Embedded Operating Systems

Agenda

- Mutex
- Multi-Threading
  - Concepts
  - Thread creation
  - Cancellation
  - Signal
  - fork() in multi-threaded program
  - POSIX Semaphore
  - POSIX Mutex
  - POSIX Condition variable
  - Threading models
  - Threads in Linux

Mutex

- Mutex is used to ensure that only one process can access the resource at a time.
- Functionally it is similar to "binary semaphore".
- Mutex can be unlocked by the same process/thread, which had locked it.

Semaphore vs Mutex

- S: Semaphore can be decremented by one process and incremented by same or another process.
- M: The process locking the mutex is owner of it. Only owner can unlock that mutex.
- S: Semaphore can be counting or binary.
- M: Mutex is like binary semaphore. Only two states: locked and unlocked.
- S: Semaphore can be used for counting, mututal exclusion or as a flag.
- M: Mutex can be used only for mutual exclusion.

Multi-Threading

Thread concept

- Threads are used to execute multiple tasks concurrently in the same program/process.
- Thread is a light-weight process.
  - For each thread new control block and stack is created. Other sections (text, data, heap, ...) are shared with the parent process.
  - Inter-thread communication is much faster than inter-process communication.
  - Context switch between two threads in the same process is faster.
- Thread stack is used to create function activation records of the functions called/executed by the thread.

Process vs Thread

- In modern OS, process is a container holding resources required for execution, while thread is unit of execution/scheduling.
- Process holds resources like memory, open files, IPC (e.g. signal table, shared memory, pipe, etc.).
- PCB contains resources information like pid, exit status, open files, signals/ipc, memory info, etc.
- CPU time is allocated to the threads. Thread is unit of execution.
- TCB contains execution information like tid, state, scheduling info (priority, sched algo, time left, ...), Execution context, Kernel stack, etc.
- terminal> ps -e -o pid,nlwp,cmd
- terminal> ps -e -m -o pid,tid,nlwp,cmd
- terminal> cat /proc/pid/maps

main thread

- For each process one thread is created by default called as main thread.
- The main thread executes entry-point function of the process.
- The main thread use the process stack.
- When main thread is terminated, the process is terminated.
- When a process is terminated, all threads in the process are terminated.=

Thread creation

- Each thread is associated with a function called as thread function/procedure.

- The thread terminates when the thread function is completed.
- Steps for thread creation
    - step1: Implement thread function
    - step2: Create thread using syscall/library

Thread library/system calls

- Linux syscall: clone() creates a thread or process.
- UNIX/Linux: POSIX thread library: pthread_create()
- Windows: Win32 API: CreateThread()

POSIX Thread Functions

pthread_create()

- Create a new thread.
- arg1: posix thread id (out param)
- arg2: thread attributes -- NULL means default attributes
    - stack size
    - scheduling policy
    - priority
- arg3: address of thread function
    - void* thread_function(void*);
        - arg1: void* -- param to the thread function (can be of any type, array or struct).
        - returns: void* -- result of thread (can be of any type)
- arg4: param to thread function
- returns: 0 on success.

Thread Programming

Join thread

- The current thread wait for completion of given thread and will collect return value of that thread.
- `pthread_join(th_id, (void**)&res)`;
    - arg1: given thread (for which current thread is to blocked).
    - arg2: address of result variable (out param to collect result of the given thread)

Thread termination

- When thread function is completed, the thread is terminated.
- To terminate current thread early use pthread_exit() function.
- pthread_exit(result);
    - arg1: result (void*) of the current thread

Thread cancellation

- A thread in a process can request to cancel execution of another thread.
- pthread_cancel(tid)
    - arg1: id of the thread to be cancelled.
- terminal> man pthread_cancel

Signal to thread

- pthread_kill() is used to send signal to a thread (in current process).
- ret = pthread_kill(tid, signum);
    - arg1: thread id to whom signal is to be sent.
    - arg2: signal number (SIGINT, SIGTERM, SIGKILL, ...);
- However signal handlers are applicable for the whole process (not individual thread) -- registered using sigaction().
- If signal handler is registered, it will be executed by the thread to whom signal is sent.
- If signal handler is not registered, default action is followed (for the whole process).

fork() in multi-threaded program

- When fork() is called by one of the thread in a multi-threaded process, a new process is created duplicating calling process.
- However, new process will have single thread and it continues executing copy of calling thread.

Thread Synchronization

- Pre-requisite
  - Race condition
  - Synchronization
- The text, data, rodata, and heap sections for any thread are shared with the parent process. So data placed in this section is directly accessible to all threads in the same process.
- However, access to such data should be synchronized to avoid the race condition.

