Embedded Operating Systems

Agenda

- Multi-Threading
  - Thread creation
  - Join thread
  - Cancel thread
  - Signals
- Synchronization
  - Semaphore
  - Mutex
  - Condition Variable

Multi-Threading

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()

clone() system call

- 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);
```

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.

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).

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

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 same as "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 blocked).
  - Unlocking a mutex owned by some other thread or already unlocked mutex, will produce undefined results.
  - Since this mutex do 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);
// ...
```

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()).
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()).