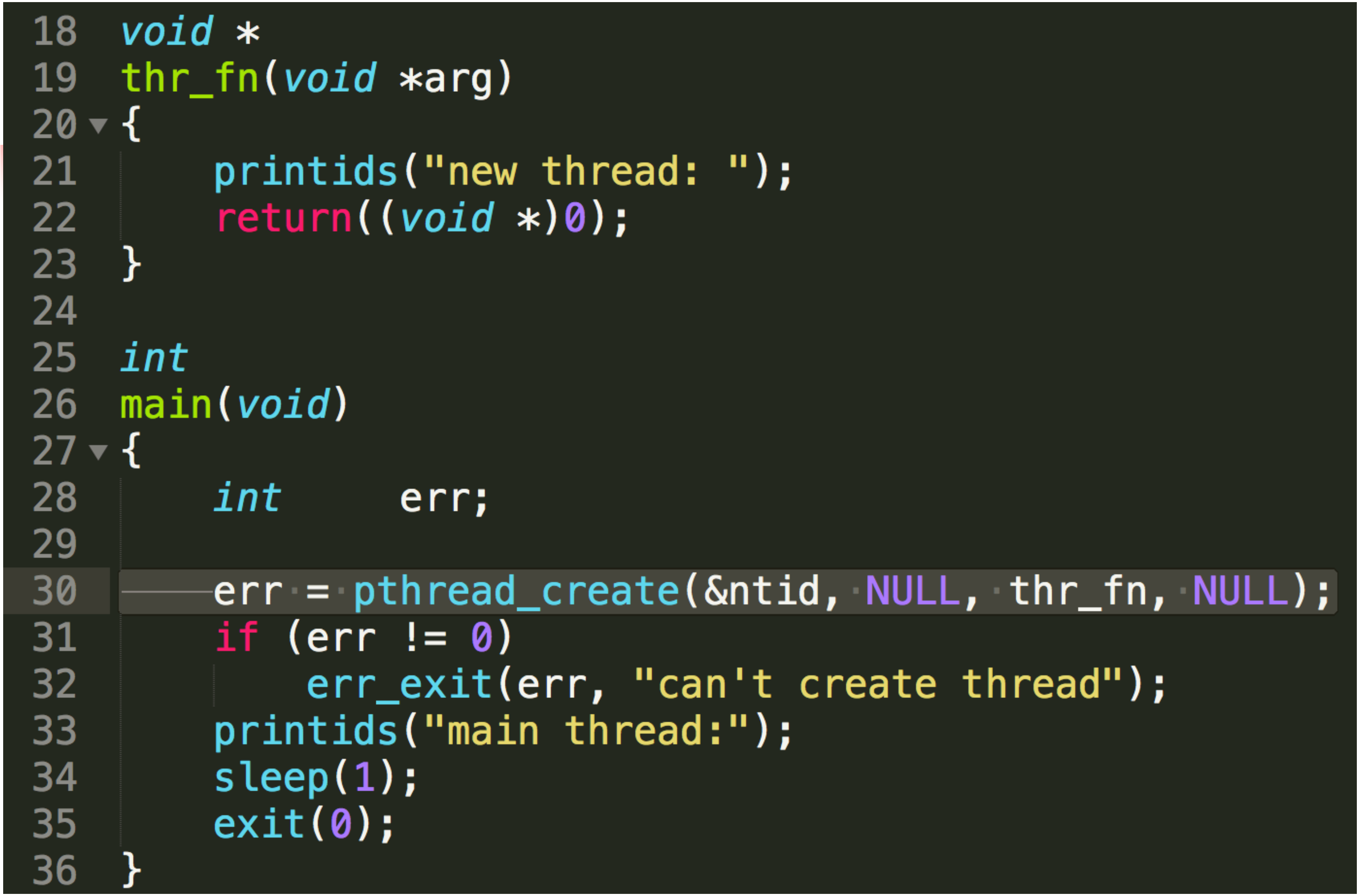


```c
#include "apue.h"
#include <pthread.h>

pthread_t ntid;

void
printids(const char *s)
{
    pid_t       pid;
    pthread_t   tid;

    pid = getpid();
    tid = pthread_self();
    printf("%s pid %lu tid %lu (0x%lx)\n", s, (unsigned long)pid,
      (unsigned long)tid, (unsigned long)tid);
}
```



```c
void *
thr_fn(void *arg)
{
    printids("new thread: ");
    return((void *)0);
}

int
main(void)
{
    int     err;

    err = pthread_create(&ntid, NULL, thr_fn, NULL);
    if (err != 0)
        err_exit(err, "can't create thread");
    printids("main thread:");
    sleep(1);
    exit(0);
}
```

Thread Termination

- A single thread can exit in three ways, thereby stopping its flow of control, without terminating the entire process.

1. The thread can simply return from the start routine.
2. The thread can be canceled by another thread in the same process.
   - int pthread_cancel(pthread_t *tid*);
3. The thread can call pthread_exit.
   - void pthread_exit(void **rval_ptr*);