UNIX Synchronization

- Semaphore
  - P(s)
  - V(s)
- Applications
  - Counting
  - Mutual exclusion
  - Flag/Condition
- System calls
  - semget()
  - semctl()
  - semop()

Linux - Kernel Synchronization

- Semaphore
- Mutex
- Spinlocks

Linux - POSIX Synchronization

- POSIX Synchronization APIs
  - Semaphore

    - Mutex
    - Condition variables
- Linker flag: -lpthread

Mutex

- Mutex is used to ensure that only one process/thread can access the resource at a time -- mutual exclusion.
- Functionally, it is similar to "binary semaphore".
- Using semaphore mutual exclusion can be implemented.

```
s = 1;

P(s);   // decrement
// use resource
V(s);   // increment
```

- Mutex simplifies mutual exclusion and also provide more efficient & readable implementation.
- Mutex has two states: locked and unlocked.

```
lock(m);
// use resource
unlock(m);
```

- The process/thread locking the mutex is owner of that mutex.
- Only owner can unlock the mutex.
- If a thread try to lock a mutex, which is already locked by another thread; then the another thread will be blocked until mutex is unlocked by the owner thread.

- POSIX APIs:
    - `#include <pthread.h>`
    - pthread_mutex_t --> Data type to represent mutex object
    - pthread_mutex_init()
    - pthread_mutex_lock()
    - pthread_mutex_unlock()
    - pthread_mutex_destroy()
    - Refer manual.
    - sudo apt-get install manpages-posix manpages-posix-dev

pthread_mutex_init()

- To create mutex (default state is unlocked).
- pthread_mutex_init(&m, &attr);
    - arg1: address of mutex id (out param)
    - arg2: mutex attribute address (pthread_mutex_attr_t*)
        - Mutex attributes: type, process shared, ...
        - NULL means default attributes.
            - type = PTHREAD_MUTEX_NORMAL
            - process shared = false (only for threads in current process)

pthread_mutex_lock()

- pthread_mutex_lock(&m);
    - arg1: Id of mutex to be locked.

pthread_mutex_unlock()

- pthread_mutex_unlock(&m);
    - arg1: Id of mutex to be unlocked.

pthread_mutex_destroy()

- pthread_mutex_destroy(&m);
  - arg1: Id of mutex to be destroyed.

Mutex Rules

- A thread cannot lock the same mutex twice.

```
lock(m);
lock(m); // block the thread
// ...
unlock(m);
```

- A thread locking mutex is owner of that mutex. Only that thread can unlock the mutex.
- A thread cannot unlock a mutex which is not in locked state.

Types of POSIX mutex

- PTHREAD_MUTEX_NORMAL:
  - Locking a mutex twice within a single thread cause deadlock (thread is locked permanently).
  - Unlocking a mutex owned by some other thread or already unlocked mutex, will produce undefined results.
  - Since this mutex does not check mutex rules, they are more efficient/faster.
- PTHREAD_MUTEX_ERRORCHECK:
  - Locking a mutex twice within a single thread will fail.
  - Unlocking a mutex owned by some other thread or already unlocked mutex, will fail.
  - This is slower than normal mutex; but useful for debugging.
- PTHREAD_MUTEX_RECURSIVE:
  - Mutex will maintain lock count. When same thread lock mutex multiple times, lock count will be incremented.
  - When unlocked, the lock count will decrement. The mutex will be released when lock count drops to zero.
  - Unlocking a mutex owned by some other thread or already unlocked mutex, will fail.

- Mutex type can be set using pthread_mutexattr_settype(&attr, type);

```
pthread_mutexattr_t ma;
pthread_mutexattr_init(&ma); // init mutex attr to default
#ifdef DEBUG
    pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_ERRCHECK);
#else
    pthread_mutexattr_settype(&ma, PTHREAD_MUTEX_NORMAL);
#endif
pthread_mutex_init(&m, &ma);
// ...
```

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.

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:

Threads in Linux - clone()

- In Linux, processes and threads both are referred as "tasks". For each task, a separate "task_struct" is created (instead of PCB or TCB).
- Process can be seen as a new task that doesn't share any section with parent task, but sends SIGCHLD signal to the parent upon its termination.

```
child_task_id = clone(task_fn, stack, SIGCHLD, NULL);
```

- Thread can be seen as a new task that shares parent's virtual address space, open files, fs and signal handler table.

```
child_task_id = clone(task_fn, stack, CLONE_VM|CLONE_FILES|CLONE_FS|CLONE_SIGHAND, NULL);
```

Assignments

1. Create a thread to sort given array of 10 integers using selection sort. Main thread should print the result after sorting is completed.
    - Hint: Pass array to thread function (via arg4 of pthread_create()).

```
void* thread_func(void *param) {
    int *arr = (int*)param;
    // ... code to sort the array
    return NULL;
}
```

2. Create a thread to sort given array of "n" integers using bubble sort. Main thread should print the result after sorting is completed.
    - Hint: `struct array { int *arr; int size; }`
    - Pass struct address to thread function (via arg4 of pthread_create()).
3. One thread prints "SUNBEAM" continuously, and other thread prints "INFOTECH" continhously. Only one should print at a time starting with "SUNBEAM". Hint: using semaphores.
4. Optional - 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.