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?

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

```c
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 */
}
```

- 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.
- If 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 */
}
```

Suddenly, application is getting complex!


Reading the Mouse

```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 */
}
```

```c
read_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 */
}

int main() {
    read_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() {
}
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 */
    /* Recompute Player Position */
    /* Update Display */
    /* AND all other lights */
    /* AND emit sounds */
    /* AND more sounds */
    /* AND move game physically */
}
```

/* The Mouse Input Function */

```c
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;
}
```
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)
```
### Thread Attributes

<table>
<thead>
<tr>
<th>Attribute object</th>
<th>pthread_attr_destroy</th>
<th>pthread_attr_init</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>pthread_attr_getdetachstate</td>
<td>pthread_attr_setdetachstate</td>
</tr>
<tr>
<td>stack</td>
<td>pthread_attr_getguardsize</td>
<td>pthread_attr_setguardsize</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getstack</td>
<td>pthread_attr_setstack</td>
</tr>
<tr>
<td>scheduling</td>
<td>pthread_attr_getinheritedsched</td>
<td>pthread_attr_setinheritedsched</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getschedparam</td>
<td>pthread_attr_setschedparam</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getschedpolicy</td>
<td>pthread_attr_setschedpolicy</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getscope</td>
<td>pthread_attr_setscope</td>
</tr>
</tbody>
</table>

### Thread Attributes: State

<table>
<thead>
<tr>
<th>Attribute object</th>
<th>pthread_attr_destroy</th>
<th>pthread_attr_init</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>pthread_attr_getdetachstate</td>
<td>pthread_attr_setdetachstate</td>
</tr>
<tr>
<td>stack</td>
<td>pthread_attr_getguardsize</td>
<td>pthread_attr_setguardsize</td>
</tr>
<tr>
<td></td>
<td>pthread_attr_getstack</td>
<td>pthread_attr_setstack</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_getinheritsched |
|                   | pthread_attr_getsetparam  
|                   | pthread_attr_getscope     |
|                   | pthread_attr_getsetpolicy |

*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

<table>
<thead>
<tr>
<th>Attribute objects</th>
<th>pthread_attr_destroy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>state</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>scheduling</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

*Scheduling policies:* SCHED_FIFO, SCHED_RR, SCHED_SPORADIC, SCHED_OTHER, ...

*contention scope* defines whether process competes at process level or at system level for resources.