pthread_join

- #include <pthread.h>

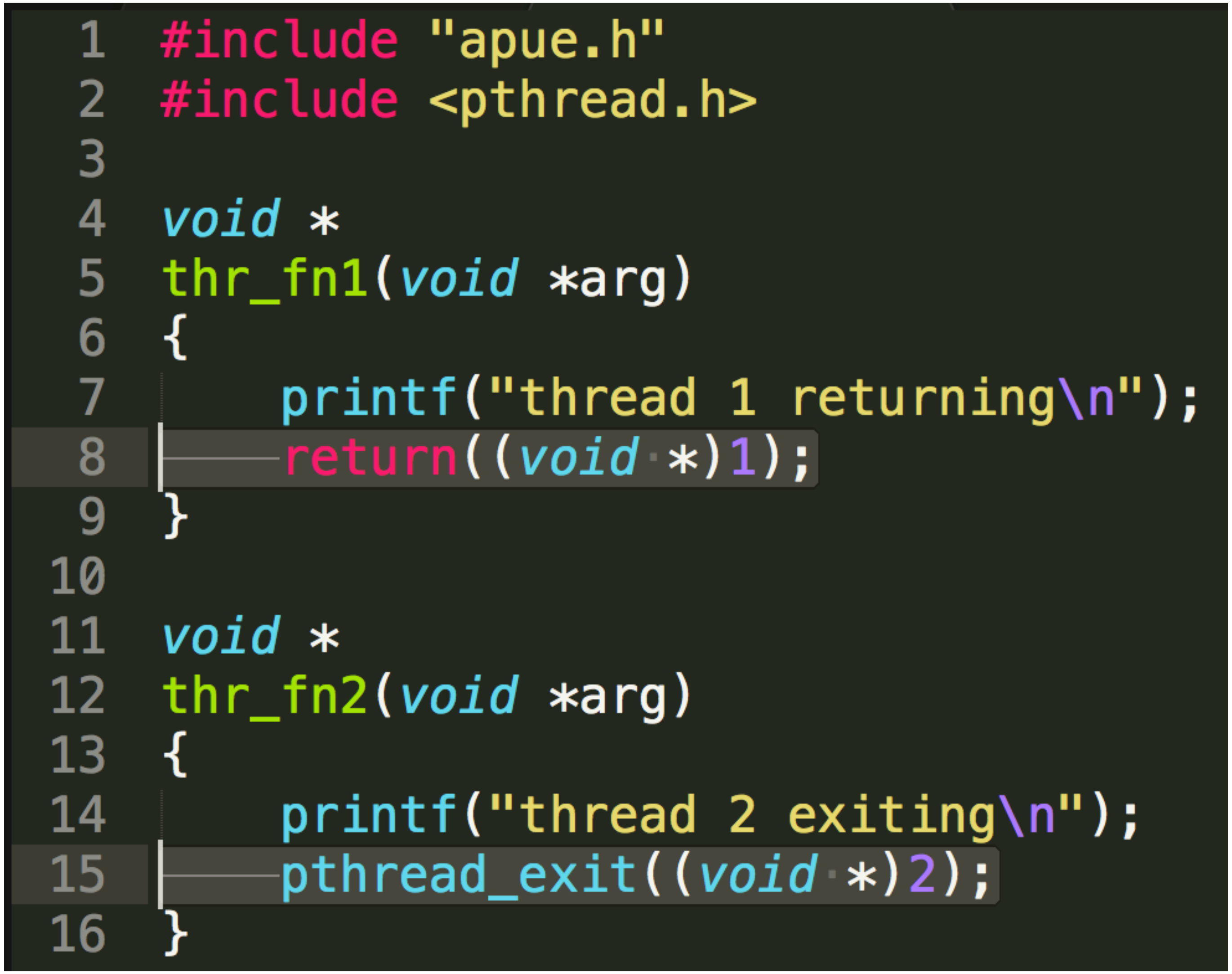
- int pthread_join(

  pthread_t *thread*,

  void ***rval_ptr*);

- Returns: 0 if OK, error number on failure

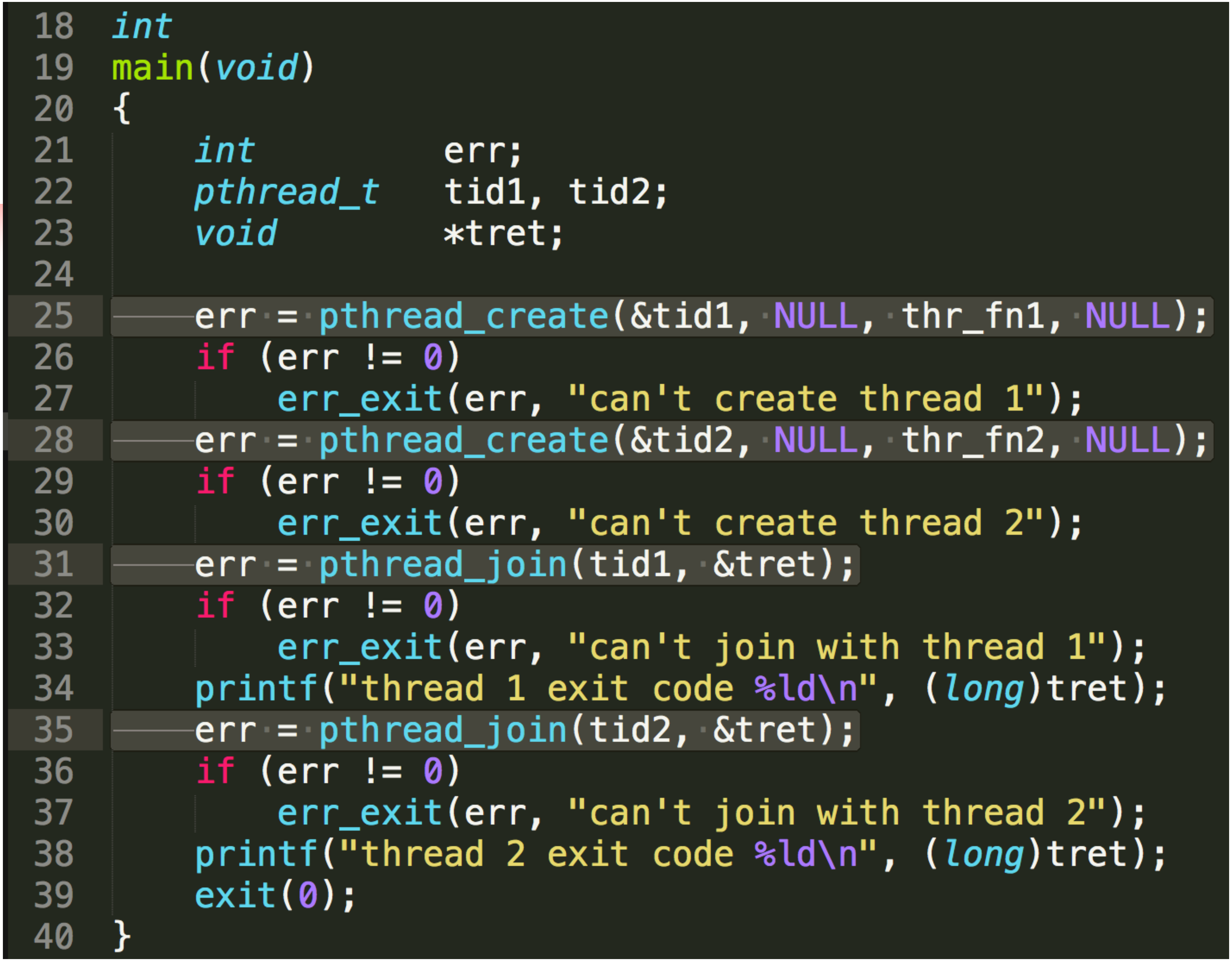
Figure 11.3 Fetching the thread exit status



```c
#include "apue.h"
#include <pthread.h>

void *
thr_fn1(void *arg)
{
    printf("thread 1 returning\n");
    return((void *)1);
}

void *
thr_fn2(void *arg)
{
    printf("thread 2 exiting\n");
    pthread_exit((void *)2);
}
```



```c
int
main(void)
{
    int         err;
    pthread_t   tid1, tid2;
    void        *tret;

    err = pthread_create(&tid1, NULL, thr_fn1, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 1");
    err = pthread_create(&tid2, NULL, thr_fn2, NULL);
    if (err != 0)
        err_exit(err, "can't create thread 2");
    err = pthread_join(tid1, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 1");
    printf("thread 1 exit code %ld\n", (long)tret);
    err = pthread_join(tid2, &tret);
    if (err != 0)
        err_exit(err, "can't join with thread 2");
    printf("thread 2 exit code %ld\n", (long)tret);
    exit(0);
}
```

pthread_cleanup_push

- include <pthread.h>
- void pthread_cleanup_push(
  void (**rtn*)(void *),
  void **arg*);
- void pthread_cleanup_pop(int *execute*);

