### Thread Programming

**Topics**
- Thread concept
- Posix threads (Pthreads) interface

### Thread Packages and Languages

A thread package is a collection of primitives related to threads:
- Thread creation and termination primitives
- Thread synchronization primitives. e.g.
  - lock provides mutually exclusive access to data structure
  - condition variable allows a thread to block until a condition is satisfied.

- Posix threads
- Solaris threads
- Java Threads
Posix threads (Pthreads) interface

*Pthreads*: Standard interface for ~60 functions that manipulate threads from C programs.

- Creating and reaping threads.
  - `pthread_create`
  - `pthread_join`
- Determining your thread ID
  - `pthread_self`
- Terminating threads
  - `pthread_cancel`
  - `pthread_exit`
  - `exit()` [terminates all threads], `ret` [terminates current thread]
- Synchronizing access to shared variables
  - `pthread_mutex_init`
  - `pthread_mutex_[u]nlock`
  - `pthread_cond_init`
  - `pthread_cond_[timed]wait`

The Pthreads "hello, world" program

```c
/*
 * hello.c - Pthreads "hello, world" program
 *
#include <pthread.h>

void *thread(void *vargp);

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```

Thread attributes (usually NULL)
Thread arguments (void *)
return value (void *)
Execution of “hello, world”

- create peer thread
- wait for peer thread to terminate
- print output
- terminate thread via `ret`
- `exit()` terminates main thread and any peer threads

Basic thread control: create a thread

```
int pthread_create(pthread_t *tidp, pthread_attr_t *attrp,
                   void *(*routine)(void *), void *argp);
```

- Creates a new peer thread
  - `tidp`: thread id
  - `attrp`: thread attributes (usually NULL)
  - `routine`: thread routine
  - `argp`: input parameters to routine

Akin to `fork()`
- but without the confusing “call once return twice” semantics.
- peer thread has local stack variables, but shares all global variables.
Basic thread control: join

```c
int pthread_join(pthread_t tid, void **thread_return);
```

Waits for a specific peer thread to terminate, and then reaps it.
- `tid`: thread ID of thread to wait for.
- `thread_return`: object returned by peer thread via `ret stmt`

Akin to `wait` and `wait_pid` but unlike `wait`...
- Any thread can reap any other thread (not just children)
- Must wait for a "specific" thread
  - no way to wait for "any" thread.
  - perceived by some as a flaw in the Pthreads design

Pthreads wrappers

We advocate Steven’s convention of providing wrappers for each system-level function call.
- wrapper is denoted by capitalizing first letter of function name.
- wrapper has identical interface as the original function.
- each wrapper does appropriate unix or posix style error checking.
- wrapper typically return nothing.
- declutters code without compromising safety.

```c
/*
* wrapper function for pthread_join
*/
void Pthread_join(pthread_t tid, void **thread_return) {
    int rc = pthread_join(tid, thread_return);
    if (rc != 0) {
        printf("%s: %s\n", rc, "Pthread_join");
        exit(0);
    }
}
```
The Pthreads "hello, world" program

```c
/*
 * hello.c - Pthreads "hello, world" program */
#include <ics.h>
void *thread(void *vargp);
int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}
/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```

What happens if w/o pthread_join?

```c
#include <ics.h>
void *thread(void *vargp);
int main() {
    pthread_t tid;
    printf("Hello from main thread! tid:%ld pid:%d\n",
          pthread_self(), getpid());
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}
void *thread(void *vargp) {
    printf("Hello from child thread! tid:%ld pid:%d ppid:%d\n",
          pthread_self(), getpid(), getppid());
    return NULL;
}
```

hellopid.c

The following routine will show us the process hierarchy of a Linux thread pool:
Linux process hierarchy for threads

```
bass> hellopid
Hello from main thread! tid:1024 pid:6024
Hello from child thread! tid:1025 pid:6026 ppid:6025
```

Thread manager supports thread abstraction using signals:
- `exit()`: kills all threads, regardless where it is called from
- Slow system calls such as `sleep()` or `read()` block only the calling thread.

