Embedded Operating Systems

Agenda

- Synchronization
    - Semaphore
    - Condition variable
    - Spinlock
- Threading model
- Thread group vs Process group vs Session
- Memory Management

POSIX

- UNIX has many flavours -- UNIX, BSD UNIX, Solaris, AIX, IRIX, HP-UX, DG-UX, Xenix, SCO UNIX, Mac OS X, Linux, ...
- To standardize user/application interaction, a standard was developed -- POSIX by IEEE.
- POSIX -- Portable Operating System Interface for X-Windows/UNIX.

Linux - POSIX Synchronization

- POSIX Synchronization APIs
    - Semaphore
    - Mutex
    - Condition variables
- Linker flag: -lpthread

Semaphore

- Same concept as of UNIX (OS) semaphore.
    - P(s) -- Decrement operation or Wait operation
    - V(s) -- Increment operation or Signal operation

- POSIX APIs:
    - `#include <semaphore.h>`
    - sem_t --> Data type to represent semaphore object
        - Semaphore has "single" counter in it.
    - sem_init() --> initialize to a count.
    - sem_wait() --> P() operation
    - sem_post() --> V() operation
    - sem_destroy() --> destroy the semaphore.
    - sem_trywait(), sem_timedwait()
    - Refer manual ...

sem_init()

- To create and initialize semaphore.
- ret = sem_init(&s, pshared, init_count);
    - arg1: Semaphore id (out param)
    - arg2: pshared is similar to SEM_KEY.
        - 0 for synchronizing threads in the same process.
        - unique key for synchronizing threads in the across processes.
    - arg3: initial semaphore count.

sem_destroy()

- sem_destroy(&s);
    - arg1: Id of semaphore to be destroyed

sem_wait()

- P(s) operation
- sem_wait(&s);
    - arg1: Id of semaphore on which P() operation is to be performed

sem_post()

- V(s) operation
- sem_post(&s);
    - arg1: Id of semaphore on which V() operation is to be performed

Condition variable

- A thread can wait for another thread completing some task.
- Condition variable always work in context with some mutex.
- POSIX APIs
    - pthread_cond_t <-- represent condition variable.
    - pthread_cond_init()
    - pthread_cond_destroy()
    - pthread_cond_wait()
    - pthread_cond_signal()
    - pthread_cond_broadcast()

pthread_cond_init()

- Initialialize given cond variable with give attributes.
- pthread_cond_init(&c, &ca);
    - arg1: id of cond var (out param)
    - arg2: cond var attributes -- NULL means default

pthread_cond_destroy()

- Destroy the cond var
- pthread_cond_destroy(&c)
    - arg1: id of cond var to be destroyed

pthread_cond_wait()

- Unlock the given mutex.
- Block the current thread on condition variable.
- When wake-up (due to pthread_cond_signal() / pthread_cond_broadcast()) lock the mutex again and resume execution.
- pthread_cond_wait(&c, &m)
    - arg1: id of cond var on which the thread is to be blocked
    - arg2: id of mutex to be unlocked

pthread_cond_signal()

- Wake-up "one" of the thread sleeping on condition variable.
- pthread_cond_signal(&c)
    - arg1: id of cond var
- Woken-up thread will lock the mutex and resume the execution.

pthread_cond_broadcast()

- Wake-up all the threads sleeping on the condition variable.
- pthread_cond_broadcast(&c)
    - arg1: id of cond var
- Woken-up threads will try to lock the mutex and resume the execution.
- Winning thread will continue with execution, while other threads will blocked again.

Spinlock

- Spinlock is harware/architecture based synchronization mechanism.
- Two processes cannot access spinlock simultaneously, in uni-processor or multi-processor environment.
- Semaphore/mutex should not be used in interrupt context (ISR), because ISR should never sleep.
- Semaphore is internally a counter and mutex is a lock. If multiple processes try to use the Semaphore/mutex simultaneously, there may be race condition for Semaphore/mutex itself.

Solution 1

- When Semaphore count is incremented (V) or decremented (P), the processor interrupts can be disabled. This will ensure that no other process will preempt P and V operation, and thus no race condition for Semaphore.
- Semaphore P operation:
  - step 1: disable interrupts.
  - step 2: P operation (decrement and block if negative).
  - step 3: enable interrupts.
- Semaphore V operation:
  - step 1: disable interrupts.
  - step 2: V operation (increment and unblock if any process sleeping).
  - step 3: enable interrupts.
- This solution is applicable for uni-processor system. In multi-processor system, if interrupts are disabled, it will disable interrupts of current processor only. The process running on another processor can still access the Semaphore.
- Disabling (masking) interrupts also increases interrupt latencies.