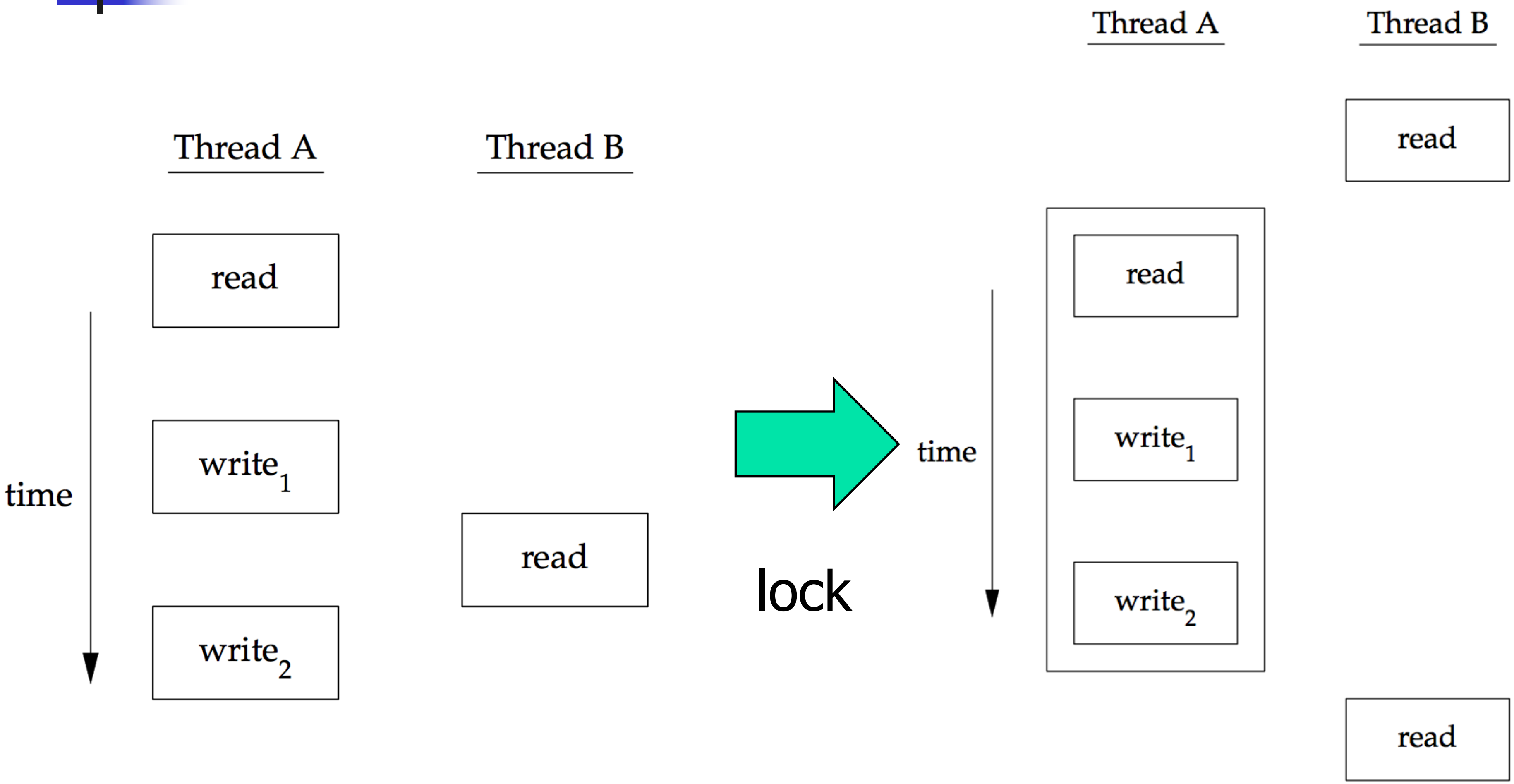
Figure 11.5 Thread cleanup handler

- *rtn* will be called with the single argument, *arg*, when the thread performs one of the following actions:
  - Makes a call to pthread_exit
  - Responds to a cancellation request
  - Makes a call to pthread_cleanup_pop with a nonzero *execute* argument

pthread_detach

- #include <pthread.h>
- int pthread_detach(pthread_t *tid*);
- Returns: 0 if OK, error number on failure

Thread Synchronization



Mutexes

- #include <pthread.h>
- int pthread_mutex_init(

  pthread_mutex_t *restrict *mutex*,

  const pthread_mutexattr_t *restrict *attr*);
- int pthread_mutex_destroy(

  pthread_mutex_t **mutex*);
- Both return: 0 if OK, error number on fai

lock

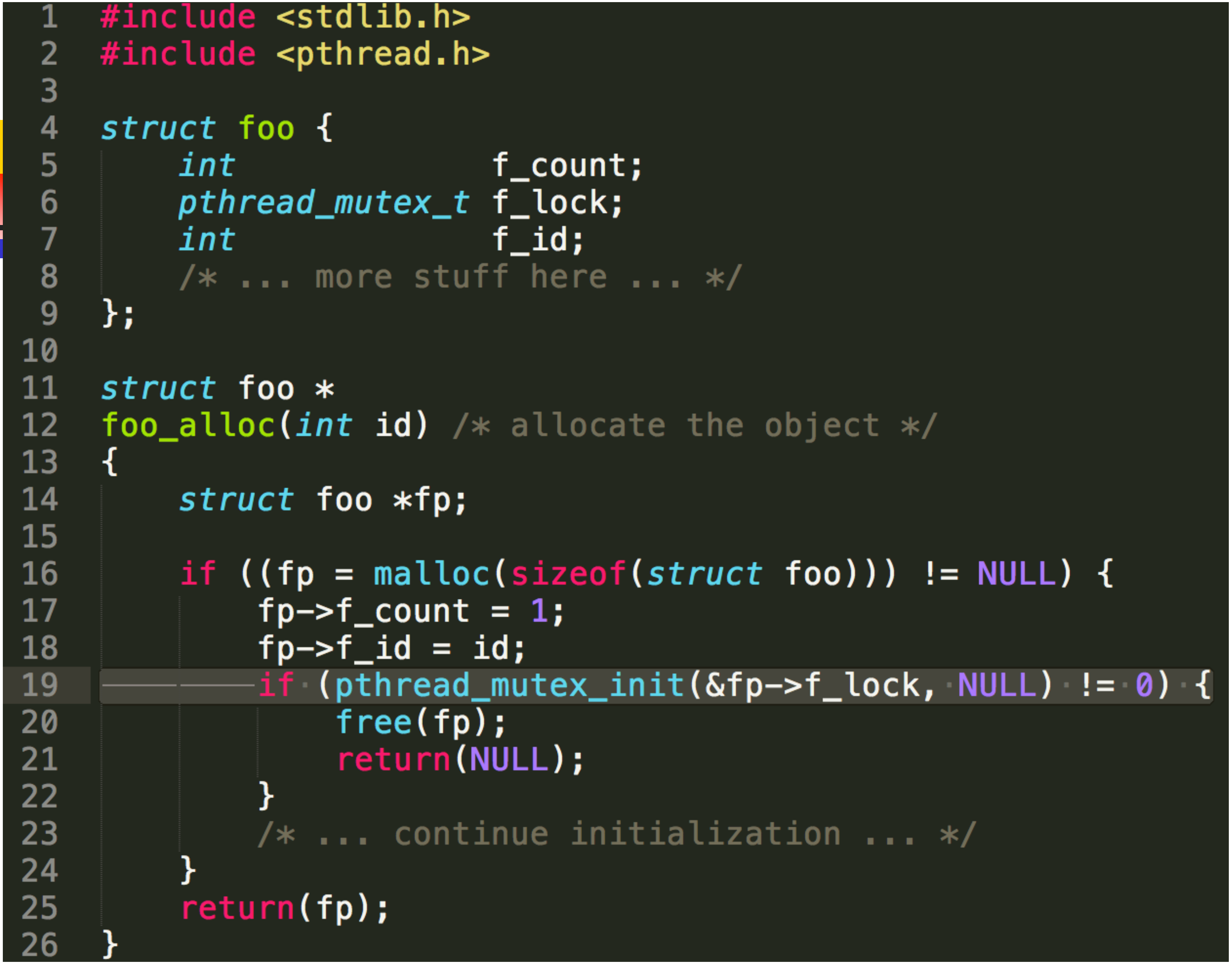
- #include <pthread.h>
- int pthread_mutex_lock(pthread_mutex_t **mutex*);
- int pthread_mutex_trylock(pthread_mutex_t **mutex*);
- int pthread_mutex_unlock(pthread_mutex_t **mutex*);
- All return: 0 if OK, error number on failure

