CSCE 313 Introduction to Computer Systems

Instructor: Dr. Guofei Gu

http://courses.cse.tamu.edu/guofei/csce313/

POSIX Threads

- Why Threads?
  - Latency Hiding / Multiprogramming (covered earlier)
  - Ease of Programming (covered now)
- POSIX Threads (R&R, Chapter 12)
  - Thread Management
  - Thread Safety
  - Thread Attributes
POSIX Threads

- Why Threads?
  - Latency Hiding / Multiprogramming (covered earlier)
  - Ease of Programming (covered now)
- POSIX Threads (R&R, Chapter 12)
  - Thread Management
  - Thread Safety
  - Thread Attributes

Why Threads?

- Many interactive applications run in loops.
- For example, an interactive game.

while (1) {
    /* Read Keyboard */
    /* Recompute Player Position */
    /* Update Display */
}

- Reference [B.O. Gallmeister,
  “POSIX.4, Programming for the
  Real World,” O’Reilly&Assoc.,
  Inc.]
Why Threads?

- Many interactive applications run in loops.
- For example, an interactive game.

```c
while (1) {
    /* Synchronize to Highest Frequency */
    /* Read Keyboard */
    /* AND Read Mouse */
    /* Recompute Player Position */
    /* Update Display */
    /* AND emit sounds */
}
```


Suddenly, application is getting complex!

Why Threads?

- Many interactive applications run in loops.
- For example, an interactive game.

- It ain’t over yet!
- What about compute-intensive operations, like AI, video compression?
- How about Signal Handlers?

```c
while (1) {
    /* Synchronize to Highest Frequency */
    /* Read Keyboard */
    /* AND Read Mouse */
    /* Recompute Player Position */
    /* Update Display */
    /* AND all other lights */
    /* AND emit sounds */
    /* AND more sounds */
    /* AND move game physically */
}
```

Reading the Mouse

while (1) {
    /* Synchronize to Highest Frequency */
    /* Read Keyboard */
    /* AND Read Mouse */
    /* Recompute Player Position */
    /* Update Display */
    /* AND all other lights */
    /* AND emit sounds */
    /* AND more sounds */
    /* AND move game physically */
}

read_mouse() {
}

Reading the Mouse (II)

while (1) {
    /* Synchronize to Highest Frequency */
    /* Read Keyboard */
    /* AND Read Mouse */
    /* Recompute Player Position */
    /* Update Display */
    /* AND all other lights */
    /* AND emit sounds */
    /* AND more sounds */
    /* AND move game physically */
}

int main() {
    read_mouse() {
    }
}

Separate Process

fork()
Reading the Mouse (III)

while (1) {
    /* Synchronize to Highest Frequency */
    /* Read Keyboard */
    /* AND Read Mouse */
    /* Recompute Player Position */
    /* Update Display */
    /* AND all other lights */
    /* AND emit sounds */
    /* AND more sounds */
    /* AND move game physically */
}

read_mouse() {
}

Separate Thread

The Thread and its Creation

/* The Mouse Input Function */
void * read_mouse() {
    char buf[BUFSIZE]; ssize_t nbytes;
    for (;;) {
        if ((nbytes = read_from_mouse(buf, BUFSIZE)) <= 0) break;
        dosomething_with(buf, nbytes);
    }
    return NULL;
}
The Thread and its Creation

```c
#include <pthread.h>

int error;
pthread_t tid;

if (error = pthread_create(&tid, NULL, read_mouse, NULL)) {
   perror("Failed to create read_mouse thread");
}

while (1) {
   /* Synchronize to Highest Frequency */
   /* Read Keyboard */
   /* AND Read Mouse */ // Handled by separate thread!
   /* Recompute Player Position */
   /* Update Display */
   /* AND all other lights */
   /* AND emit sounds */
   /* AND more sounds */
   /* AND move game physically */
}
```

Thread Management

- `pthread_create` (create a thread)
- `pthread_self` (what is my id?)
- `pthread_equal` (two thread ids equal?)
- `pthread_detach` (have thread release res’ s)
- `pthread_join` (wait for a thread)
- `pthread_cancel` (terminate another thread)
- `pthread_exit` (exit a thread)
- `pthread_kill` (send a signal to a thread)

Most POSIX thread functions return 0 if successful and nonzero (error code) if unsuccessful
Thread Management

