Thread and Synchronization

pthread Programming
(Module 19)

Yann-Hang Lee
Arizona State University
yhlee@asu.edu
(480) 727-7507

Summer 2014
Pthread APIs

- `pthread_create( )`
- `pthread_detach( )`
- `pthread_equal( )`
- `pthread_exit( )`
- `pthread_join( )`
- `pthread_self( )`
- `pthread_cancel()`
- `pthread_mutex_init()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `pthread_mutex_trylock()`
- `pthread_mutex_unlock()`
- `sched_yield( )`

```c
int pthread_create(  
    pthread_t *tid,             // Thread ID returned by the system  
    const pthread_attr_t *attr,  // optional creation attributes  
    void *(*start)(void *),     // start function of the new thread  
    void *arg                   // Arguments to start function
);
```
Example of Thread Creation

```c
#include <pthread.h>
#include <stdio.h>

void *thread_routine(void* arg){
    printf("Inside newly created thread \n");
}

void main(){
    pthread_t    thread_id;  // threat handle
    void   *thread_result;

    pthread_create( &thread_id, NULL, thread_routine, NULL );

    printf("Inside main thread \n");
    pthread_join( thread_id, &thread_result );
}
```
Shared Code and Reentrancy

- A single copy of code is invoked by different concurrent tasks must be reentrant
  - pure code
  - variables in task stack guarded
    - global and static variables (with semaphore or taskLock)
  - variables in task content

- Can a driver manage multiple devices?
- Can multiple threads running in kernel space at the same time?

```c
void taskOne ()
{
    ....
    myFunc ();
    ....
}

void taskTwo ()
{
    ....
    myFunc ();
    ....
}

void myFunc ()
{
    ....
    ....
}
```
Synchronization in Linux Kernel

- The old Linux system ran all system services to completion or till they blocked (waiting for IO).
  - When it was expanded to SMP, a lock was put on the kernel code to prevent more than one CPU at a time in the kernel.

- Kernel preemption
  - A process running in kernel mode can be replaced by another process while in the middle of a kernel function
  - In the example, process B may be waked up by a timer and with higher priority
  - Why – dispatch latency

(Christopher Hallinan, "Embedded Linux Primer: A Practical Real-World Approach").
When Synchronization in Necessary

- A race condition can occur when the outcome of a computation depends on how two or more interleaved kernel control paths are nested.

- To identify and protect the critical regions in exception handlers, interrupt handlers, deferrable functions, and kernel threads:
  - On single CPU, critical region can be implemented by disabling interrupts while accessing shared data.
  - If data is shared among threads, critical region can be done by disabling preemption.
  - If the same data is shared only by the service routines of system calls, critical region can be implemented by disabling kernel preemption (interrupt is allowed) while accessing shared data.

- How about multiprocessor systems (SMP):
  - Different synchronization techniques are necessary for data to be accessed by multiple CPUs.
Thread Synchronization -- Mutex

- **Mutual exclusion (mutex):**
  - guard against multiple threads modifying the same shared data simultaneously
  - provides locking/unlocking critical code sections where shared data is modified

- **Basic Mutex Functions:**
  
  ```c
  int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
  int pthread_mutex_lock(pthread_mutex_t *mutex);
  int pthread_mutex_unlock(pthread_mutex_t *mutex);
  int pthread_mutex_destroy(pthread_mutex_t *mutex);
  ```

  - data type named `pthread_mutex_t` is designated for mutexes
  - the attribute of a mutex can be controlled by using the `pthread_mutex_init()` function
#include <pthread.h>

pthread_mutex_t my_mutex; // should be of global scope

... 
int main()
{
    int tmp;
    ...
    tmp = pthread_mutex_init( &my_mutex, NULL ); // initialize the mutex
    ...
    // create threads
    ...
    pthread_mutex_lock( &my_mutex );
    do_something_private();
    pthread_mutex_unlock( &my_mutex );
    ...
    return 0;
}