timelock

- #include <pthread.h>
- #include <time.h>
- int pthread_mutex_timedlock(

  pthread_mutex_t *restrict *mutex*,

  const struct timespec *restrict *tsptr*);
- Returns: 0 if OK, error number on failure

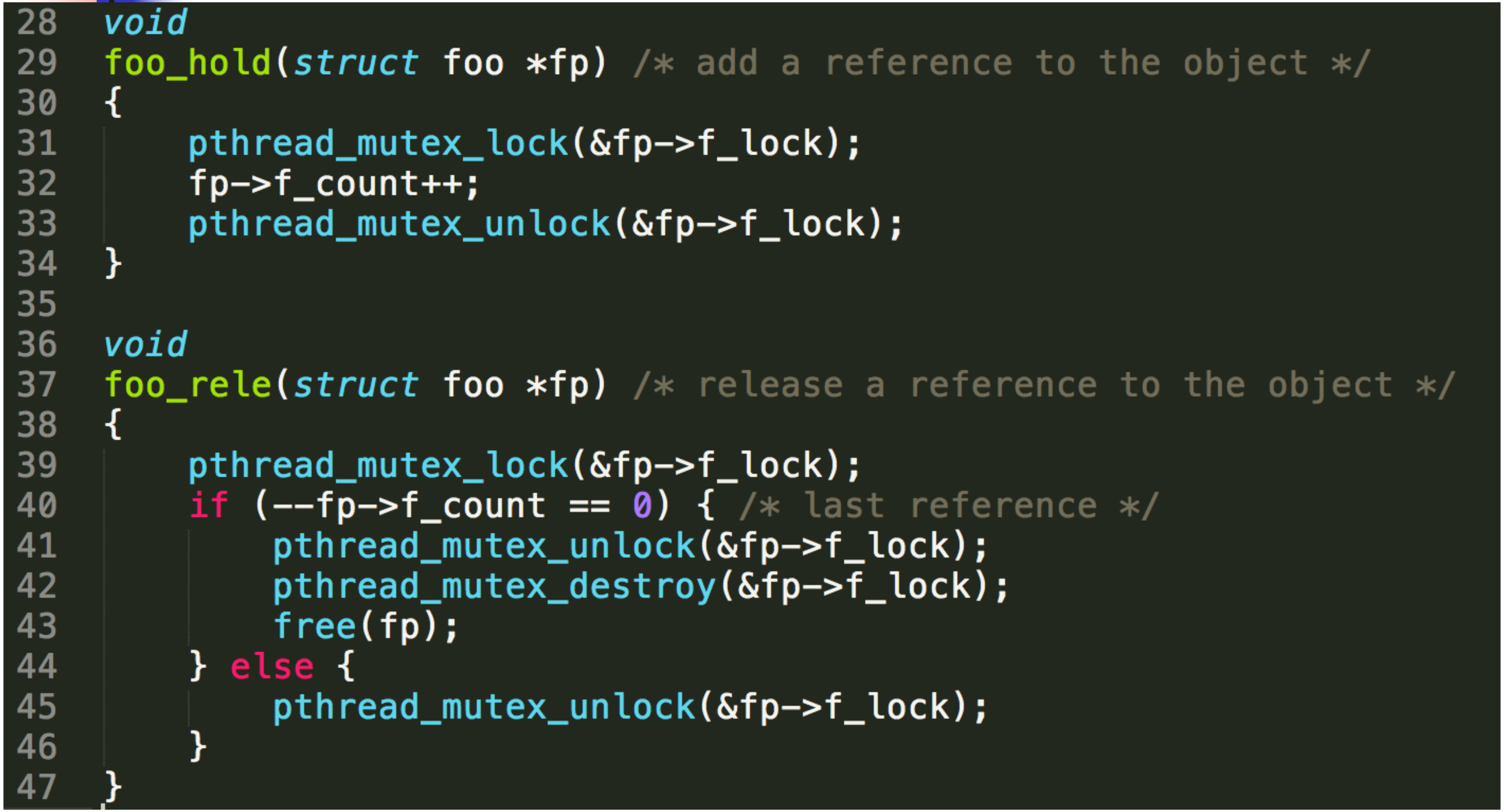


```c
#include <stdlib.h>
#include <pthread.h>

struct foo {
    int                 f_count;
    pthread_mutex_t f_lock;
    int                 f_id;
    /* ... more stuff here ... */
};

struct foo *
foo_alloc(int id) /* allocate the object */
{
    struct foo *fp;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        /* ... continue initialization ... */
    }
    return(fp);
}
```



```c
void
foo_hold(struct foo *fp) /* add a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    fp->f_count++;
    pthread_mutex_unlock(&fp->f_lock);
}

