PThreads

A Note About Makefiles

- In OneFS/BSD be sure to build using `gmake`
- You may have to remove `-pedantic-errors` from `CFLAGS`

Creating/Starting a Thread (1 of 4)

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

`thread`

Pointer to a private thread ID.

`attr`

Pointer to an thread attributes block; may be `NULL`.

`start_routine`

Pointer to a function to control the thread.

`arg`

Argument passed to `start_routine`.

Creating/Starting a Thread (2 of 4)

Returns:

0 if successful, non-zero otherwise.

`pthread_create` creates and launches a thread by calling `start_routine` and passing `arg`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>

static void *thread_proc( void * );
```

Creating/Starting a Thread (3 of 4)

```
int main( int argc, char **argv )
{
    pthread_t   thread;
    int stat =
        pthread_create( &thread, NULL, thread_proc, NULL );
    if ( stat != 0 )
    {
        const char  *reason = strerror( stat );
        fprintf( stderr,
                 "thread create failure, %d: \"%s\"",
                 stat,
                 reason
               );
        abort();
    }
    stat = pthread_join( thread, NULL );
    if ( stat != 0 )
    {
        . . .
    }
    return 0;
}
```

Creating/Starting a Thread (4 of 4)

```
static void *thread_proc( void *arg )
{
    for ( int inx = 0 ; inx < 2 ; ++inx )
    {
        puts( "thread" );
        sleep( 2 );
    }

    return NULL;
}
```

Waiting For Thread Completion

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **retval);
```

`thread`
: Thread ID.

`retval`
: Pointer for returning thread termination value; may be `NULL`.

Returns:
: 0 if successful, non-zero otherwise.

Returns immediately if thread has already expired, otherwise returns after thread expiration.

The PTH Module

```
#include <pth.h>
void PTH_create(
    pthread_t             *thread,
    const pthread_attr_t *attr,
    PTH_PROC_p_t          start_routine,
    void                  *arg
);
void PTH_join(
    pthread_t thread,
    void **retval
);
void PTH_err(
    int         status,
    const char *msg,
    const char *file,
    int line
);
```

All raise SIGABRT on error; PTH_err raises SIGABRT unconditionally.

See also:

```
pthread_tryjoin_np
pthread_attr_init
pthread_attr_destroy
pthread_getattr_np
pthread_attr_*
```

Exercises

1. Download the `pth` module from the class web site. Complete the functions `PTH_create` and `PTH_join`. Add the module to `libisi.a`.
2. Download `thr_test1.c` from the class web site. Complete the program so that `main` creates two threads based on `proc1` and `proc2`, and then joins the threads. Run the program.
3. For this exercise you will revise exercise 1 so that it runs both threads based on a single `start_routine`. Download `thr_test2.c` from the class web site. Complete `main` and `proc` according to the enclosed instructions. Run the program.
4. Download `thr_test3.c` from the class web site. Complete `main` according to the enclosed instructions. Run the program, redirecting `stdout` to a text file; `grep` the text file for lines containing "invalid." What went wrong?

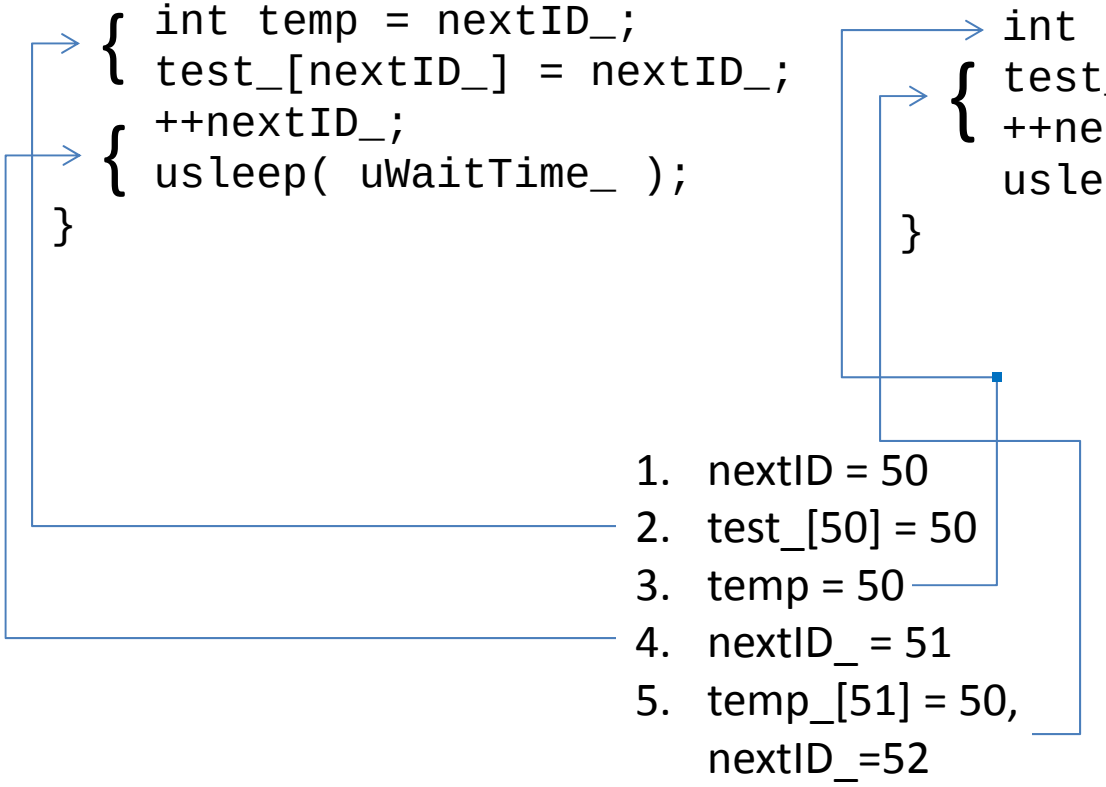
Race Conditions

THREAD1

```
while ( nextID_ < TEST_LEN )
{
    int temp = nextID_;
    test_[nextID_] = nextID_;
    ++nextID_;
    usleep( uWaitTime_ );
}
```