Solution 2

- When Semaphore count is incremented (V) or decremented (P), some hardware level synchronization mechanism should be used to access Semaphore by only one process.
- Spinlock is hardware level synchronization mechanism. Spinlocks are implemented using bus-holding instructions -- test_and_set() kind i.e. only one task can access the bus at a time. The test and set operations are done in same bus cycle i.e. bus remains locked until both operations are completed.
  - Example: ARM7 -- SWP, ARM Cortex -- LDREX, STREX.
- https://developer.arm.com/documentation/den0013/d/Multi-core-processors/Exclusive-accesses

Spinlock working

- Spinlock is a variable -- 0 (unlocked/available) or 1 (locked/busy).
- Spinlock initialization. It is unlocked.

```
lock = 0;
```

- To lock a spinlock: If spinlock is busy, wait (busy wait loop); otherwise lock.

```
while(lock == 1)
    ;
lock = 1;
```

- To unlock a spinlock, clear it.

```
lock = 0;
```

ARM7 SWP instruction

- Spinlock implementation

```
spin:
    MOV r1, =lock
    MOV r2, #1

    SWP r3, r2, [r1] ; hold the bus until complete

    CMP r3, #1
    BEQ spin
```

- SWP instruction
  - SWP r3, r2, [r1]
    - r3 = *r1 and *r1 = r2;

ARM Cortex LDREX/STREX

- Refer: Joseph Yiu

```
spin_lock:                  ; an assembly function to get the lock
    LDR        r0, =Lock_Var
    MOVS       r2, #1           ; use for locking STREX
lock_loop:
    LDREX      r1, [r0]
    CMP        r1, #0
    BNE        lock_loop        ; It is locked, retry again
    STREX      r1, r2, [r0]     ; Try set Lock_Var to 1 using STREX
    CMP        r1, #0           ; Check return status of STREX
    BNE        lock_loop        ; STREX was not successful, retry
    DMB                         ; Data Memory Barrier
    BX         LR               ; Return
```

```
spin_unlock:               ; an assembly function to free the lock
    LDR r0, =Lock_Var
    MOVS r1, #0
    DMB                         ; Data Memory Barrier
    STR r1, [r0]                ; Clear lock
    BX LR                       ; Return
```

Semaphore vs Mutex vs Spinlock

- Spinlock are busy wait (not sleep).
- Can be used in interrupt context.
- Available only in kernel space.

Starvation vs Deadlock

- Deadlock
  - Deadlock happens when all four conditions hold true at same time i.e. No preemption, Mutual exclusion, Hold and wait, and Circular wait.
  - The processes involved in deadlock are in blocked state. They are in waiting queue (not in ready queue).

    - Deadlock can be prevented by designing systems properly and/or avoided using some algorithms like safe state, resource allocation graph, or banker algorithm.
- Starvation
    - Due to other high priority processes some low priority process is not getting CPU time for the execution.
    - The starved process is in ready state. They are in ready queue.
    - The starved process's priority can be increased dynamically, so that it will be scheduled (later). This technique is called as "aging".

Threading models

- Threads created by thread libraries are used to execute functions in user program. They are called as "user threads".
- Threads created by the syscalls (or internally into the kernel) are scheduled by kernel scheduler. They are called as "kernel threads".
- User threads are dependent on the kernel threads. Their dependency/relation (managed by thread library) is called as "threading model".
- There are four threading models:
    - Many to One
    - Many to Many
    - One To One
    - Two Level Model

Many to One

- Many user threads depends on single kernel thread.
- If one of the user thread is blocked, remaining user threads cannot function.
- Example:

Many to Many

- Many user threads depend on equal or less number of kernel threads.
- If one of the user thread is blocked, other user thread keep executing (based on remaining kernel threads).
- Example:

One To One

- One user thread depends on one kernel thread.

- Example:

Two Level Model

- OS/Thread library supports both one to one and many to many model
- Example:

Process group vs Thread group

- Session
  - Set of commands given in a shell --> "session".
  - Shell program is leader of the session.
  - Shell process id is referred as session id (sid).
  - terminal> ps -o sid,pid,cmd
- Process group or Job ** Set of processes executed under single command --> "process group".
  - terminal> cat -n file | head -15 | tail -n +5 | sort
  - This command is group of 4 processes.
  - The first process in the command is leader of the process group (in this example: cat).
  - Process group leader pid is referred as process group id (pgid).
  - terminal> ps -o pgid,pid,cmd
- Thread group
  - A multi-threaded process --> "thread group".
  - The main thread (process) is leader of the thread group.
  - Thread group leader tid is referred as thread group id (tgid).
  - All single-threaded process have single thread in their thread group.
  - terminal> ps -o tgid,pid,cmd

Memory Management

- In multi-programming OS, multiple programs are loaded in memory.
- RAM memory should be divided for multiple processes running concurrently.
- Memory Mgmt scheme used by any OS depends on the MMU hardware used in the machine.
- There are three memory management schemes are available (as per MMU hardware).

1. Contiguous Allocation
2. Segmentation
3. Paging

Assignments

1. In first thread, print "SUNBEAM" and in second thread print "INFOTECH". Use "semaphore" to ensure that INFOTECH is printed only after SUNBEAM.
2. Implement producer consumer across two processes using POSIX semaphores and Mutexes. Hints for communication across the processes.
    - Hint 1: Semaphore and Mutex must be in shared memory.
    - Hint 2: Mutex pshared attribute should be set to PTHREAD_PROCESS_SHARED.
    - Hint 3: Semaphore should be created with non-zero key (arg2 of sem_init()).
    - Hint 4: Use signal handlers to properly cleanup shared memory and synchronization objects.