void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    pthread_mutex_lock(&fp->f_lock);
    if (--fp->f_count == 0) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Deadlock

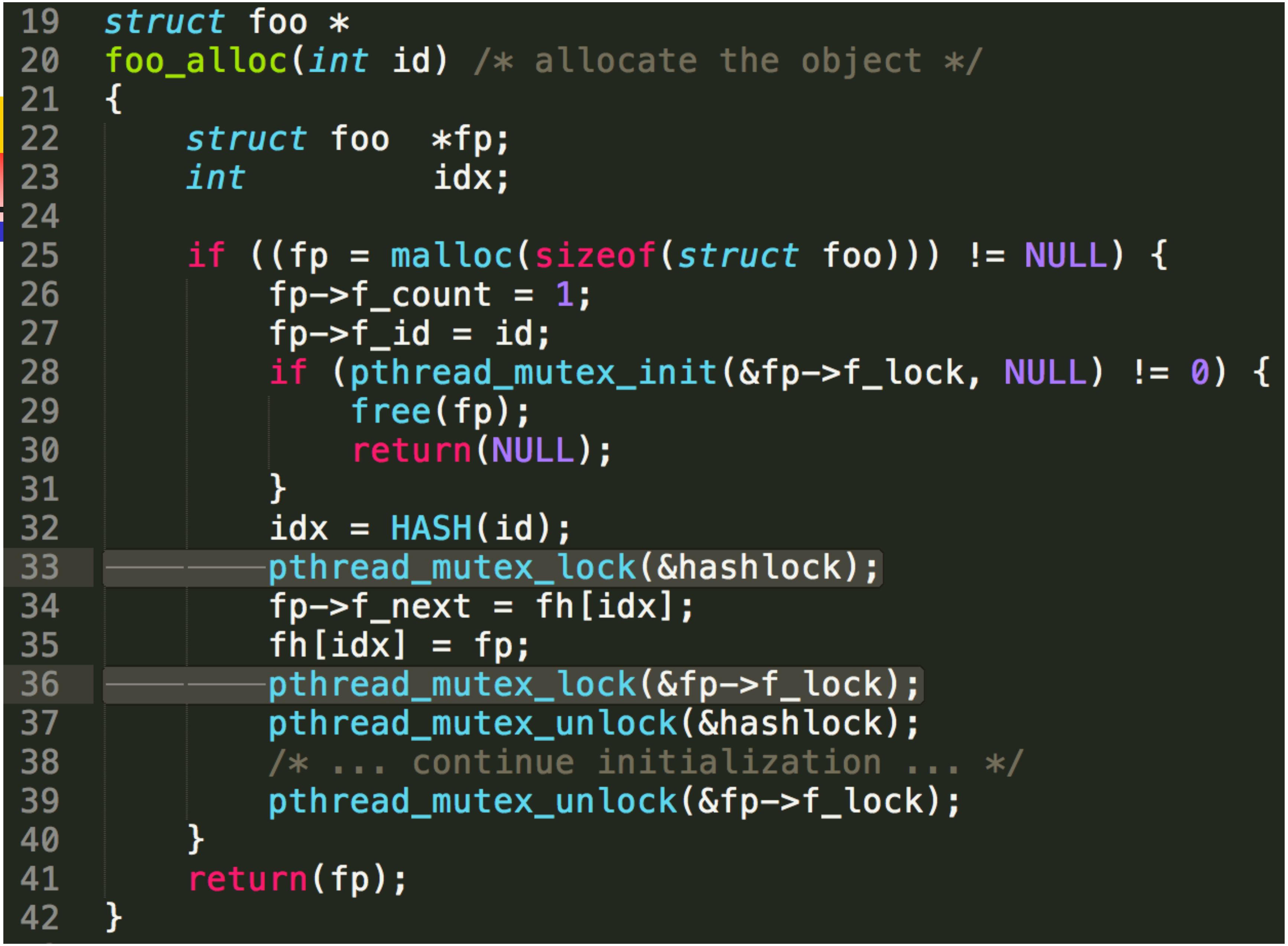
- A thread will deadlock itself if it tries to lock the same mutex twice.
- When we use more than one mutex in our programs, a deadlock can occur if one thread T1 to hold a mutex A and block while trying to lock a second mutex B at the same time that another thread T2 holding the second mutex B tries to lock the first mutex A.

	T1	T2
Have lock	A	B
Want	B	A

Deadlock Avoidance

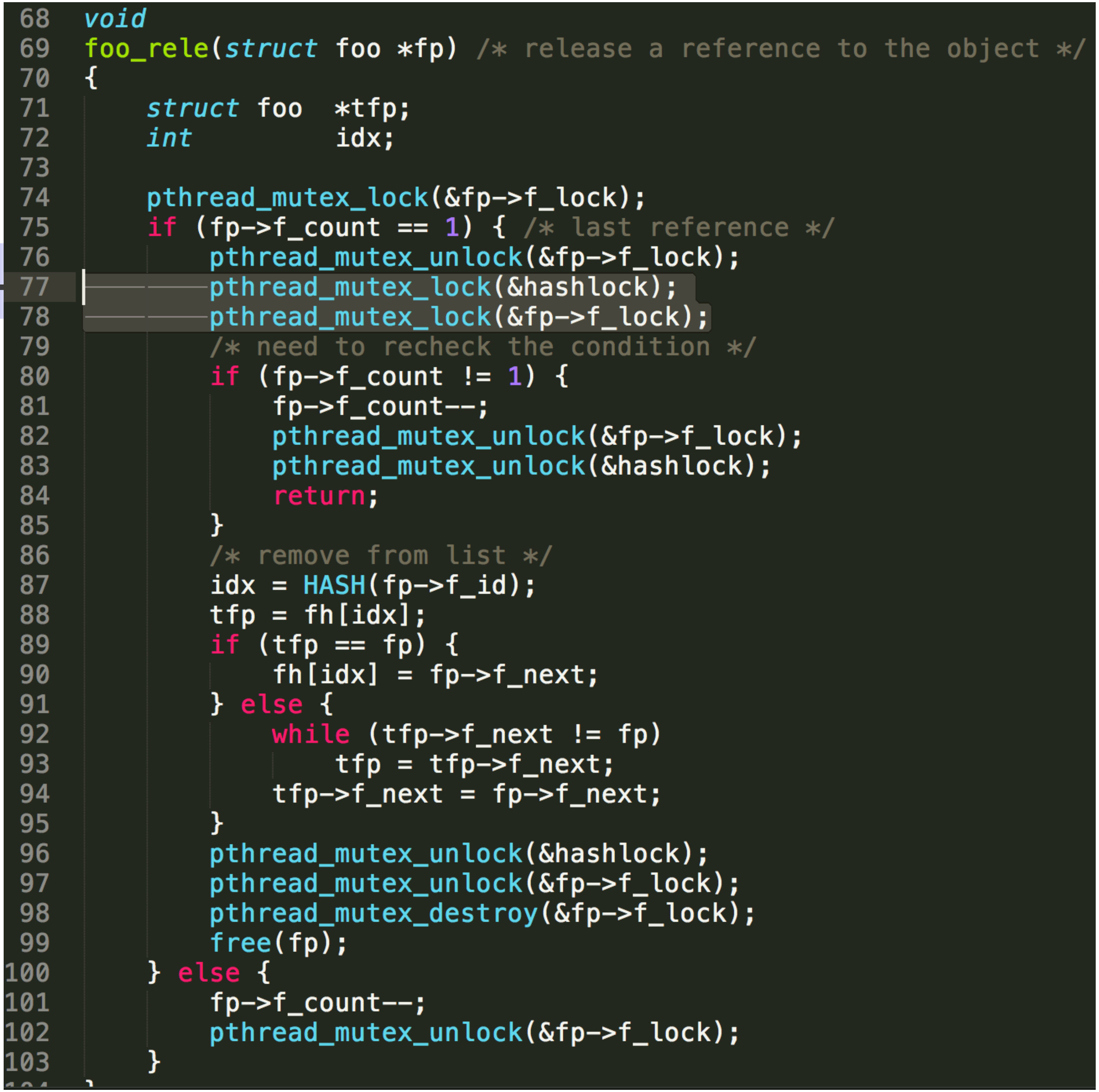
- Deadlocks can be avoided by carefully controlling the order in which mutexes are locked.
- If all threads always lock mutex A before mutex B, no deadlock can occur from the use of the two mutexes.

T1	T2
A	A
B	B