- `pthread_cancel` (terminate another thread)
- `pthread_create` (create a thread)
- `pthread_detach` (have thread release res’s)
- `pthread_equal` (two thread id’s equal?)
- `pthread_exit` (exit a thread)
- `pthread_kill` (send a signal to a thread)
- `pthread_join` (wait for a thread)
- `pthread_self` (what is my id?)

```c
int pthread_create(pthread_t *restrict thread, 
                  const pthread_attr_t * restrict attr, 
                  void *(*start_routine)(void *),
                  void *restrict arg)
```

```c
fd = open("my.dat", O_RDONLY);
if (error = pthread_create(&t_id, NULL, processfd, &fd))
    fprintf(stderr, "Failed create thread: \n", strerror(error));
```

Thread Safety

- A function is thread-safe if multiple threads can execute simultaneous active invocations of the function without interference
  - Interference typically occurs when threads access the same (shared) data

- Some functions (not thread-safe) can have a thread-safe version designated with an `_r` suffix (e.g., `strerror` -> `strerror_r`)
Thread Attributes

<table>
<thead>
<tr>
<th>Attribute Objects</th>
<th>pthread_attr_destroy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pthread_attr_init</td>
</tr>
<tr>
<td>State</td>
<td>pthread_attr_getdetachstate</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setdetachstate</td>
</tr>
<tr>
<td>Stack</td>
<td>pthread_attr_getguardsize</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setguardsize</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getstack</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setstack</td>
</tr>
<tr>
<td>Scheduling</td>
<td>pthread_attr_getinheritedsched</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setinheritedsched</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getschedparam</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setschedparam</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getschedpolicy</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setschedpolicy</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getscope</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setscope</td>
</tr>
</tbody>
</table>

Thread Attributes: State

<table>
<thead>
<tr>
<th>Attribute Objects</th>
<th>pthread_attr_destroy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pthread_attr_init</td>
</tr>
<tr>
<td>State</td>
<td>pthread_attr_getdetachstate</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setdetachstate</td>
</tr>
<tr>
<td>Stack</td>
<td>pthread_attr_getguardsize</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setguardsize</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getstack</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setstack</td>
</tr>
<tr>
<td>Scheduling</td>
<td>pthread_attr_getinheritedsched</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setinheritedsched</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getschedparam</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setschedparam</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getschedpolicy</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setschedpolicy</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getscope</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_setscope</td>
</tr>
</tbody>
</table>

Detached threads release resources when terminate.

Attached states hold on to resources until parent thread calls pthread_join.
### Thread Attributes: Stack

| attribute objects | pthread_attr_destroy  
|--------------------|-----------------------
|                    | pthread_attr_init      |
| state              | pthread_attr_getdetachstate 
|                    | pthread_attr_setdetachstate |
| stack              | pthread_attr_getguardsize 
|                    | pthread_attr_setguardsize  
|                    | pthread_attr_getstack    
|                    | pthread_attr_setstack    |
| scheduling         | pthread_attr_getinheritedsched 
|                    | pthread_attr_setinheritedsched  
|                    | pthread_attr_getschedparam 
|                    | pthread_attr_setschedparam 
|                    | pthread_attr_getschedpolicy 
|                    | pthread_attr_setschedpolicy 

- **setstack** defines location and size of stack.
- **setguardsize** allocates additional memory. If the thread overflows into this extra memory, an error is generated.

### Thread Attributes: Scheduling

| attribute objects | pthread_attr_destroy |
|--------------------|----------------------

- **PTHREAD_INHERIT_SCHED** defines that scheduling parameters are inherited from parent thread. (as opposed to **PTHREAD_EXPLICIT_SCHED**).

<table>
<thead>
<tr>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

- **contention scope** defines whether process competes at process level or at system level for resources.
```c
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#define NUMTHREADS 10

static void *printarg(void *arg) {
    fprintf(stderr, "Thread received id:\n", *((int *)arg);
    return NULL;
}

int main (void) {    /* program incorrectly passes parameters to threads */
    int error;
    int i;
    int j;
    pthread_t tid[NUMTHREADS];

    for (i = 0; i < NUMTHREADS; i++)
        if (error = pthread_create(tid + i, NULL, printarg, (void *)&i)) {        
            fprintf(stderr, "Failed to create thread: %s\n", strerror(error));
            tid[i] = pthread_self();
        }

    for (j = 0; j < NUMTHREADS; j++) {
        if (pthread_equal(pthread_self(), tid[j]))
            continue;
        if (error = pthread_join(tid[j], NULL))
            fprintf(stderr, "Failed to join thread: %s\n", strerror(error));
    }

    printf("All threads done\n");
    return 0;
}
```