Linux System Programming

Faculty: Rajesh Sola, Srinivas.K & Bharath.G

L&T Technology Services

LTTS GLOBAL ENGINEERING ACADEMY

Inter process communication (IPC)

IPC

Requirement of IPC

- Data exchange
- Synchronization
  - Dependency / Sequencing
  - Mutual Exclusion
- Data exchange → shared memory, message queues, FIFOs/pipes
- Mutual exclusion → semaphore, mutex, spinlocks
- Dependency →semaphores, condition variables / event flags

Semaphores, mutex, message Queues

- Process that updates data is PRODUCER and process take and use is CONSUMER

Semaphores

- Semaphores are used for process and thread synchronization
- Clubbed with message queues and shared memory under the IPC
- Two (2) varieties of semaphores
  - Traditional System V semaphores
  - POSIX semaphores.
- Two (2) types of POSIX semaphores
  - Named and Unnamed
- Kernel level data structure
- Types of usage
  - Binary Semaphore
    - Value of semaphore ranges between 0 & 1
  - Continuous Semaphore
    - Value of semaphore can be 0 (zero) & any positive value

Unnamed Semaphores

- POSIX Unnamed Semaphore calls
  - #include <semaphore.h>, #include <errno.h>

  sem_init(sem_t *sem, int pshared, unsigned int value)
  - sem_init is to initialize unnamed semaphore

  sem_wait(sem_t *sem)
  - sem_wait for the lock of unnamed semaphore
    - Check sem_trywait & sem_timedwait

  sem_post(sem_t *sem)
  - sem_post for the unlock of unnamed semaphore

  sem_destroy(sem_t *sem)
  - sem_destroy destroys the unnamed semaphore

All calls return 0 on success, -1 on error and '**errno'** variable is set to error number

Race conditions

Race conditions

- more than one process accessing same resources will cause resources will be corrupt
- Memory / devices accessed by more than one resources will cause race condition
  - e.g shared printer, concurrent writes to a file POSIX Unnamed Semaphore calls

Process switching scenarios under consideration

- Switching between instructions
- Switching after instructions

Critical section

Critical section

- Process / threads using shared resources is referred as critical section
- During process execution in critical section, no switching allowed
- Among multiple process, only one section should be defined as critical section
- By mutual exclusion, only one critical section will be allowed to access the resources

Mutual exclusion

How to achieve mutual exclusion

- Disable interrupts (for very shorter duration)
  - Limitations
    - for longer duration, inconsistency occurs
    - other CPU can access the resources
- Hardware support instructions
- Atomic operation
  - Resources can't be accessed by other process
- Data bus locking techniques
  - CPU level bus locking techniques

- Above techniques have limitations and not scalable
- Software level solution for mutual exclusion is semaphore & mutex

Named Semaphore

Named semaphore

- Uses internal shared memory for resources access
- Name is given to semaphore and can be access by parent and child
- sem_t *ps; (declare a semaphore variable)
- ps = sem_open("s1", O-CREAT, 0666, 1) (internal shared memory)
- sem_wait(ps) (lock the semaphore)
- sem_post(ps) (unlock the semaphore)
- sem_close(ps) (close semaphore from process)
- sem_unlink(ps) (remove named semaphore)

Mutex

Mutex

- Mutex will have "ownership" as compared to semaphore
- Only locked process(es) / threads can unlock the resources
- Any other process / threads trying to unlock is referred as “unauthorized operation”

Operations of Mutex

- No random operations are not allowed
- Unlocking twice or unlocking before locking is not allowed
- lock in one thread and unlock in another thread is not allowed
- Strictly lock & unlock in the same thread only

- Binary semaphore can be replaced by Mutex

Mutex API's

- #include <pthread.h>
- pthread_mutex_t m1=PTHREAD_MUTEX_INITIALIZER (declare & initialize)
- pthread_mutex_lock(&m1) (lock)
- pthread_mutex_unlock(&m1) (unlock)
- pthread_mutex_destroy (&m1) (destroy)

- Always check return value for Success or Failure

Deadlock

- Two are more processes infinitely blocked (forever) due to circular dependency of resources
  - Arbitrary locking of multiple semaphores
  - Parent & child - unlocking semaphore after waitpid
  - Producer consumer problem - order of locking

Avoid deadlock

- If multiple locks are required, lock all of them at once (atomic locking)
- Don't apply mutual exclusion, before resolving dependency

Queries?

Thank You !

L&T Technology Services