```c
struct foo *
foo_alloc(int id) /* allocate the object */
{
    struct foo  *fp;
    int         idx;

    if ((fp = malloc(sizeof(struct foo))) != NULL) {
        fp->f_count = 1;
        fp->f_id = id;
        if (pthread_mutex_init(&fp->f_lock, NULL) != 0) {
            free(fp);
            return(NULL);
        }
        idx = HASH(id);
        pthread_mutex_lock(&hashlock);
        fp->f_next = fh[idx];
        fh[idx] = fp;
        pthread_mutex_lock(&fp->f_lock);
        pthread_mutex_unlock(&hashlock);
        /* ... continue initialization ... */
        pthread_mutex_unlock(&fp->f_lock);
    }
    return(fp);
}
```



```
void
foo_rele(struct foo *fp) /* release a reference to the object */
{
    struct foo  *tfp;
    int         idx;

    pthread_mutex_lock(&fp->f_lock);
    if (fp->f_count == 1) { /* last reference */
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_lock(&hashlock);
        pthread_mutex_lock(&fp->f_lock);
        /* need to recheck the condition */
        if (fp->f_count != 1) {
            fp->f_count--;
            pthread_mutex_unlock(&fp->f_lock);
            pthread_mutex_unlock(&hashlock);
            return;
        }
        /* remove from list */
        idx = HASH(fp->f_id);
        tfp = fh[idx];
        if (tfp == fp) {
            fh[idx] = fp->f_next;
        } else {
            while (tfp->f_next != fp)
                tfp = tfp->f_next;
            tfp->f_next = fp->f_next;
        }
        pthread_mutex_unlock(&hashlock);
        pthread_mutex_unlock(&fp->f_lock);
        pthread_mutex_destroy(&fp->f_lock);
        free(fp);
    } else {
        fp->f_count--;
        pthread_mutex_unlock(&fp->f_lock);
    }
}
```

Reader-writer locks

- Three states are possible:

1. locked in read mode
2. locked in write mode
3. unlocked

rwlock api

- #include <pthread.h>
- int pthread_rwlock_init(
  pthread_rwlock_t *restrict _rwlock_,
  const pthread_rwlockattr_t *restrict _attr_);
- int pthread_rwlock_destroy(
  pthread_rwlock_t *_rwlock_);
- Both return: 0 if OK, error number on failure

- #include <pthread.h>
- int pthread_rwlock_rdlock(pthread_rwlock_t **rwlock*);
