Thread and Synchronization

Yann-Hang Lee

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ 85287

yhlee@asu.edu
(480) 727-7507
Why Talk About This Subject

- **A thread of program execution**
  - How a program start and end its execution
  - waiting for an event or a resource, delay a period, etc.
- **For concurrent operations → multiple threads of program execution**
- **How can we make this happen?**
  - support for program execution
  - sharing of resources
  - scheduling
  - communication between threads
Thread and Process

- **process:**
  - an entity to which system resources (CPU time, memory, etc.) are allocated
  - an address space with 1 or more threads executing within that address space, and the required system resources for those threads

- **thread:**
  - a sequence of control within a process and shares the resources in that process

- **lightweight process (LWP):**
  - LWP may share resources: address space, open files, …
  - clone or fork
  - An implementation of thread: to associate a lightweight process with each thread
Why Threads

- **Advantages:**
  - the overhead for creating a thread is significantly less than that for creating a process
  - multitasking, i.e., one process serves multiple clients
  - switching between threads requires the OS to do much less work than switching between processes

- **Drawbacks:**
  - not as widely available as the process features
  - writing multithreaded programs require more careful thought
  - more difficult to debug than single threaded programs
  - for single processor machines, creating several threads in a program may not necessarily produce an increase in performance (only so many CPU cycles to be had)
IEEE's POSIX Threads Model:
- programming models for threads in a UNIX platform
- pthreads are included in the international standards

pthreads programming model:
- creation of threads
- managing thread execution
- managing the shared resources of the process

main thread:
- initial thread created when main() is invoked
- has the ability to create daughter threads
- if the main thread returns, the process terminates even if there are running threads in that process
- to explicitly avoid terminating the entire process, use pthread_exit()
struct task_struct {
    volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage;
    unsigned int flags;       /* per process flags, defined below */
    unsigned int ptrace;
    int lock_depth;           /* BKL (big kernel lock) lock depth */
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    const struct sched_class *sched_class;
    ...
    struct mm_struct *mm, *active_mm;
    struct thread_struct thread;    /* CPU-specific state of this task */
    struct fs_struct *fs;   /* filesystem information */
    struct files_struct *files;  /* open file information */
Process -- task_struct data structure

- **state**: process state
  - TASK_RUNNING: executing
  - TASK_INTERRUPTABLE: suspended (sleeping)
  - TASK_UNINTERRUPTABLE: (no process of signals)
  - TASK_STOPPED (stopped by SIGSTOP)
  - TASK_TRACED (being monitored by other processes such as debuggers)
  - EXIT_ZOMBIE (terminated before waiting for parent)
  - EXIT_DEAD

- **thread_info**: low-level information for the process
- **mm**: pointers to memory area descriptors
- **tty**: tty associated with the process
- **fs**: current directory
- **files**: pointers to file descriptors
- **signal**: signals received .............
Linux Processor State

/* This is the TSS (task State Segment) defined by the hardware and saved in stack. */
struct x86_hw_tss {
  unsigned short back_link, __blh;
  unsigned long sp0;
  unsigned short ss0, __ss0h;
  unsigned long sp1;
  /* ss1 caches MSR_IA32_SYSENTER_CS: */
  unsigned short ss1, __ss1h;
  unsigned long sp2;
  unsigned short ss2, __ss2h;
  unsigned long __cr3;
  unsigned long ip;
  unsigned long flags;
  unsigned long ax;
  unsigned long cx;
  unsigned long dx;
  unsigned long bx;
Linux Thread State Transition

- Ready
  - Start
  - Scheduled
- Running
  - Preempted
  - Wait satisfied
- Blocked
  - Wait for resource
- Terminated
  - Done or cancelled
Task Management in vxWorks

- Task structure in task control block –
  - priority (initial and inherited), stack frame, task current state,
  - entry point, processor states (program counter, registers)
  - callback function (hook) pointers for OS events
  - spare variables

Diagram:

- Execution
  - Ready
  - Blocked

- Execution:
  - pending
  - ready
  - delayed
  - suspended

- TaskInit()
typedef struct windTcb /* WIND_TCB - task control block */
{
    char * name;  /* 0x34: pointer to task name */
    UINT status; /* 0x3c: status of task */
    UINT priority; /* 0x40: task's current priority */
    UINT priNormal; /* 0x44: task's normal priority */
    UINT priMutexCnt; /* 0x48: nested priority mutex owned */
    UINT lockCnt; /* 0x50: preemption lock count */
    FUNC_PTR entry; /* 0x74: entry point of task */
    char * pStackBase; /* 0x78: points to bottom of stack */
    char * pStackLimit; /* 0x7c: points to stack limit */
    char * pStackEnd; /* 0x80: points to init stack limit */
#if (CPU_FAMILY==I80X86) /* function declarations */
    EXC_INFO excInfo; /* 0x118: exception info */
    REG_SET regs; /* 0x12c: register set */
    DBG_INFO_NEW dbgInfo0; /* 0x154: debug info */
#endif /* CPU_FAMILY==I80X86 */
}
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 reentrant
  - pure code
  - variables in task stack (parameters)
  - guarded global and static variables (with semaphore or taskLock)
  - variables in task content (taskVarAdd)

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

//taskTwo
{
    ....
    myFunc();
    ....
}
```
Thread Synchronization -- Mutex (1)

- **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
Example: Mutex

```c
#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;
}
```
Thread Synchronization -- Semaphore (2)

- **creating a semaphore:**
  
  ```c
  int sem_init(sem_t *sem, int pshared, unsigned int value);
  ```

  - initializes a semaphore object pointed to by sem
  - pshared is a sharing option; a value of 0 means the semaphore is local to the calling process
  - gives an initial value to the semaphore

- **terminating a semaphore:**
  
  ```c
  int sem_destroy(sem_t *sem);
  ```

- **semaphore control:**
  
  ```c
  int sem_post(sem_t *sem);
  int sem_wait(sem_t *sem);
  ```

  - sem_post atomically increases the value of a semaphore by 1,
  - sem_wait atomically decreases the value of a semaphore by 1; but always waits until the semaphore has a non-zero value first
Example: Semaphore

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

void *thread_function( void *arg ) {
    sem_wait( &semaphore ); perform_task(); pthread_exit( NULL );
}

sem_t semaphore; // also a global variable just like mutexes

int main() {
    int tmp;
    tmp = sem_init( &semaphore, 0, 0 ); // initialize the semaphore
    pthread_create( &thread[i], NULL, thread_function, NULL ); // create threads
    while ( still_has_something_to_do() )
    {
        sem_post( &semaphore );
        ...
    }
    pthread_join( thread[i], NULL );
    sem_destroy( &semaphore );
    return 0;
}
```
Condition Variables

- A variable of type `pthread_cond_t`
- Use condition variables to atomically block threads until a particular condition is true.
- Always use condition variables together with a mutex lock.
  ```c
  pthread_mutex_lock();
  while( condition_is_false )
      pthread_cond_wait();
  pthread_mutex_unlock();
  ```
- Use `pthread_cond_wait()` to atomically release the mutex and to cause the calling thread to block on the condition variable
- The blocked thread can be awakened by `pthread_cond_signal()`, `pthread_cond_broadcast()`, or when interrupted by delivery of a signal.
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 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.
- Note that interrupts can be nested, but they are non-blocking, not preempted by system calls.
Atomic Operations

- Atomic operations provide instructions that are
  - executable atomically;
  - without interruption
  - Not possible for two atomic operations by a single CPU to occur concurrently

- Atomic 80x86 instructions
  - Instructions that make zero or one aligned memory access
  - Read-modify-write instructions (inc or dec)
  - Read-modify-write instructions whose opcode is prefixed by the lock byte (0xf0)

- Linux kernel
  - two sets of interfaces for atomic operations: one for integers and another for individual bits
Linux Atomic Operations

- Uses atomic_t data type
- Atomic operations on integer counter in Linux

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_read(v)</td>
<td>Return *v</td>
</tr>
<tr>
<td>atomic_set(v, i)</td>
<td>set *v to i</td>
</tr>
<tr>
<td>atomic_add(i, v)</td>
<td>add i to *v</td>
</tr>
<tr>
<td>atomic_sub(i, v)</td>
<td>subtract i from *v</td>
</tr>
<tr>
<td>atomic_sub_and_test(i, v)</td>
<td>subtract i from *v and return 1 if result is 0</td>
</tr>
<tr>
<td>atomic_inc(v)</td>
<td>add 1 to *v</td>
</tr>
<tr>
<td>atomic_dec(v)</td>
<td>subtract 1 from *v</td>
</tr>
<tr>
<td>atomic_dec_and_test(v)</td>
<td>subtract 1 from *v and return 1 if result is 0</td>
</tr>
<tr>
<td>atomic_inc_and_test(v)</td>
<td>add 1 to *v and return 1 if result is 0</td>
</tr>
<tr>
<td>atomic_add_negative(i, v)</td>
<td>add i to *v and return 1 if result is negative</td>
</tr>
</tbody>
</table>