THREAD2

```
while ( nextID_ < TEST_LEN )
{
    int temp = nextID_;
    test_[nextID_] = nextID_;
    ++nextID_;
    usleep( uWaitTime_ );
}
```



1. nextID = 50
2. test_[50] = 50
3. temp = 50
4. nextID_ = 51
5. temp_[51] = 50, nextID_=52

Managing Race Conditions: Mutexes

```
#include <pthread.h>
typedef pthread_mutex_t …
#define 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);
```

`pthread_mutex_lock`

Locks a mutex. If the mutex is already locked, the thread is suspended until it is unlocked.

`pthread_mutex_trylock`

Trys to lock a mutex. If the mutex is already locked, returns a value of EBUSY.

`pthread_mutex_unlock`

Unlocks a mutex.

See also: machine/atomic.h for freebsd

See also:

```
pthread_mutex_destroy
pthread_mutex_timedlock
pthread_mutexattr_init
pthread_mutexattr_destroy
pthread_mutexattr_*
```

Using a Mutex

```
static pthread_mutex_t  mutex_          =
    PTHREAD_MUTEX_INITIALIZER;
static int              store_inx_      = 0;
static int              storage_[100];

// May be called from multiple threads
static void store_val( int val )
{
    pthread_mutex_lock( &mutex_ );
    if ( store_inx_ < ISI_CARD( storage_ ) )
        storage_[store_inx_++] = val;
    pthread_mutex_unlock( &mutex );
}
```

For dynamic initialization use pthread_mutex_init

Exercises

5. Revise `thr_test3.c` so that it uses a mutex to fix the race condition.

Threading 1-Step Processes For Multiple Clients

```
do
{
    client = nextClient()
    if ( client != NULL )
    {
        dispatch client via new thread
        store thread
    } while ( client != NULL )
sequentially join all threads
```

Threading N-Step Processes

```
wait for new component
post component to step 1 queue
```

Initiator thread

```
wait for next component
process component
post component to step 2 queue
```

Step 1 thread

```
wait for next component
process component
post component to step 3 queue
```

