

## Overview

- We take a look at how threads are implemented in Linux
- We examine the POSIX threads API and look at some examples of using the API to create and manipulate threads in C
- *Linux Kernel Development*, by Love, chapter 2
- *Programming with POSIX Threads*, by Butenhof

## Threads in Linux

- Linux takes a unique approach to implementing threads: there is in fact no concept of a thread inside the kernel, there are only processes that can share certain resources
- Each thread appears as an ordinary process to the kernel and is scheduled as such
- There are no thread-specific data structures i.e. each thread is represented by a `task_struct` just like an ordinary process
- Threads (like processes) are created by invoking the `clone` system call (`fork` calls `clone`)

## Threads in Linux

- `clone` forks a child process but allows control over which resources will be shared between the parent and child
- Other operating systems (Windows and Solaris) have explicit thread support built into the kernel
- Such support exploits the fact that a thread resides within a process to optimise the time required to context switch two threads from the same process
- The disadvantage of this approach is that the kernel becomes more complicated: extra data structures specific to threads are required and context switching code needs to take them into account

## Threads in Linux

- In Linux all threads are represented as processes and some efficiency is sacrificed for the sake of simplicity
- In fact, experiment shows that the time taken to create a process in Linux compares favourably with the time taken to create a thread in some other operating systems
- Furthermore, the Linux scheduler makes no effort to successively schedule threads from the same process

## POSIX Threads

- With many different thread support packages a standard became required to enable developers to write portable multithreaded applications
- By adhering to the standard developers were guaranteed their threads would behave in a standard way irrespective of how they were actually implemented
- IEEE defined the POSIX threads standard and thread packages that adhere to the defined API are `pthread`

## POSIX Threads

- Most Unix operating systems support `pthread` and Linux is no exception
- The standard defines over 60 function calls but we will here limit ourselves to looking at only a subset
- To finish our look at POSIX threads we examine how a simple producer-consumer problem can be solved using `pthread`

## POSIX Threads

### Some pthread functions

- pthread\_create, pthread\_exit
- pthread\_join
- pthread\_mutex\_init, pthread\_mutex\_lock, pthread\_mutex\_unlock, pthread\_mutex\_destroy
- pthread\_cond\_init, pthread\_cond\_wait, pthread\_cond\_timedwait, pthread\_cond\_signal, pthread\_cond\_broadcast, pthread\_cond\_destroy
- pthread\_cancel

## POSIX Threads

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

```
void *  
sayhello(void *p) {  
    char *q = (char *)p;  
    printf("Hello %s\n", q);  
}
```

```
pthread_t t;  
char *n = "Ruby";  
pthread_create(&t, NULL, sayhello, (void *)n);
```

## POSIX Threads

```
int pthread_join(pthread_t thread,  
                 void **value_ptr);
```

...

```
pthread_create(&t, NULL, sayhello, (void *)n);  
pthread_join(t, NULL);
```

...

## POSIX Threads

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutexattr_t *attr);

void *
foo(void *p) {
    pthread_mutex_lock(&mutex);
    /* Update p */
    pthread_mutex_unlock(&mutex);
}

pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_create(&t1, NULL, foo, (void *)n);
pthread_create(&t2, NULL, foo, (void *)n);
```

## POSIX Threads

- Clearly, creating and starting threads is straightforward
- However, the ability to terminate threads may also be useful:
  1. A user initiates a search for a document
  2. The application starts a thread to carry out the search
  3. The user gets sick of waiting and cancels the search
  4. The application will need to shut down the search thread
- A cancellation request is sent to a thread with `pthread_cancel` and acknowledged with `pthread_testcancel` at a cancellation point
- The following is a very simple example. . .

## POSIX Threads

```
int pthread_cancel(pthread_t thread);
void pthread_testcancel(void);

void *
counter(void *p) {
    for (i = 0; ; i++) {
        /* Do something */
        pthread_testcancel();
    }
}

pthread_create(&t, NULL, counter, (void *)n);
/* Sick of waiting... */
pthread_cancel(t);
pthread_join(t);
```

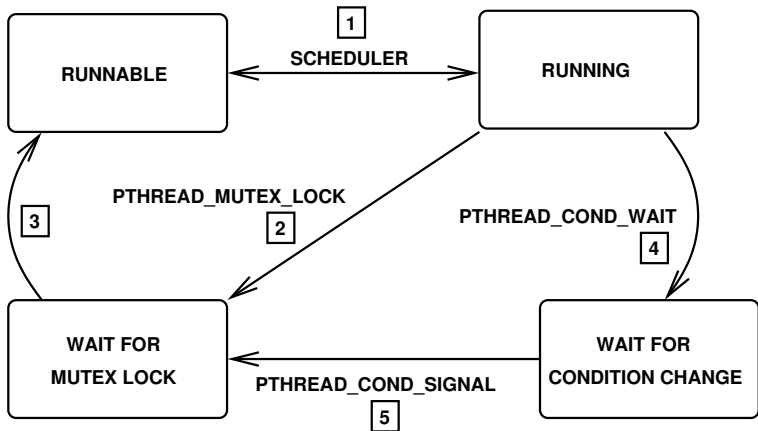
## POSIX Threads

- Condition variables allow us to define if and when processes are to wait and on what conditions
- Common in producer-consumer problems
- Used to start and stop threads according to the state of some shared data structure

## POSIX Threads

- The condition variable mechanism releases a held mutex when a thread waits on a condition (essential to allow other threads to alter the condition waited upon)
- Compare this with Java's `wait` and `notify` methods: the release of a lock was implicit in Java's `wait`, with a condition variable the release of a lock is made explicit
- To wait on some condition we use `pthread_cond_wait` and to notify waiters that some condition has changed we use `pthread_cond_signal` or `pthread_cond_broadcast`

# POSIX Threads



## POSIX Threads

```
typedef struct {
    int *buffer;
    int size;
    int occupied;
    int nextin;
    int nextout;
    int ins;
    int outs;
    pthread_mutex_t mutex;
    pthread_cond_t more;
    pthread_cond_t less;
} bb_t;
```

## POSIX Threads

```
pthread_t p, c;
bb_t bb;

/* Initialise lock and condition variables */
pthread_mutex_init(&bb.mutex, NULL);
pthread_cond_init(&bb.more, NULL);
pthread_cond_init(&bb.less, NULL);

/* Create our threads */
pthread_create(&p, NULL, producer, (void *)&bb);
pthread_create(&c, NULL, consumer, (void *)&bb);

/* Wait for our threads */
pthread_join(p, NULL);
pthread_join(c, NULL);
```

## POSIX Threads

```
void *
producer(void *bb_in)
{
    bb_t  *bb = (bb_t *)bb_in;

    while (1) {
        pthread_mutex_lock(&bb->mutex);
        while (bb->occupied == bb->size) {
            pthread_cond_wait(&bb->less, &bb->mutex);
        }

        /* Insert item */

        pthread_cond_signal(&bb->more);
        pthread_mutex_unlock(&bb->mutex);
    }
}
```