



Pthreads, Locks, Semaphores

CS 241 Discussion Section 3
6 February – 9 February

Outline

- pthreads
- Thread Safety
- Synchronization
 - Mutex locks
 - Semaphores

Thread functions

- A function that is used as a thread must have a special format.

```
void* threadfunction(void* arg);
```

- `void*` - can be a pointer to anything
 - Use a pointer to a `struct` to effectually pass any number of parameters.
 - Return any data structure too.

POSIX Thread Functions

- Most POSIX functions return 0 on success and a nonzero error code on failure.
- `errno` not set
 - returns value `errno` would have
- POSIX thread functions never return `EINTR`

Creating a thread

A thread is created with `pthread_create`

```
int thread_id;
int pthread_create(
    pthread_t *thread_id,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void *args );
```

1. Thread ID - Calling thread must provide location for this
2. Thread attributes - `NULL` pointer indicates default attributes
3. Function for thread to run
4. Arguments

Detaching Threads

- Detach a thread to make it release its resources upon exiting
- Detached thread cannot be joined.
- **Example 12.5**

```
void *processfd(void *arg);

int error;
int fd
pthread_t tid;

if (error = pthread_create(&tid, NULL, processfd, &fd))
    fprintf(stderr, "Failed to create thread: %s\n",
strerror(error));
else if (error = pthread_detach(tid))
    fprintf(stderr, "Failed to detach thread: %s\n",
strerror(error));
```

- Make a thread detach itself
 - Replace `tid` with `pthread_self()`

Joining Threads

- Suspends caller until specified thread exits
 - Similar to `waitpid` for processes
- Good practice – call `pthread_detach` or `pthread_join` for every thread
- Following gets value passed to `pthread_exit` by terminating thread
- **Program 12.5**

```
void *copyfilemalloc(void *arg);

if (((fds[0] = open(argv[1], READ_FLAGS)) == -1) ||
    ((fds[1] = open(argv[2], WRITE_FLAGS, PERMS)) == -1)) {
    perror("Failed to open the files");
    return 1;
}
if (error = pthread_create(&tid, NULL, copyfilemalloc, fds)) {
    fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
    return 1;
}
if (error = pthread_join(tid, (void **)&bytesptr)) {
    fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
    return 1;
}
```

What if?: `pthread_join(pthread_self())`

Exiting and Cancelling

- Process & all threads terminate when:
 - The process or any thread calls `exit`
 - The process executes `return` from `main`
 - It receives a signal

```
void pthread_exit(void *value_ptr);
```

- Main thread should block or call `pthread_exit(NULL)` to wait for all threads
-

- Threads can request another thread to be cancelled by:

```
void pthread_cancel(pthread_t thread);
```

Table 12.1

POSIX function	Description
pthread_cancel	terminate another thread
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_kill	send a signal to a thread
pthread_join	wait for a thread
pthread_self	find out own thread ID

Thread Memory Safety

■ Do not share thread-specific memory

■ Bad

```
int count = 0;
void* a = malloc(4000);
while (<waiting for connections>) {
    *a = <connection specific
           information>
    pthread_create(&tid, NULL,
                  threadfun, a);

    count++;
}
```

■ Good

```
int count = 0;
void* a;
while (<waiting for connections>) {
    a = malloc(4000);
    *a = <connection specific
           information>

    pthread_create(&tid, NULL,
                  threadfun, a);
    count++;
}
```

Shared Variables

■ Protect shared variable accesses to avoid problems

```
int globalvar;
static int globalstaticvar;

void* func(void* a) {
    int i, j;
    int localvar = 0;
    static int localstaticvar = 0;
    int temp[4];

    for (i = 0; i < 10000; i++) {
        temp[0] = localvar;
        temp[1] = localstaticvar;
        temp[2] = globalvar;
        temp[3] = globalstaticvar;

        for (j = 0; j < 4; j++)
            temp[j] += 1;
```

```
        localvar = temp[0];
        localstaticvar = temp[1];
        globalvar = temp[2];
        globalstaticvar = temp[3];
    }

    fprintf(stderr, "Local: %d  Local
Static: %d  Global: %d
Global Static:
%d\n", localvar, localstaticvar, gl
obalvar, globalstaticvar);

    pthread_exit(NULL);
}
```

Mutex Locking

- Standard Initializer

```
int pthread_mutex_init(pthread_mutex_t *mutex);
```

- Static Initializer

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
```

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

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

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

Thread Safety

- Table 12.2: Possibly unsafe POSIX functions
- `count` is a shared variable and needs to be protected from race conditions

■ Bad

```
static int count;
void increment(void) {
    count++;
}
void decrement(void) {
    count--;
}
int getcount() {
    return count;
}
```

■ Good