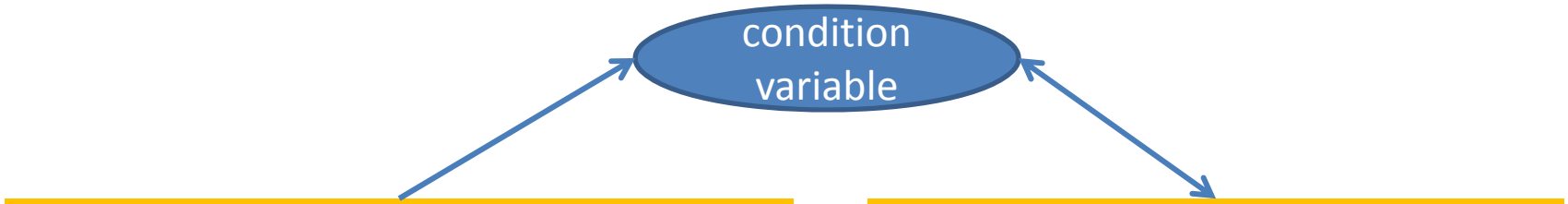
Step 2 thread

. . .

```
wait for next component
dispose component
```

Finalizer thread

Waiting: Condition Variables



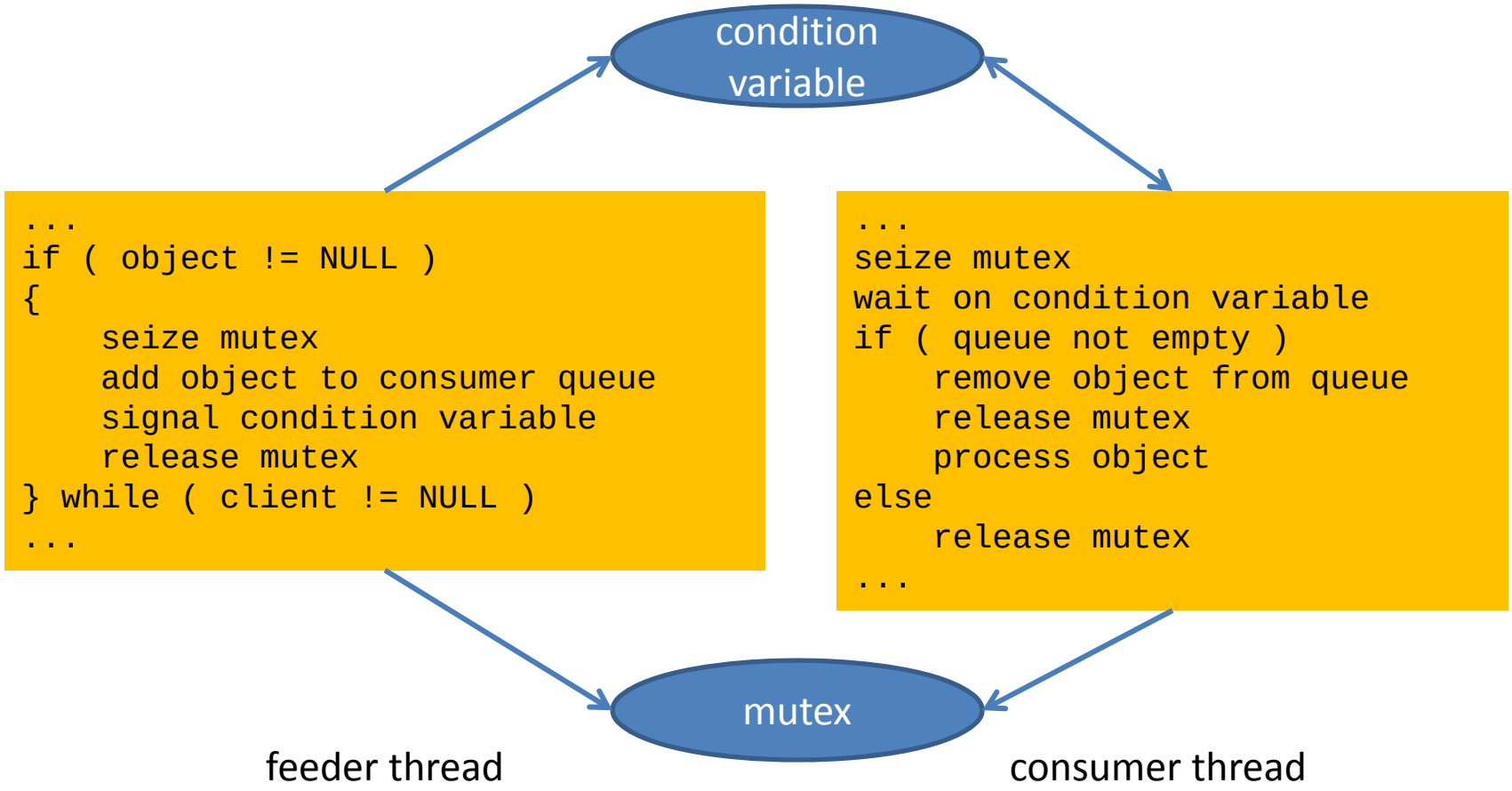

```
create consumer threads
do
{
    object = nextObject()
    if ( object != NULL )
    {
        add object to consumer queue
        signal condition variable
    } while ( client != NULL )
signal termination
sequentially join all threads
```

feeder thread

```
while ( !termination )
{
    wait on condition variable
    if ( queue not empty )
        remove object from queue
        process object
}
```

consumer thread

Synchronizing Condition Variable Usage



pthread_cond_wait

```
int pthread_cond_wait(
    pthread_cond_t  *cond,
    pthread_mutex_t *mutex
);
```

- Call wait function with `mutex` locked
- If cond has been signaled wait returns immediately...
- ... else thread is suspended and `mutex` is unlocked
- Thread is always wakened with `mutex` locked

pthread_cond_timedwait

```
int pthread_cond_timedwait(
    pthread_cond_t         *cond,
    pthread_mutex_t        *mutex,
    const struct timespec *abstime
);
```

- Like `pthread_condition_wake` except may return `ETIMEDOUT`
- `abstime` is the time to wake the thread, *not* an interval

struct timespec

```
#include <time.h>
struct timespec
{
    time_t tv_sec;   /* seconds */
    long   tv_nsec;  /* nanoseconds */
};
```

`tv_nsec` must be in the range 0 to 999,999,999

pthread_cond_signal

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_signal`: If more than one thread is waiting on cond, only one is wakened.
- `pthread_cond_broadcast`: All threads waiting on cond are wakened.

Condition Variable Example (1 of 8)

```
typedef struct control_s
{
    pthread_mutex_t     mutex;
    pthread_cond_t      cond;
    int                 ident;
} CONTROL_t, *CONTROL_p_t;

static void *thrProc( void * );

static int          inProgress_ = 1;
static CONTROL_t    controls_[2];
```

Condition Variable Example (2 of 8)

```
int main( int argc, char **argv )
{
   pthread_t   thr1;
   pthread_t   thr2;

   PTH_mutex_init( &controls_[0].mutex, NULL );
   PTH_cond_init( &controls_[0].cond, NULL );
   controls_[0].ident = 1;
   PTH_create( &thr1, NULL, thrProc, &controls_[0] );

   PTH_mutex_init( &controls_[1].mutex, NULL );
   PTH_cond_init( &controls_[1].cond, NULL );
   PTH_create(&thr2, NULL, thrProc, &controls_[1]);
   controls_[1].ident = 2;
   puts( “start” );
   . . .
```