- int pthread_rwlock_wrlock(pthread_rwlock_t **rwlock*);
- int pthread_rwlock_unlock(pthread_rwlock_t **rwlock*);
- int pthread_rwlock_tryrdlock(pthread_rwlock_t **rwlock*);
- int pthread_rwlock_trywrlock(pthread_rwlock_t **rwlock*);
- int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict *rwlock*, const struct timespec *restrict *tsptr*);
- int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict *rwlock*, const struct timespec *restrict *tsptr*);
- All return: 0 if OK, error number on failure

Condition Variables

- #include <pthread.h>
- int pthread_cond_init(

  pthread_cond_t *restrict *cond*,

  const pthread_condattr_t *restrict *attr*);
- int pthread_cond_destroy(

  pthread_cond_t **cond*);
- Both return: 0 if OK, error number on failure

wait

- #include <pthread.h>
- int pthread_cond_wait(

  pthread_cond_t *restrict *cond*,

  pthread_mutex_t *restrict *mutex*);
- int pthread_cond_timedwait(

  pthread_cond_t *restrict *cond*, pthread_mutex_t *restrict *mutex*,

  const struct timespec *restrict *tsptr*);
- Both return: 0 if OK, error number on failure

✓ atomically places the calling thread on the list of threads waiting for the condition and unlocks the mutex.

✓ When pthread_cond_wait returns, the mutex is again locked.

signal

- #include <pthread.h>
- int pthread_cond_signal(

  pthread_cond_t **cond*);

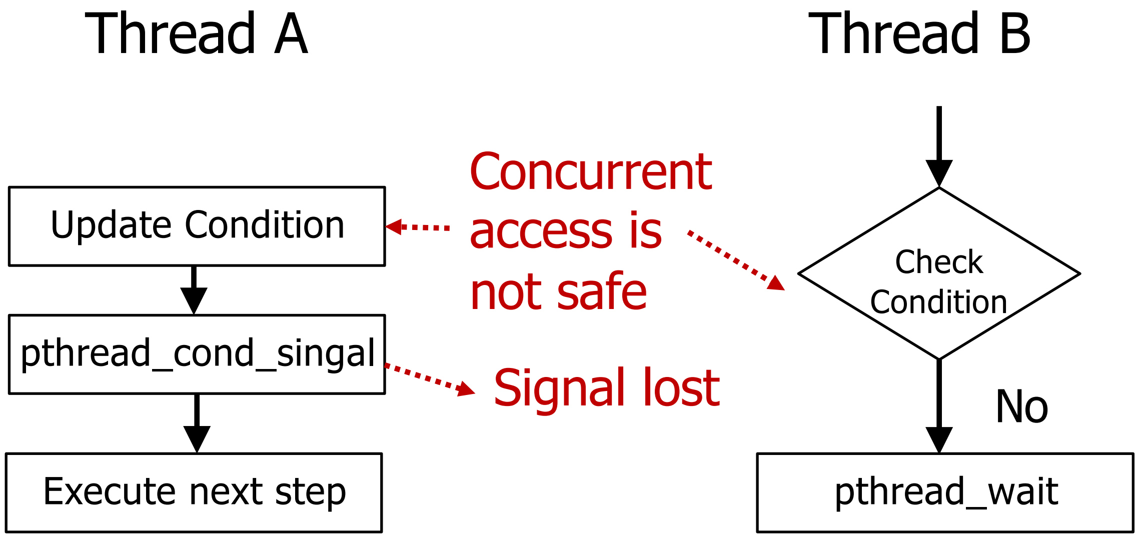
- int pthread_cond_broadcast(

  pthread_cond_t **cond*);

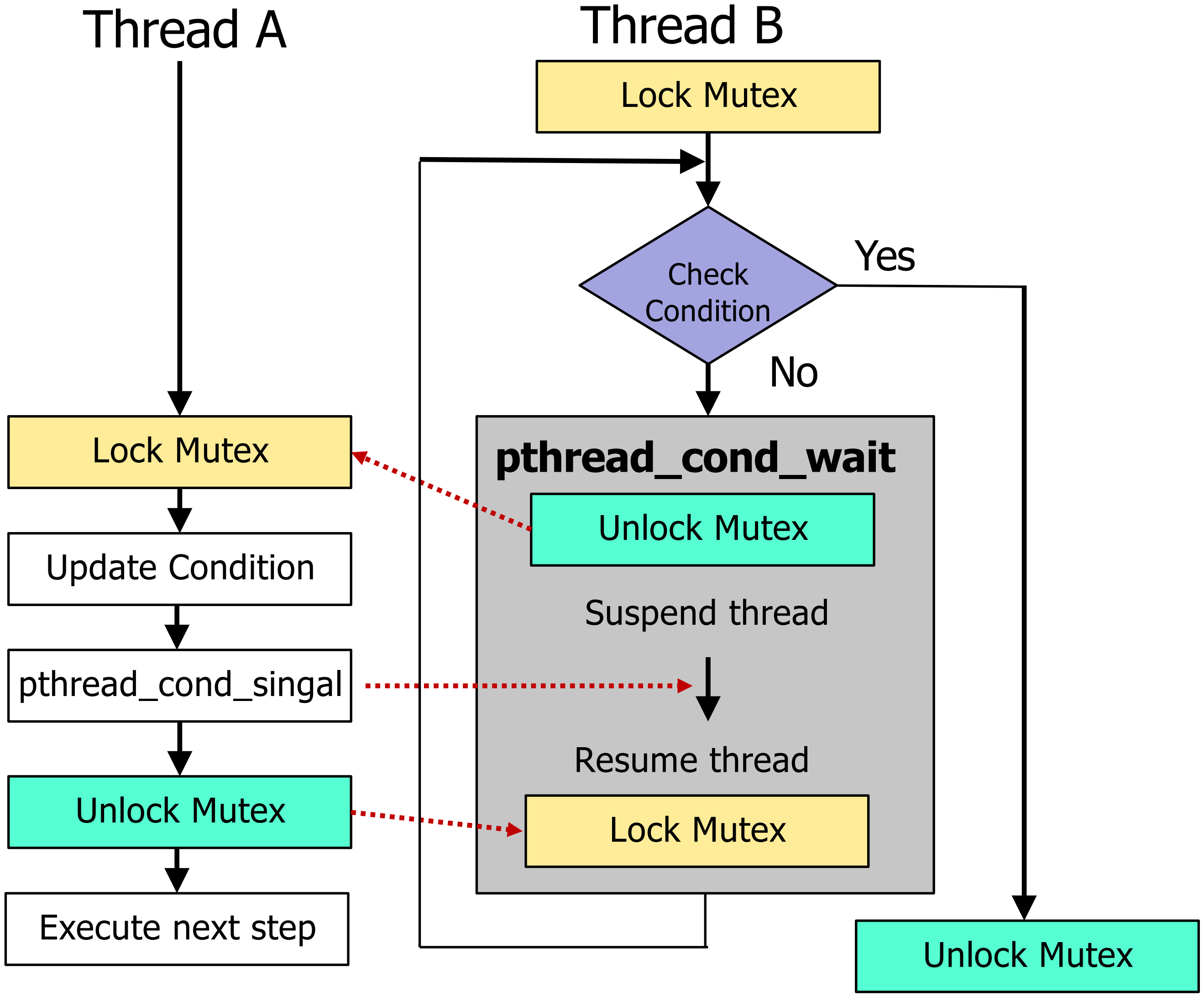
- Both return: 0 if OK, error number on failure

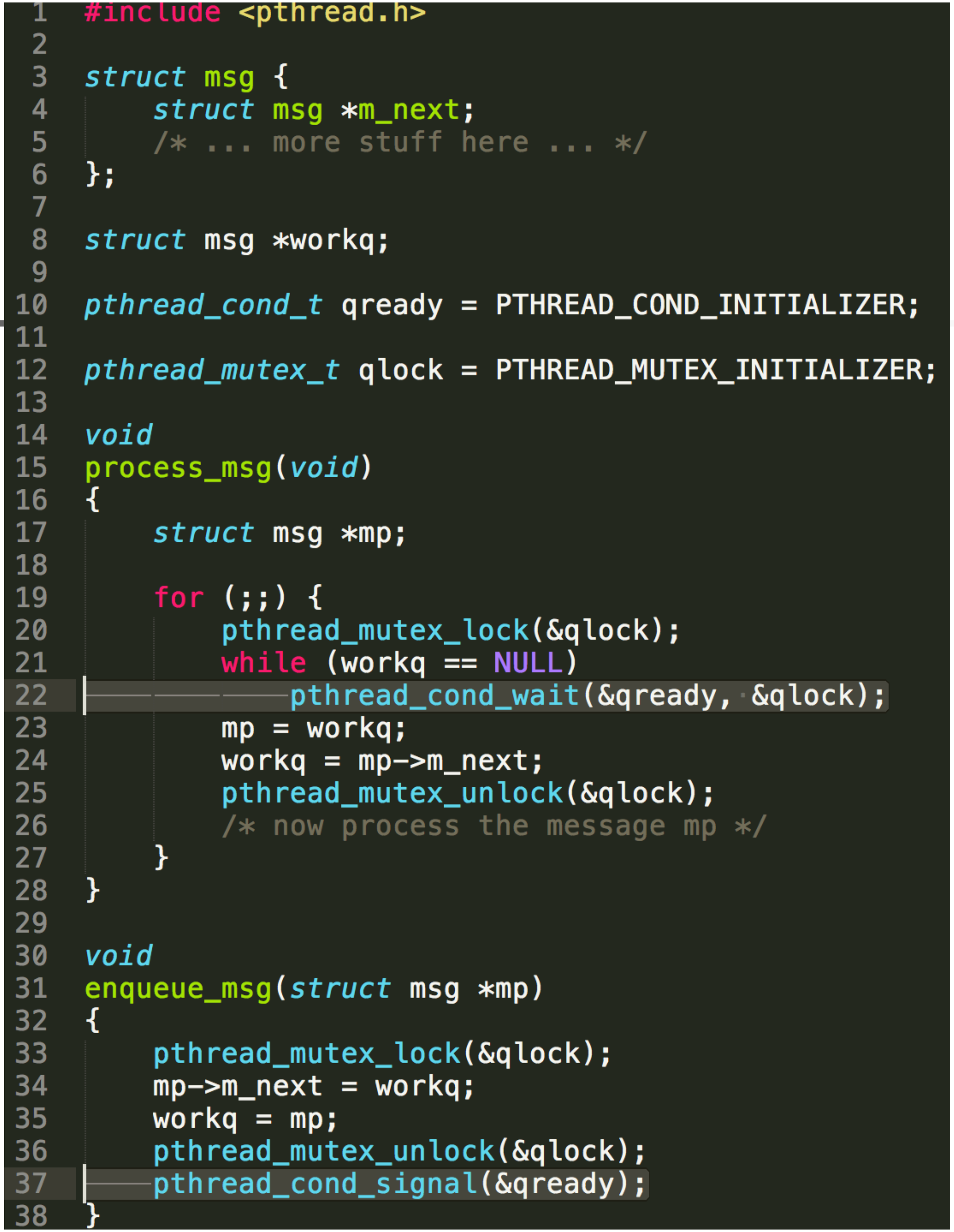
Problem

- P1: Thread_B in right side runs first. If Thread_A modifies the condition during condition check in Thread_B, Thread_B could not notice this change.
- P2: pthread_cond_signal in Thread_A can execute before pthread_cond_wait of Thread_B, Thread_B will not wake up forever.



Singal <----> Wait


Thread A
Thread B
Lock Mutex
Check Condition
Yes
No
pthread_cond_wait
Unlock Mutex
Suspend thread
Resume thread
Lock Mutex
Lock Mutex
Update Condition
pthread_cond_singal
Unlock Mutex
Execute next step
Unlock Mutex



