Linux (POSIX) Semaphores - Two types
  
  o Named Semaphores
    
    Introduction
    
    • Semaphore that can be operated on by two processes that
don’t share an address space
    • Implemented in Linux 2.6 kernel
    • Created in the virtual file system under /dev/shm
    • Takes the name: sem.somename
    • File permissions can be set to control which processes can
access
    • sem_open(char* name, int flags, mode_t mode, unsigned int value)
      
      • Used to open a named semaphore
      • Will create the semaphore if it does not already exist
      • mode: specifies access permissions (should be read/write)
      • value: the initial value of the semaphore
    • sem_close(sem_t *sem)
      
      • Closes semaphore for the process that calls it
      • Semaphore still exists in the kernel
    • sem_unlink(char* name)
      
      • Name is removed immediately
      • Semaphore exists until all processes that have it open call
sem_close
      • Named semaphores persist in the kernel until sem_unlink is
called
        • Even if no processes are accessing it
  
  o Unnamed Semaphores (memory-based semaphores)
    
    Introduction
    
    • No name is associated with these semaphores
    • Placed in a region of main memory that is shared between
processes/threads
    • For threads this is done by simply making the semaphore a
global variable
    • sem_init(sem_t *sem, int shared, unsigned int value)
      
      • Initializes an unnamed semaphore
      • shared: Zero for sharing among threads, non-zero for
process sharing
        • If shared between processes, must allocate some
memory using shm_open (shared memory open)
      • value: initial value of the semaphore
    • sem_destroy(sem_t *sem)
      
      • Destroys the semaphore pointed to by sem
      • Not persistent in the kernel
      • If other threads/processes waiting, behavior is undefined
(bad)
  
  o Semaphore operations
• int sem_post(sem_t *sem)
  • Synonymous with *up* presented last time
  • Increments the semaphore by one
  • If another process/thread is waiting on the semaphore, one is woken up
  • Returns zero on success or -1 on failure
    • Always check return value on system calls like this
    • If incrementing would exceed the maximum semaphore value, an error (EOVERFLOW) is put into errno
      • If ignored, an inconsistency will exist
• int sem_wait(sem_t *sem)
  • Synonymous with *down* presented last time
  • Decrements the semaphore by one
    • If *sem* is positive, decrements and immediately returns
    • If *sem* is zero, blocks until it becomes positive (or signal is received)
  • Returns error and sets errno to EINTR if signal received
• int sem_trywait(sem_t *sem)
  • Sometimes a thread/process might desire not to block if the semaphore isn’t positive
    • Useful for situations where other work can be done
  • If decrement can’t be immediately performed, call returns an error instead of blocking
  • Returns -1 and sets errno to EAGAIN
• int sem_timedwait(sem_t *sem, struct timespec *timeout)
  • Sometimes the thread/process might be willing to wait awhile, but not block indefinitely on *sem_wait*
    • Useful for situations where other work *might* become available later, but right now there isn’t anything to do
  • Blocks on the semaphore for a set amount of time
  • Uses struct timespec to specify the absolute time (since 1 January 1970) when the timeout will occur
    • Seconds and nanoseconds specified
  • Returns -1 if timeout occurs before semaphore is incremented
    • Sets errno to ETIMEDOUT
• int sem_getvalue(sem_t *sem, int *val)
  • Used to determine the current value of the semaphore
  • Copies the value of *sem* into *val*
  • Should not be used for synchronization purposes
    • *val* might not reflect the current *sem* when the call returns
Mutex

- Introduction
  - Sometimes the counting feature of a semaphore is not needed
  - Mutex – specialized binary semaphore that provides mutually exclusive access to a critical area
- Properties
  - Zero when unlocked and available to any thread
  - One when a thread is in its critical section
- Implementation that doesn’t require busy waiting

```
mutex_lock:
    TSL REGISTER,MUTEX | copy mutex to register and set mutex to 1
    CMP REGISTER,#0 | was mutex zero?
    JZE ok | if it was zero, mutex was unlocked, so return
    CALL thread_yield | mutex is busy; schedule another thread
    JMP mutex_lock | try again
ok:
    RET | return to caller; critical region entered
```

```
mutex_unlock:
    MOVE MUTEX,#0 | store a 0 in mutex
    RET | return to caller
```

- `mutex_lock`: called right before entering critical region
- `mutex_unlock`: called right after exiting critical region

- Linux (POSIX) mutexes – only one type
  - Generic Operations
    - int pthread_mutex_init(pthread_mutex_t *mutex, attributes)
      - Creates a new mutex
    - int pthread_mutex_destroy(pthread_mutex_t *mutex)
      - Destroys a mutex
      - Removes it from memory
  - Locking/Unlocking operations
    - int pthread_mutex_lock(pthread_mutex_t *mutex)
      - Attempts to lock the mutex
      - If already locked, blocks until the mutex is unlocked and thread is chosen
    - int pthread_mutex_unlock(pthread_mutex_t *mutex)
      - Unlocks the mutex
    - int pthread_mutex_trylock(pthread_mutex_t *mutex)
      - Same behavior of sem_trywait
      - Immediately returns and error and sets errno to EBUSY if the mutex is already locked
    - int pthread_mutex_timedlock(pthread_mutex_t *mutex, struct timespec *timeout)
      - Same behavior as sem_timedwait
Returns error and sets errno to ETIMEDOUT if mutex wasn’t able to be locked before timeout

Wrinkle – Mutexes can be created with four different forms
  - Introduction
    - Specified in attributes of pthread_mutex_init
    - Deadlock: situation where a thread/process blocks forever waiting for a resource it will never get
    - The four different forms of mutexes deal with possible deadlocks in different ways
      - Efficiency is often traded for deadlock detection
  - PTHREAD_MUTEX_NORMAL
    - No deadlock detection
      - If a thread tries to lock a mutex it already has locked, deadlock immediately occurs
      - Undefined behavior:
        - unlocking an unlocked mutex
        - unlocking a mutex that is locked by another thread
      - Dangerous, but efficient
  - PTHREAD_MUTEX_ERRORCHECK
    - Deadlock detection and error checking
      - If a thread tries to lock a mutex it already has, an error will occur
        - errno is set with EDEADLK
      - Errors are also returns for undefined behavior from above
      - Very safe, but inefficient
        - Must check all error codes
  - PTHREAD_MUTEX_RECURSIVE
    - Adds a “lock count” to the mutex (essentially an exclusive semaphore)
      - When a thread locks the mutex, the count increments to one
      - Each time the thread successively locks, it is incremented again
      - The mutex must be unlocked “lock count” times for it to become available to other threads
    - Implications
      - Useful for nested function calls that each lock/unlock the mutex
      - Must be very careful to match the number of locks with unlocks to avoid deadlock
      - Additional overhead for this feature
  - PTHREAD_MUTEX_DEFAULT
    - Features
      - Recursive locking is undefined
- Unlocking an unlocked thread is undefined
- Unlocking when another thread has locked the mutex is undefined

- Fast, but must be sure the code is correct