- A counter to be incremented by multiple threads
- Atomic operate at the bit level, such as

```c
unsigned long word = 0;
set_bit(0, &word);    /* bit zero is now set (atomically) */
```
Spinlock

- **Ensuring mutual exclusion using a busy-wait lock.**
  - if the lock is available, it is taken, the mutually-exclusive action is performed, and then the lock is released.
  - If the lock is not available, the thread busy-waits on the lock until it is available.
  - It keeps spinning, thus wasting the processor time.
  - If the waiting duration is short, faster than putting the thread to sleep and then waking it up later when the lock is available.
  - really only useful in SMP systems

- **Spinlock with local CPU interrupt disable**
  ```c
  spin_lock_irqsave( &my_spinlock, flags );
  // critical section
  spin_unlock_irqrestore( &my_spinlock, flags );
  ```

- **Reader/writer spinlock** – allows multiple readers with no writer
Semaphore

Kernel semaphores

- struct semaphore: count, wait queue, and number of sleepers

```c
void sem_init(struct semaphore *sem, int val);
// Initialize a semaphore's counter sem->count to given value
inline void down(struct semaphore *sem);
// try to lock the critical section by decreasing sem->count
inline void up(struct semaphore *sem); // release the semaphore
```

- blocked thread can be in TASK_UNINTERRUPTIBLE or TASK_INTERRUPTIBLE (by timer or signal)

- Special case – mutexes (binary semaphores)
  ```
  void init_MUTEX(struct semaphore *sem)
  void init_MUTEX_LOCKED(struct semaphore *sem)
  ```

- Read/Write semaphores
Spin lock vs Semaphore

- Only a spin lock can be used in interrupt context,
- Only a semaphore can be held while a task sleeps.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Recommended Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low overhead locking</td>
<td>Spin lock</td>
</tr>
<tr>
<td>Short lock hold time</td>
<td>Spin lock</td>
</tr>
<tr>
<td>Long lock hold time</td>
<td>Semaphore</td>
</tr>
<tr>
<td>Need to lock from interrupt context</td>
<td>Spin lock</td>
</tr>
<tr>
<td>Need to sleep while holding lock</td>
<td>Semaphore</td>
</tr>
</tbody>
</table>

Other mechanisms:
- Completion: synchronization among multiprocessors
- The global kernel lock (a.k.a big kernel lock, or BKL)
  - Lock_kernel(), unlock_kernel()
- RCU – read-copy update, for mostly-read access
Reader/Writer -- ISR and Buffering

- Input: single producer (ISR) and single consumer (thread)
- If a read is initialized by the thread
  - calls "read" with a buffer of $n$ bytes
  - initiate IO operation, enable interrupt
  - ISR reads input and store in the buffer.
  - If done, single the completion
- Blocking or nonblocking
  - in thread context (e.g. vxWorks) – semaphore, lock
  - in kernel context (Linux) – wait queue
- Guarded access
  - Lock (mutex) and interrupt lock (disable)
Ring Buffer

- if \( p_{\text{read}} = p_{\text{write}} \), empty
  \[ (p_{\text{write}} + 1) \% \text{size} = p_{\text{read}} \], full
- Invariant: \( p_{\text{write}} \) never incremented up to \( p_{\text{read}} \)
- Thread safe if memory accesses are ordered
  - no write concurrency

- **Queue operation**
  - New data is lost when full
  - overwrite old element when full

- **Multiple consumers & producers**
## Thread Safe Producer Consumer Queue

### Writing elements

```cpp
bool WriteElement(Type &Element) {
    int next = (p_Write + 1) % Size;
    if(next != p_Read) {
        Data[p_Write] = Element;
        p_Write = next;
        return true;
    } else
        return false;
}
```

### Reading elements

```cpp
bool ReadElement(Type &Element) {
    if(p_Read == p_Write)
        return false;
    int next = (p_Read + 1) % Size;
    Element = Data[p_Read];
    p_Read = next;
    return true;
}
```