Condition Variable Example (3 of 8)

```
int main( int argc, char **argv )
{
    . . .
    for ( int inx = 0 ; inx < 5 ; ++inx )
    {
        sleep( 2 );
        pthread_mutex_lock( &controls_[0].mutex );
        pthread_cond_signal( &controls_[0].cond );
        pthread_mutex_unlock( &controls_[0].mutex );
        if ( inx %2 == 0 )
        {
            PTH_mutex_lock( &controls_[1].mutex );
            PTH_cond_signal( &controls_[1].cond );
            PTH_mutex_unlock( &controls_[1].mutex );
        }
    }
    . . .
```

Condition Variable Example (4 of 8)

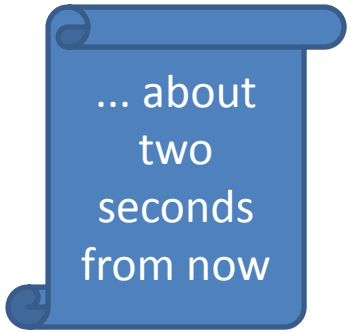
```
    . . .
    inProgress_ = 0;
    PTH_join( thr1, NULL );
    PTH_join( thr2, NULL );
    puts( "end" );

    return 0;

}
```

Condition Variable Example (5 of 8)

```
static void *thrProc( void *arg )
{
    CONTROL_p_t control = arg;
    . . .
```

Condition Variable Example (6 of 8)

```
. . .
while ( inProgress_ )
{
    struct timespec waitTime = { time( NULL ) + 2 , 0 };

    PTH_mutex_lock( &control->mutex );
    errno = 0;
    PTH_cond_timedwait( &control->cond,
                        &control->mutex,
                        &waitTime
                      );
    const char *msg =
        errno == ETIMEDOUT ? "timeout" : "signal";
    printf("Thread %d: %s\n", control->ident, msg );
    PTH_mutex_unlock( &control->mutex );
}
. . .
```



Condition Variable Example (7 of 8)

```
    . . .
    printf( "Thread %d exiting\n", control->ident );

    return NULL;
}
```

Condition Variable Example (8 of 8)

```
$ ./thr_test5
start
Thread 1: timeout
Thread 2: timeout
Thread 1: signal
Thread 2: signal
Thread 1: timeout
Thread 2: timeout
Thread 1: signal
Thread 2: timeout
. . .
Thread 1: timeout
Thread 2: timeout
Thread 1: signal
Thread 2: timeout
Thread 1: timeout
Thread 2: timeout
Thread 1: signal
Thread 1 exiting
Thread 2: signal
Thread 2 exiting
end
```

See also:

```
pthread_cond_destroy
pthread_condattr_init
pthread_condattr_destroy
pthread_condattr_*
```

Exercises

6. Download `cond_exercise.c` from the class website. This program uses three threads to (1) obtain an item to process (`thrGet/procGet`), (2) process the item (`thrOne/procOne`), and (3) finalize the processing (`thrFin/procFin`). `thrGet` posts the item to `anchorOne`, synchronizing via `mutexOne` and `condOne`. When ready, `thrOne` posts the item to `anchorFin`, synchronizing via `mutexFin` and `condFin`. Add another step to this process. Create an anchor, mutex and condition variable for `thrTwo/procTwo`. Change `procOne` so that it posts to the queue for `thrTwo`, and write `procTwo` so that it processes the item (via `work()`) and posts it to `anchorFin`. Create a new thread in `main` to drive `thrTwo`.
7. Modify `cond_exercise.c`. Add an object (`struct`) that can contain an anchor, mutex and condition variable. Use this object to encapsulate the anchor, mutex and condition variable for each of the four threads. You will probably have to initialize the mutex and condition variable via PTH_mutex_init and PTH_cond_init.
8. Design an object that can encapsulate an anchor, mutex, condition variable, pointer to thread function, thread ID, process ID (type `int`) and a `struct timespec` object.

Synchronizing via Signals

Threads can communicate via signals.

- The signals to communicate with must be *blocked*; use `pthread_sigmask`.
- Signal the thread using `pthread_kill`.
- The thread waits for the signal using `sigwait` or `sigwaitinfo`.

pthread_sigmask (1)

```
#include <pthread.h>
#include <signal.h>
int pthread_sigmask(
    int             how,
    const sigset_t *set,
    sigset_t       *oldset
);
```

how
: use SIG_BLOCK to block, SIG_UNBLOCK to unblock; may be NULL.

set
: signals affected by this call.

oldset
: previous signal mask; may be NULL.

pthread_sigmask (2)

```
sigset_t    mask;
sigemptyset( &mask );
sigaddset( &mask, SIGHUP );
sigaddset( &mask, SIGUSR1 );
PTH_sigmask( SIG_BLOCK, &mask, NULL );
```

pthread_kill

```
#include <pthread.h>
#include <signal.h>
int pthread_kill(
    pthread_t thread,
    int       sig
);
```

`thread`
: the thread to signal.

`sig`
: the signal to send.

sigwait (1)

```
#include <signal.h>
int sigwait(
    sigset_t set,
    int      *sig
);
```

`set`

the signals to wait on.

`sig`

the signal that ended the wait.