beep.c: Performing concurrent tasks

```
/*
 * beeps until the user hits a key
 */
#include <ics.h>
void *thread(void *vargp);

/* shared by both threads */
char shared = '\0';

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL,
                  thread, NULL);
    while (shared == '\0') {
        printf("BEEP
");
        sleep(1);
    }
    pthread_join(tid, NULL);
    printf("DONE
");
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    shared = getchar();
    return NULL;
}
```
badcnt.c: Sharing data between threads

```c
/* bad sharing */
#include <ics.h>
#define NITERS 1000
void *count(void *arg);

struct {
    int counter;
} shared;

int main() {
    pthread_t tid1, tid2;
    pthread_create(&tid1, NULL, count, NULL);
    pthread_create(&tid2, NULL, count, NULL);
    if (shared.counter != NITERS*2)
        printf("BOOM! counter=%d\n", shared.counter);
    else
        printf("OK counter=%d\n", shared.counter);
    return NULL;
}
```

/* thread routine */
void *count(void *arg) {
    int i, val;
    for (i=0; i<NITERS; i++) {
        val = shared.counter;
        printf("%d: %d\n", (int)pthread_self(), val);
        shared.counter = val + 1;
    }
    return NULL;
}

Key point:
"struct shared" is visible to all threads.
"i" and "val" are visible only to the count thread.

Running badcnt.c

<table>
<thead>
<tr>
<th>Output of run 1</th>
<th>Output of run 2</th>
<th>Output of run 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025: 0</td>
<td>1025: 0</td>
<td>1025: 0</td>
</tr>
<tr>
<td>1025: 1</td>
<td>1025: 1</td>
<td>1025: 1</td>
</tr>
<tr>
<td>1025: 2</td>
<td>1025: 2</td>
<td>1025: 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1025: 997</td>
<td>1025: 997</td>
<td>1025: 997</td>
</tr>
<tr>
<td>1025: 998</td>
<td>1025: 998</td>
<td>1025: 998</td>
</tr>
<tr>
<td>1025: 999</td>
<td>1025: 999</td>
<td>1025: 999</td>
</tr>
<tr>
<td>2050: 969</td>
<td>2050: 969</td>
<td>2050: 969</td>
</tr>
<tr>
<td>2050: 970</td>
<td>2050: 970</td>
<td>2050: 970</td>
</tr>
<tr>
<td>2050: 971</td>
<td>2050: 971</td>
<td>2050: 971</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

So what’s the deal?
We must synchronize concurrent accesses to shared thread data (the topic of our next lecture)
Thread Synchronization

Signaling With Semaphores

Common synchronization pattern:
- Producer waits for slot, inserts item in buffer, and signals consumer.
- Consumer waits for item, removes it from buffer, and signals producer.

Examples
- Multimedia processing:
  - Producer creates MPEG video frames, consumer renders the frames
- Graphical user interfaces
  - Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer.
  - Consumer retrieves events from buffer and paints the display.
Producer-consumer (1-buffer)

```c
#include <ics.h>

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
    int buf; /* shared var */
    sem_t full; /* sems */
    sem_t empty;
} shared;

int main() {
    pthread_t tid_producer;
    pthread_t tid_consumer;
    /* initialize the semaphores */
    Sem_init(&shared.empty, 0, 1);
    Sem_init(&shared.full, 0, 0);
    /* create threads and wait */
    Pthread_create(&tid_producer, NULL,
                   producer, NULL);
    Pthread_create(&tid_consumer, NULL,
                   consumer, NULL);
    Pthread_join(tid_producer, NULL);
    Pthread_join(tid_consumer, NULL);
    exit(0);
}

-producer thread */
void *producer(void *arg) {
    int i, item;
    for (i=0; i<NITERS; i++) {
        /* produce item */
        item = i;
        printf("produced %d\n", item);
        /* write item to buf */
        P(&shared.empty);
        shared.buf = item;
        V(&shared.full);
    }
    return NULL;
}

-consumer thread */
void *consumer(void *arg) {
    int i, item;
    for (i=0; i<NITERS; i++) {
        /* read item from buf */
        P(&shared.full);
        item = shared.buf;
        V(&shared.empty);
        /* consume item */
        printf("consumed %d\n", item);
    }
    return NULL;
}
```

Initially: empty = 1, full = 0.
Basic operations on mutex variables

```
int pthread_mutex_init(pthread_mutex_t *mutex, 
                         pthread_mutexattr_t *attr)
```

Initializes a mutex variable (mutex) with some attributes (attr).
- attributes are usually NULL.
- like initializing a mutex semaphore to 1.