```
static int count = 0;
static pthread_mutex_t countlock = PTHREAD_MUTEX_INITIALIZER;

int increment(void) { /* decrement() similar */
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    count++;
    return pthread_mutex_unlock(&countlock);
}

int getcount(int *countp) {
    int error;
    if (error = pthread_mutex_lock(&countlock))
        return error;
    *countp = count;
    return pthread_mutex_unlock(&countlock);
}
```

Critical Sections

■ Program 14.1

```
for (i = 1; i < n; i++)
    if (childpid = fork())
        break;
    snprintf(buffer, BUFSIZE,
        "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
        i, (long)getpid(), (long)getppid(), (long)childpid);

    c = buffer;
    /** start of critical section **/
    while (*c != '\0') {
        fputc(*c, stderr);
        c++;
        for (i = 0; i < delay; i++)
            dummy++;
    }
    /** end of critical section **/
```

■ Why is this a critical section?

Semaphores

- “Integer value & list of processes waiting for a signal operation”
- Atomic operations
 - Logically indivisible
- `sem_wait (wait / semaphore lock / P / down)`
 - $S > 0 \rightarrow$ decrements S
 - $S == 0 \rightarrow$ blocks caller
- `sem_post (signal / semaphore unlock / V / up)`
 - Threads are blocked ($S == 0$) \rightarrow Unblocks one thread
 - No threads blocked \rightarrow Increments S

Semaphores

■ Sample implementation using Mutex locks

```
void Semaphore::wait() {
    pthread_mutex_lock(mutex);
    while ( count <= 0 ) {
        pthread_mutex_unlock(mutex);
        RelinquishProcessor();
        pthread_mutex_lock(mutex);
    }
    count--;
    pthread_mutex_unlock(mutex);
}
```

```
void Semaphore::signal() {
    pthread_mutex_lock(mutex);
    count++;
    pthread_mutex_unlock(mutex);
}
```

Semaphore Operations

- Initialize semaphore

```
int sem_init(sem_t *sem, int pshared, unsigned value);
```

- `sem` = semaphore to initialize
- `pshared = 0` → Only this process can access (usually ignored)
- `value` → Current value for semaphore (e.g. available resources)

```
if (sem_init(&semA, 0, 1) == -1)
    perror("Failed to initialize semaphore semA");
```

- Destroy semaphore

- If already destroyed or thread is blocked on it → undefined

```
if (sem_destroy(&semA) == -1)
    perror("Failed to destroy semA");
```

Semaphore Operations

```
#include <semaphore.h>
int sem_post(sem_t *sem); /*signal safe*/

int sem_trywait(sem_t *sem); /*returns -1 if
    semaphore is zero - can be interrupted by
    signal*/

int sem_wait(sem_t *sem); /* blocks if
    semaphore is zero - can be interrupted by
    signal*/
```

Allocating dynamic arrays

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

```
void *calloc(size_t nelem, size_t elsize);
```

- *nelem*
 - Number of elements
- *elsize*
 - Size of each element.
- If *nelem* or *elsize* is zero (0), a unique pointer (free-able) or NULL is returned.

Thread-safe Shared data

Program 14.6

```
sem_t semlock;
pthread_t *tids;

n = atoi(argv[1]);
tids = (pthread_t *)calloc(n, sizeof(pthread_t));
if (tids == NULL) {
    perror("Failed to allocate memory for thread IDs");
    return 1;
}
if (sem_init(&semlock, 0, 1) == -1) {
    perror("Failed to initialize semaphore");
    return 1;
}
for (i = 0; i < n; i++)
    if (error = pthread_create(tids + i, NULL, threadout, &semlock)) {
        fprintf(stderr, "Failed to create thread:%s\n",
                strerror(error));
        return 1;
    }
for (i = 0; i < n; i++)
    if (error = pthread_join(tids[i], NULL)) {
        fprintf(stderr, "Failed to join thread:%s\n", strerror(error));
        return 1;
    }
}
```

Program 14.5

```
void *threadout(void *args) {  
  
    ... \SNIP\  
  
    c = buffer;  
    /***** entry section *****/  
    while (sem_wait(semlockp) == -1)      /* Entry section */  
        if(errno != EINTR) {  
            fprintf(stderr, "Thread failed to lock semaphore\n");  
            return NULL;  
        }  
    /***** start of critical section *****/  
    while (*c != '\0') {  
        fputc(*c, stderr);  
        c++;  
        nanosleep(&sleeptime, NULL);  
    }  
    /***** exit section *****/  
    if (sem_post(semlockp) == -1)      /* Exit section */  
        fprintf(stderr, "Thread failed to unlock semaphore\n");  
    /***** remainder section *****/  
    return NULL;  
}
```

Summary

- pthreads
 - Detach to automatically release resources
 - Join to wait for thread to finish & release resources
 - Allocate memory for each thread if needed
 - Be careful with shared memory between threads
- Make sure critical sections and shared structures are protected
 - Mutex
 - Semaphore