`return`

0 for success, error number otherwise.

sigwait (2)

```
sigset_t    sigset;
sigemptyset( &sigset );
sigaddset( &sigset, SIGHUP );
sigaddset( &sigset, SIGUSR1 );

int stat    = sigwait( &sigset, &sig );
if ( stat != 0 )
{
    const char  *msg    = strerror( stat );
    fprintf( stderr, "sigwait: \"%s\"\n", msg );
    abort();
}
```

`sigwait` (3)

- If a signal is pending returns immediately
- If a signal has been posted more than once when `sigwait` is called, behavior is system-dependent: one signal may be cleared or all signals may be cleared

See also:

`sigwaitinfo`
`sigtimedwait`

sigwait Example(1)

```
static void init( void )
{
    sigset_t    mask;
    sigemptyset( &mask );
    sigaddset( &mask, SIGHUP );
    sigaddset( &mask, SIGUSR1 );
    PTH_sigmask( SIG_BLOCK, &mask, NULL );
}
```

sigwait Example(2)

```
static void *thrProc( void *arg )
{
    sleep( 2 );
    sigset_t    sigset;
    int         sig     = SIGUSR1;
    int         count   = 0;

    sigemptyset( &sigset );
    sigaddset( &sigset, SIGHUP );
    sigaddset( &sigset, SIGUSR1 );
    . . .
```

sigwait Example(3)

```
    . . .
    do
    {
        int stat    = sigwait( &sigset, &sig );
        if ( stat != 0 )
        {
            const char  *msg    = strerror( stat );
            fprintf( stderr, "%s\n", msg );
            abort();
        }
        if ( sig == SIGUSR1 )
            printf( "SIGUSR1: %d\n", ++count );
        if ( sig == SIGHUP )
            printf( "SIGHUP\n" );
    } while ( sig == SIGUSR1 );

    return NULL;
}
```

sigwait Example(4)

```
int main( int argc, char **argv )
{
    init();
    pthread_t   thr;
    printf( "start\n" );
    PTH_create( &thr, NULL, thrProc, NULL );
    PTH_kill( thr, SIGUSR1 );
    PTH_kill( thr, SIGUSR1 );
    PTH_kill( thr, SIGUSR1 );
    sleep( 3 );
    PTH_kill( thr, SIGUSR1 );
    sleep( 3 );
    PTH_kill( thr, SIGUSR1 );
    PTH_kill( thr, SIGUSR1 );
    sleep( 3 );
    PTH_kill( thr, SIGHUP );
    PTH_join( thr, NULL );
    printf( "end\n" );

    return 0;
}
```

sigwait Example(5)

```
$ ./thr_sigwait
start
SIGUSR1: 1
SIGUSR1: 2
SIGUSR1: 3
SIGHUP
end
```

Exercises

9. Download `sigwait_exercise.c`. Complete the code as instructed in the file.

Read/Write Locks

For a resource, a read/write lock:

- May be seized for read or write access, and must subsequently be released
- Maintains a read count and a write count
- When write count is 0, unlimited read access is allowed
- When write count is 1, all access is blocked until the read count is 0, then the write access is allowed

pthread_rwlock_...

```
pthread_rwlock_t rwlock =
    PTHREAD_RWLOCK_INITIALIZER;
int pthread_rwlock_init(
    pthread_rwlock_t    *rwlock,
    pthread_rwlockattr_t *attr
);
int pthread_rwlock_rdlock(
    pthread_rwlock_t *rwlock
);
int pthread_rwlock_tryrdlock(
    pthread_rwlock_t *rwlock
);
int pthread_rwlock_wrlock(
    pthread_rwlock_t *rwlock
);
int pthread_rwlock_unlock(
    pthread_rwlock_t *rwlock
);
```

PTH_rwlock_...

```
. . .
void PTH_rwlock_tryrdlock(
    pthread_rwlock_t *rwlock
);
```

If the operation fails with a status of EPERM, errno is set to EPERM.

Read/Write Lock Example (1)

```
typedef struct control_s
{
    int         pauseBefore;
    int         pauseAfter;
    int         type;
    int         ident;
    pthread_t   thrID;
} CONTROL_t, *CONTROL_p_t;
static pthread_rwlock_t rwlock  =
    PTHREAD_RWLOCK_INITIALIZER;
. . .
```

Read/Write Lock Example (2)

```
CONTROL_t   threads[]   =
{
    { 0000, 4000, TYPE_READ,  0 },
    { 1000, 4000, TYPE_READ,  0 },
    { 2000, 4000, TYPE_READ,  0 },
    { 3000, 5000, TYPE_WRITE, 0 },
    { 4000, 2000, TYPE_READ,  0 },
    { 5000, 2000, TYPE_READ,  0 },
};

for ( int inx = 0 ;
      inx < ISI_CARD( threads ) ; ++inx )
{
    threads[inx].ident = inx;
    PTH_create( &threads[inx].thrID, NULL,
                    driver, &threads[inx] );
}
```

Read/Write Lock Example (3)

```
static void *driver( void *arg )
{
    CONTROL_p_t contr   = arg;
    pause( contr->pauseBefore );
    switch ( contr->type )
    {
        case TYPE_READ:
            printf( “... rdlock ...” );
            PTH_rwlock_rdlock( &rwlock );
            break;
        case TYPE_WRITE:
            printf( “... write lock ...” );
            PTH_rwlock_wrlock( &rwlock );
            break;
    }
    printf( “.. lock acquired ...” );
    pause( contr->pauseAfter );
    printf( “... unlocking\n”... );
    PTH_rwlock_unlock( &rwlock );
```

