

Creating and using threads

```
pthread_t thread;
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
                  void *(*start)(void *), void *arg);
```

```
pthread_t pthread_self(void);
```

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

```
int pthread_detach(pthread_t thread);
```

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

Example

```
/*
 * lifecycle.c
 * Demonstrate the "life cycle" of a typical thread. A thread is
 * created, and then joined.
 */
#include <pthread.h>
#include "errors.h"

/* Thread start routine. */

void *thread_routine (void *arg)
{
    return arg;
}
```

```
main (int argc, char *argv[])
{
    pthread_t thread_id;
    void *thread_result;
    int status;

    status = pthread_create (
        &thread_id, NULL, thread_routine, NULL);
    if (status != 0)
        err_abort (status, "Create thread");
    status = pthread_join (thread_id, &thread_result);
    if (status != 0)
        err_abort (status, "Join thread");
    if (thread_result == NULL)
        return 0;
    else return 1;
}
```

Creating and using threads

- Thread states
 - Ready
 - Running
 - Blocked
 - Terminated
- the main thread is special
- detaching a thread has no impact on a running thread except the system to know that it can free up resources being used by that thread when it terminates

Example: using threads

```
#include <pthread.h>
#include "errors.h"

typedef struct alarm_tag {
    int        seconds;
    char       message[64];
} alarm_t;

void *alarm_thread (void *arg)
{
    alarm_t *alarm = (alarm_t*)arg;
    int status;

    status = pthread_detach (pthread_self ());
```

```

    if (status != 0)
        err_abort (status, "Detach thread");
    sleep (alarm->seconds);
    printf ("%d) %s\n", alarm->seconds, alarm->message);
    free (alarm);
    return NULL;
}
int main (int argc, char *argv[])
{
    int status;
    char line[128];
    alarm_t *alarm;
    pthread_t thread;

    while (1) {
        printf ("Alarm> ");
        if (fgets (line, sizeof (line), stdin) == NULL) exit (0);
        if (strlen (line) <= 1) continue;

```

```

alarm = (alarm_t*)malloc (sizeof (alarm_t));
if (alarm == NULL)
    errno_abort ("Allocate alarm");

/*
 * Parse input line into seconds (%d) and a message
 * (%64[^\n]), consisting of up to 64 characters
 * separated from the seconds by whitespace.
 */
if (sscanf (line, "%d %64[^\n]",
    &alarm->seconds, alarm->message) < 2) {
    fprintf (stderr, "Bad command\n");
    free (alarm);
} else {
    status = pthread_create (
        &thread, NULL, alarm_thread, alarm);
    if (status != 0)
        err_abort (status, "Create alarm thread");
}

```

Solaris threads

- Solaris supports both the POSIX API and “Solaris” API
- Programs written using POSIX threads are more portable
- Multithreaded Programming Guide examples use Solaris API
- document on POSIX API available; man page for a call describes both APIs

Solaris Synchronization Primitives

- Mutex Locks
- Condition Variables
- Reader/Writer Locks
- Semaphores

Mutex Locks

- Mutual Exclusion Locks
- Example:

```
pthread_mutex_t count_mutex = PTHREAD_MUTEX_INITIALIZER;  
int count;
```

```
increment_count()  
{  
    pthread_mutex_lock(&count_mutex);  
    count = count + 1;  
    pthread_mutex_unlock(&count_mutex);  
}
```

```
get_count()  
{  
    int c;  
    pthread_mutex_lock(&count_mutex);  
    c = count;  
    pthread_mutex_unlock(&count_mutex);  
    return(c);  
}
```

Condition Variables

- based on **monitor** condition variables
- Easier to understand and use than semaphores
- `cond_wait` and `cond_signal` operations
 - different semantics from semaphore wait and signal operations
- always use in conjunction with a mutex lock

Producer-Consumer using condition variables

```
char buf[BSIZE];
int occupied;
int nextin;
int nextout;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

producer(char item)
{
    pthread_mutex_lock(&mutex);
    while (occupied == BSIZE)
        pthread_cond_wait(&not_full, &mutex);

    /* insert item */
    buf[next_in++] = item;
    next_in = next_in % BSIZE;
    occupied++;
    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);
}
```

```
consumer()
{
    pthread_mutex_lock(&mutex);
    while (occupied == 0)
        pthread_cond_wait(&not_empty, &mutex);

    /* consume item */
    item = buf[next_out++];
    next_out = next_out % BSIZE;
    occupied--;
    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&mutex);
}
```