```c
#include <pthread.h>

struct msg {
    struct msg *m_next;
    /* ... more stuff here ... */
};

struct msg *workq;

pthread_cond_t qready = PTHREAD_COND_INITIALIZER;

pthread_mutex_t qlock = PTHREAD_MUTEX_INITIALIZER;

void
process_msg(void)
{
    struct msg *mp;

    for (;;) {
        pthread_mutex_lock(&qlock);
        while (workq == NULL)
            pthread_cond_wait(&qready, &qlock);
        mp = workq;
        workq = mp->m_next;
        pthread_mutex_unlock(&qlock);
        /* now process the message mp */
    }
}

void
enqueue_msg(struct msg *mp)
{
    pthread_mutex_lock(&qlock);
    mp->m_next = workq;
    workq = mp;
    pthread_mutex_unlock(&qlock);
    pthread_cond_signal(&qready);
}
```

Spin Locks

- A spin lock is like a mutex, except that instead of blocking a process by sleeping, the process is blocked by busy-waiting (spinning) until the lock can be acquired.

Spin lock

- #include <pthread.h>
- int pthread_spin_init(pthread_spinlock_t **lock*, int *pshared*);
- int pthread_spin_destroy(pthread_spinlock_t **lock*);
- int pthread_spin_lock(pthread_spinlock_t **lock*);
- int pthread_spin_trylock(pthread_spinlock_t **lock*);
- int pthread_spin_unlock(pthread_spinlock_t **lock*);
- All return: 0 if OK, error number on failure

Barriers

- pthread_join function acts as a barrier to allow one thread to wait until another thread exits.

- A barrier allows each thread to wait until all cooperating threads have reached the same point, and then continue executing from there.

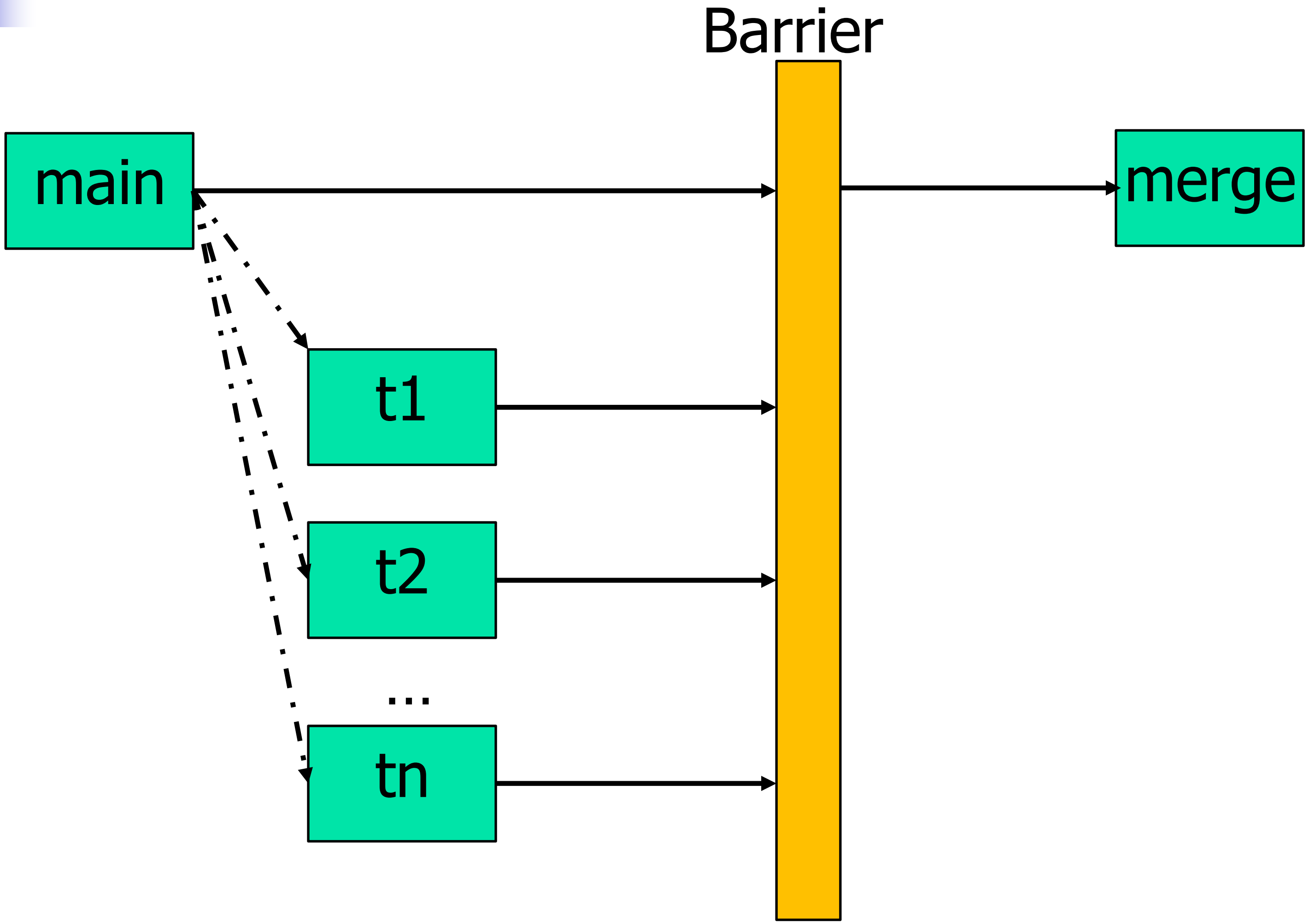
Barrier API

- #include <pthread.h>
- int pthread_barrier_init(

  pthread_barrier_t *restrict *barrier*,
  const pthread_barrierattr_t *restrict *attr*,
  unsigned int *count*);
- int pthread_barrier_destroy(pthread_barrier_t **barrier*);
- Both return: 0 if OK, error number on failure

- int pthread_barrier_wait(pthread_barrier_t **barrier*);
- Returns: 0 or PTHREAD_BARRIER_SERIAL_THREAD if OK, error number on failure

Example


Barrier
main
merge
t1
t2
…
tn