Read/Write Lock Example (4)

```
$ ./rwlock_test2
thread 0 attempting read_lock
thread 0 lock acquired
   thread 1 attempting read_lock
   thread 1 lock acquired
      thread 2 attempting read_lock
      thread 2 lock acquired
         thread 3 attempting write_lock
            thread 4 attempting read_lock
thread 0 unlocking
   thread 1 unlocking
               thread 5 attempting read_lock
      thread 2 unlocking
         thread 3 lock acquired
         thread 3 unlocking
            thread 4 lock acquired
               thread 5 lock acquired
               thread 5 unlocking
            thread 4 unlocking
```

Preventing Writer Starvation (1)

When read-locks and write-locks are blocked, priority is given to write-locks when unblocked.

```
CONTROL_t   threads[]   =
{
    { 0000, 4000, TYPE_WRITE, 0 },
    { 1000, 4000, TYPE_READ,  0 },
    { 2000, 4000, TYPE_READ,  0 },
    { 3000, 5000, TYPE_WRITE, 0 }
};
```

Preventing Writer Starvation (2)

```
$ ./rwlock_test3
thread 0 attempting write_lock
thread 0 lock acquired
   thread 1 attempting read_lock
      thread 2 attempting read_lock
         thread 3 attempting write_lock
thread 0 unlocking
         thread 3 lock acquired
         thread 3 unlocking
      thread 2 lock acquired
   thread 1 lock acquired
      thread 2 unlocking
   thread 1 unlocking
```

nanosleep

```
int nanosleep(
    struct timespec *req,
    struct timespec *rem
);
struct timespec {
    time_t tv_sec;        /* seconds */
    long   tv_nsec;       /* nanoseconds */
};
```

Sleeps for the number of seconds/nanoseconds indicated by `req`. If the operation is interrupted, `errno` is set to `EINTR`. and the remaining (unelapsed) time is returned in `rem` (if it is non-NULL).

See also:

```
pthread_rwlock_destroy
pthread_rwlockattr_init
pthread_rwlockattr_destroy
pthread_rwlockattr_*
```

Exercises

10. Write a program that launches eleven threads; ten of them should wait one quarter of a second; lock a read-write mutex for reading; wait another quarter of a second; release the mutex; and repeat. The eleventh should wait two seconds; lock the mutex for writing; wait another quarter of a second; release the mutex; and repeat. Each thread should keep a count of the number of times a lock succeeded. Use a global flag to notify your threads when to quit. When all of the threads have been terminated, your main thread should print out the counts for each thread. The program should run for 15 seconds.

    Suggestions: a) write two start-routines, one for the readers and one for the writer. b) use a `struct` to store the thread counts and thread IDs; use a pointer to the `struct` as the start-routine argument.

Spin Locks

- A spin lock is a compute-intensive type of lock
- Spin locks are intended for multiprocessor environments
- Spin locks are rarely used in application-level code
- Spin locks should be used with great care

```
static volatile bool available_;
while ( !available_ )
    ;
available = false;
execute();
```

See also:

```
void PTH_spin_init(
    pthread_spinlock_t *lock,
    int                pshared
);
void PTH_spin_destroy(
    pthread_spinlock_t *destroy
);
void PTH_spin_lock(
    pthread_spinlock_t *lock
);
void PTH_spin_unlock(
    pthread_spinlock_t *lock
);
```

Critical Sections

- A critical section is a block of code that must be executed by at most one thread at a time
- Critical sections can be implemented using mutexes
- Code in critical sections should *not* perform blocking operations

```
static pthread_mutex_t  critSecMutex_   =
    PTHREAD_MUTEX_INITIALIZER;

void thread_funk( void )
{
    PTH_mutex_lock( &critSecMutex_ );
    . . .
    PTH_mutex_unlock( &critSecMutex_ );
}
```

Thread Management – Pay As You Go

- Create a new thread every time you need one
- Thread must be detached
- Once a thread is detached it can't be joined

```
while ( !shutDown_ )
{
    pthread_t thread;
    DATA_    *data = nextClient();
    PTH_create( &thread, NULL, driver, data );
    PTH_detach( thread );
}
```

pthread_detach

```
int pthread_detach(
    pthread_t thread_id
);
```

Normally thread resources for a thread are not released until the thread is joined. Use detach when you don't expect to join with a thread.

- You cannot join a detached thread.
- You cannot communicate with the thread via its ID.
- All thread resources are freed when the thread exits.

Problems With This Approach

- Since threads can't be joined you need a way to know when all threads have exited.
- You may need an infinite supply of threads

```
while ( !shutDown_ )
{
    pthread_t thread;
    DATA_    *data = nextClient();
    PTH_create( &thread, NULL, driver, data );
    PTH_detach( thread );
}
```

Tracking Detached Thread Progress

- Since threads can't be joined you need a way to know when all threads have exited.
- You may need an infinite supply of threads

```
static int num_threads_    = 0;
. . .
while ( (data = nextClient()) == NULL )
{
    nextThread( data );
}

while ( num_threads_ > 0 )
    pauseMillis( 125 );
```

Governing Thread Creation (1 of 3)

```
static void nextThread( int *arg )
{
    PTH_mutex_lock( &mutex_ );
    ISI_ASSERT( num_threads_ <= MAX_THREADS );
    ISI_ASSERT( num_threads_ >= 0 );

    bool    done    = false;
    while ( !done )
    {
        if ( num_threads_ == MAX_THREADS )
        {
            printf( "waiting...\n" );
            PTH_cond_wait( &cond_, &mutex_ );
        }
    . . .
```

Governing Thread Creation (2 of 3)

```
        . . .
            if ( num_threads_ < MAX_THREADS )
            {
                pthread_t   thread;
                PTH_create(&thread, NULL, driver, arg);
                PTH_detach( thread );
                ++num_threads_;
                done = true;
            }
        }
        PTH_mutex_unlock( &mutex_ );
    }
```

Governing Thread Creation (3 of 3)

```
static void *driver( void *arg )
{
    int *ident  = arg;
    printf( "thread %2d starting\n", *ident );
    pauseMillis( 500 );

    PTH_mutex_lock( &mutex_ );
    --num_threads_;
    PTH_cond_signal( &cond_ );
    PTH_mutex_unlock( &mutex_ );

    printf( "thread %2d terminating\n", *ident );

    return NULL;
}
```

Problems With This Implementation

- The main thread may be blocked for a long time.

