Pthreads
Threads vs. Processes

- **Process**
  - One address space per process
  - Each process has its own data (global variables), stack, heap

- **Thread**
  - Multiple threads share on address space
    - But its own stack and register context
  - Threads within the same address space share data (global variables), heap
Shared Memory Programming

- Multiple threads (processes) on shared address space
  - More convenient programming model
  - Careful control required when shared data are accessed

- Programming models
  - Threads libraries (classes): Pthreads, Java threads
  - New programming languages: Ada
  - Modifying syntax of existing languages: UPC, Cilk
  - Compiler directives: OpenMP
Pthreads

- Standard thread APIs for UNIX
  - Created by IEEE
  - Called POSIX 1003.1c in 1995

- Pthreads = POSIX threads
Pthreads – creation & join

- `pthread_create`, `pthread_join`

```
main()

pthread_create(&th_id, NULL, proc1, &arg)

pthread_join(th_id, *status)
```

Proc 1:
```
proc1(*arg)
{
    return (&status);
}
```
Pthreads – detached thread

- Pthread_attr_setdetachstate

```c
main()

pthread_attr_setdetachstate(&attr, PTHREAD_DETACHED)

pthread_create(&th_id1, &attr, proc1, NULL)

proc1 {
}

proc2 {
}

terminates!

terminates!
```
#include <pthread.h>
#include <stdio.h>

void * thread_fn(void *arg) {
    printf("Hello World!\n");
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thr;
    if (pthread_create(&thr, NULL, thread_fn, NULL)) {
        printf("thread creation failed\n");
        return -1;
    }
    if (pthread_join(thr, NULL)) {
        printf("thread join failed\n");
        return -1;
    }
    return 0;
}
Pthreads – compilation

- Pthreads are supported by almost all compilers
  - GNU Compiler
    - `gcc -Wall -o hello hello.c -lpthread`
    - `-lxxx` : specifies which static library to link
    - `-Wall` : specifies to print out all types of warnings
Pthreads – Exiting a Thread

- **4 ways to exit threads**
  - Thread will naturally exit after starting thread function returns
  - Thread itself can exit by calling `pthread_exit`
  - Other threads can terminate a thread by calling `pthread_cancel`
  - Thread exits if the process that owns the thread exits

- **APIs**
  - void `pthread_exit (void *retval);`
  - int `pthread_cancel (pthread_t thread)`
Synchronization

- Accessing shared data
  - Example: two threads increase the same variable $x$

```c
int x = 0;
inc ()
{
    x = x + 1;
}
main()
{
    pthread_create(&th1, NULL, inc, NULL);
    pthread_create(&th2, NULL, inc, NULL);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
    printf("x = %d\n", x);
}
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>load r1 &lt;= x</td>
<td>load r1 &lt;= x</td>
</tr>
<tr>
<td></td>
<td>add r1 &lt;= r1, 1</td>
<td>add r1 &lt;= r1, 1</td>
</tr>
<tr>
<td></td>
<td>store r1 =&gt; x</td>
<td>store r1 =&gt; x</td>
</tr>
</tbody>
</table>

Thread 1

Thread 2
Critical Sections

- **Critical section**
  - Need to guarantee that one process (thread) can access a certain resource at a time
  - Implemented mechanism is known as “mutual exclusion”

- **Locks**
  - Simple mechanism for mutual exclusion
  - A lock can have only two values
    - 1 – a thread entered the critical section
    - 0 – no thread is in the critical section
  - Acquire the lock before entering the critical section (set to 1)
  - Release the lock after leaving the critical section (set to 0)
Acquiring Locks

- Simple C code is adequate?

```c
lock(lock_var) {
    if (lock_var == 0)  // lock is free
        lock_var = 1;
}
```

```c
unlock(lock_var) {
    lock_var = 0;
}
```

- Special atomic instruction should be used
  - Pthread library provides APIs
Pthreads - Lock

- Use a special mutex variable & APIs

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);
  // critical section
pthread_mutex_unlock(&lock);
```

- `pthread_mutex_lock`
  - A thread will wait until it can acquire the lock

- `pthread_mutex_unlock`
  - If multiple threads are waiting, only one thread is selected to receive the lock
  - Only the thread that acquires the lock can unlock it
Serialization

- Critical sections serialize the code execution
  - Too many or large critical sections can slow down the performance – sequential code may run faster
Condition Variables

- Wait until a condition is satisfied
  - A global variable is used to indicate condition (predicate value)

- A mutex lock and a condition variable are used together
  - `pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;`
  - `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`

```
thread1

action() {
    ...
    pthread_mutex_lock(&lock);
    while (predicate == 0) // test predicate
        pthread_cond_wait(&cond, &lock);
    pthread_mutex_unlock(&lock);
    // perform action
    ...
}

thread2

signal() {
    ...
    pthread_mutex_lock(&lock);
    predicate = 1; // set predicate
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&lock);
    ...
}
```
Condition Variables (2)

- Three variables are linked all together
  - mutex lock, condition variable, predicate

- `pthread_cond_wait(pthread_cond_t*, pthread_mutex_t*)`
  - If condition is not signaled yet, waits inside the function after releasing the mutex lock
  - If condition is signaled, returns after acquiring the mutex lock

- Other related functions
  - `pthread_cond_timedwait(pthread_cond_t*, pthread_mutex_t*, struct timespec*)`
    - After waiting for a given time, check signal
  - `pthread_cond_broadcast(pthread_cond_t*)`
    - If multiple threads wait on condition, wake them all