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

Indivisibly waits for mutex to be unlocked and then locks it.
- like P(mutex)

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

Unlocks mutex.
- like V(mutex)

---

Basic operations on condition variables

```
int pthread_cond_init(pthread_cond_t *cond, 
                        pthread_condattr_t *attr)
```

Initializes a condition variable (cond) with some attributes (attr).
- attributes are usually NULL.

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

Awakens one thread waiting on condition cond.
- if no threads waiting on condition, then it does nothing.
- key point: signals are not queued!

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

Indivisibly unlocks mutex and waits for signal on condition cond
- When awakened, indivisibly locks mutex.
Advanced operations on condition variables

**int pthread_cond_broadcast(pthread_cond_t *cond)**

Awakens _all_ threads waiting on condition `cond`.
- If no threads waiting on condition, then it does nothing.

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

Waits for condition `cond` until absolute wall clock time is `abstime`
- Unlocks `mutex` on entry, locks `mutex` on awakening.
- Use of absolute time rather than relative time is strange.

---

Signaling and waiting on conditions

**Basic pattern for signaling**

```c
Pthread_mutex_lock(&mutex);
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&mutex);
```

**Basic pattern for waiting**

```c
Pthread_mutex_lock(&mutex);
Pthread_cond_wait(&cond, &mutex);
Pthread_mutex_unlock(&mutex);
```

A mutex is always associated with a condition variable.
Guarantees that the condition cannot be signaled (and thus ignored) in the interval when the waiter locks the mutex and waits on the condition.
Barrier synchronization

Call to `barrier` will not return until every other thread has also called `barrier`.

Needed for tightly-coupled parallel applications that proceed in phases. E.g., physical simulations.

```c
#include <ics.h>
static pthread_mutex_t mutex;
static pthread_cond_t cond;
static int nthreads;
static int barriercnt = 0;
void barrier_init(int n) {
    nthreads = n;
    Pthread_mutex_init(&mutex, NULL);
    Pthread_cond_init(&cond, NULL);
}
void barrier() {
    Pthread_mutex_lock(&mutex);
    if (++barriercnt == nthreads) {
        barriercnt = 0;
        Pthread_cond_broadcast(&cond);
    } else
        Pthread_cond_wait(&cond, &mutex);
    Pthread_mutex_unlock(&mutex);
}
```

timebomb.c: timeout waiting example

A program that explodes unless the user hits a key within 5 seconds.

```c
#include <ics.h>
#define TIMEOUT 5
/* function prototypes */
void *thread(void *vargp);
struct timespec *maketimeout(int secs);
/* condition variable and its associated mutex */
pthread_cond_t cond;
pthread_mutex_t mutex;
/* thread id */
pthread_t tid;
```
timebomb.c (cont)

A routine for building a timeout structure for pthread_cond_timewait.

```c
/* maketimeout - builds a timeout object that times out in secs seconds */
struct timespec *maketimeout(int secs) {
    struct timeval now;
    struct timespec *tp = (struct timespec *)malloc(sizeof(struct timespec));
    gettimeofday(&now, NULL);
    tp->tv_sec = now.tv_sec + secs;
    tp->tv_nsec = now.tv_usec * 1000;
    return tp;
}
```

Main routine for timebomb.c

```c
int main() {
    int i, rc;
    /* initialize the mutex and condition variable */
    Pthread_cond_init(&cond, NULL);
    Pthread_mutex_init(&mutex, NULL);
    /* start getchar thread and wait for it to timeout */
    Pthread_mutex_lock(&mutex);
    Pthread_create(&tid, NULL, thread, NULL);
    for (i=0; i<TIMEOUT; i++) {
        printf("BEEP\n");
        rc = pthread_cond_timedwait(&cond, &mutex, maketimeout(1));
        if (rc != ETIMEDOUT) {
            printf("WHEW!\n");
            exit(0);
        }
    }
    printf("BOOM!\n");
    exit(0);
}
```
Thread routine for timebomb.c

```c
/*
 * thread - executes getchar in a separate thread
 */
void *thread(void *vargp) {
    (void) getchar();
    pthread_mutex_lock(&mutex);
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Threads summary

Threads provide another mechanism for writing concurrent programs.

Threads are growing in popularity
- Somewhat cheaper than processes.
- Easy to share data between threads.

However, the ease of sharing has a cost:
- Easy to introduce subtle synchronization errors.

For more info:
- man pages (man -k pthreads)