Solution:

- Create a separate thread to dispatch worker threads.
- The main (or some other) thread simply enqueues clients on a dispatch queue.

```
while ( !shutDown_ )
{
    pthread_t thread;
    DATA_    *data = nextClient();
    PTH_create( &thread, NULL, driver, data );
    PTH_detach( thread );
}
```

Problems With This Implementation

- The main thread may be blocked for a long time.

Solution:

- Create a separate thread to dispatch worker threads.
- The main (or some other) thread simply enqueues clients on a dispatch queue.

```
while ( !shutDown_ )
{
    pthread_t thread;
    DATA_    *data = nextClient();
    PTH_create( &thread, NULL, driver, data );
    PTH_detach( thread );
}
```

Exercises

11. Download `pay_as_you_go.c` from the class website. Modify it so that it creates a dispatcher thread which takes client data from a queue and uses it to create and detach a new thread.

    Suggestions:

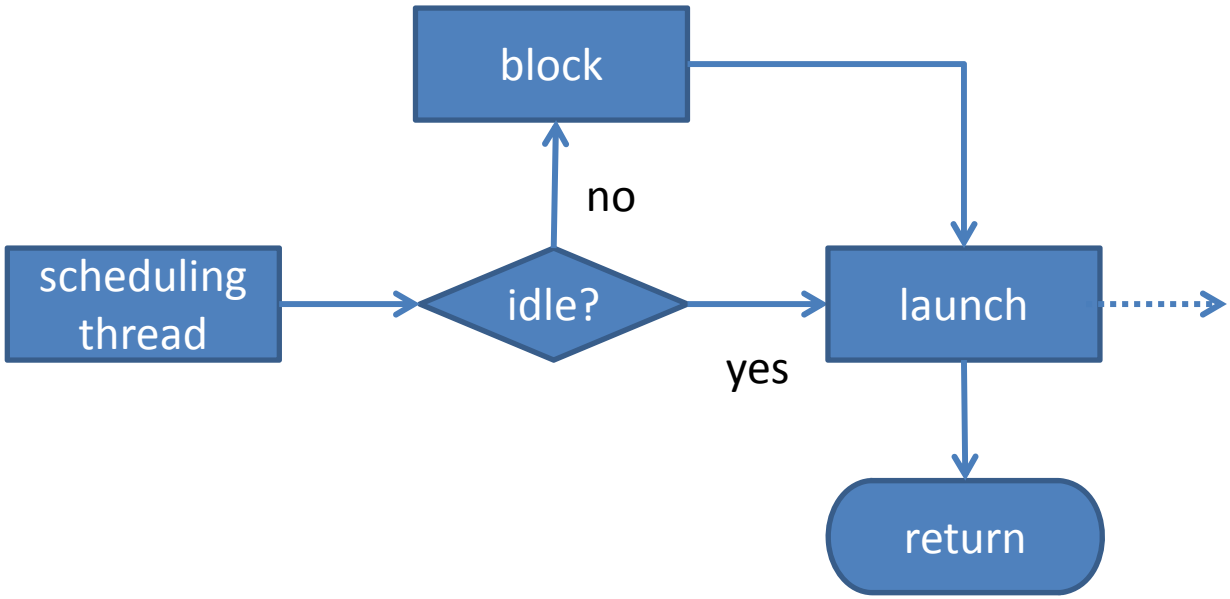
    a) Create an object that contains an anchor, mutex, condition variable and shutdown flag. Pass an object of this kind as the arg of the dispatcher thread.
    b) Create a subclass of `ENQ_ITEM_t` that contains a field for client data.
    c) After the while loop in `main` terminates, set the shutdown flag and join with the dispatcher thread.
    d) In the dispatcher thread create a loop that runs as long as the shutdown flag isn't set or the queue isn't empty.
    e) After the loop in the dispatcher thread terminates wait for all threads to complete then exit.

Thread Management – Thread Pools

- Designate a fixed number of persistent threads to manage all tasks
- Eliminates overhead associated with thread “churn”
- Two basic models: *task-blocked* vs. *thread-blocked*

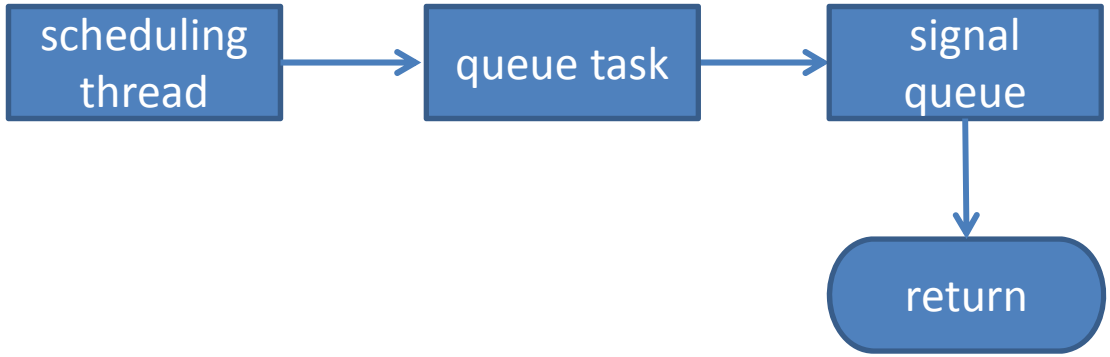
Task-Blocked Thread Pools

- Idle threads reside on a stack
- A new task is assigned to thread at top of the stack
- If the stack is empty, the calling thread is blocked
- Advantage: efficient
- Disadvantage: calling thread may be blocked



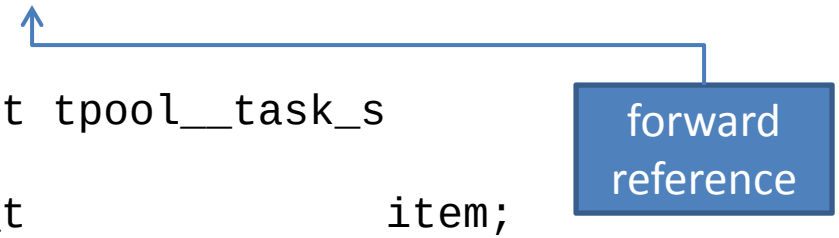
Thread-Blocked Thread Pools

- Idle threads block on a queue
- New task is queued, queue is signaled
- Pool manager immediately returns
- Advantage: calling thread is never blocked
- Disadvantage: less efficient



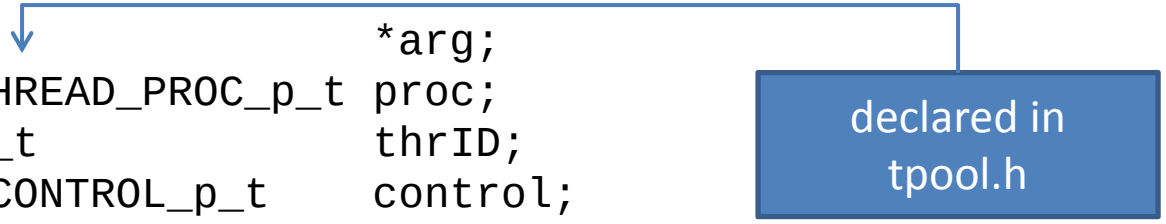
Thread Pool API: `tpoolp.h` (1 of 2)

```
typedef struct tpool__control_s
    *TPOOL__CONTROL_p_t;


typedef struct tpool__task_s
{
    ENQ_ITEM_t              item;
    TPOOL_THREAD_PROC_p_t   proc;
    void                    *arg;
} TPOOL__TASK_t, *TPOOL__TASK_p_t;
```



Thread Pool API: `tpoolp.h` (2 of 2)

```
typedef struct tpool__thread_block_s
{
    void                    *arg;
    TPOOL_THREAD_PROC_p_t proc;
    pthread_t               thrID;
    TPOOL__CONTROL_p_t    control;
} TPOOL__THREAD_BLOCK_t, *TPOOL__THREAD_BLOCK_p_t;

typedef struct tpool__control_s
{
    int                         numThreads;
    ENQ_ANCHOR_p_t              tasks;
    TPOOL__THREAD_BLOCK_p_t threads;
    pthread_mutex_t             flagMutex;
    pthread_mutex_t             taskMutex;
    pthread_cond_t              taskCond;
    bool                        quiesce;
} TPOOL__CONTROL_t;
```



Thread Pool API: `tpool.h`

```
#define TPOOL_NULL_ID   (NULL)

typedef struct tpool__control_s *TPOOL_ID_t;

typedef void *TPOOL_THREAD_PROC_t( void * );
typedef TPOOL_THREAD_PROC_t *TPOOL_THREAD_PROC_p_t;
```

Thread Pool API: Methods (1 of 2)

```
TPOOL_ID_t TPOOL_create(int numThreads );
```

Create a thread pool of size `numThreads`.

```
TPOOL_ID_t TPOOL_destroy(TPOOL_ID_t  pool );
```

Destroy a thread pool.

```
void TPOOL_add_task(
    TPOOL_ID_t              pool,
    TPOOL_THREAD_PROC_p_t   proc,
    void                    *arg
);
```

Add a task to the task queue.

Thread Pool API: Methods (2 of 2)

```
void TPOOL_quiesce(TPOOL_ID_t pool );
```
Begin gradual shutdown; threads exit when all tasks are complete.

```
bool TPOOL_has_task(TPOOL_ID_t pool );
```
Returns true if the task queue is empty.

```
void TPOOL_join( TPOOL_ID_t pool );
```
Returns after every thread in the pool has been joined.

See Also

`threadpool.tgz`
On the class web site.
Author:
Tomer Heber
http://sourceforge.net/projects/cthreadpool/

`gdb`:
`info thread` command
`break` command, `thread` parameter:

```
(gdb) b 37 thread 4
```