Operating Systems
CS F372

Synchronization

BIJU K RAVEENDRAN

Semaphores

- The hardware solution is not easy to generalize for more complex problems.
- Special variable called a semaphore is used for signaling
- Semaphore is a variable that has an integer value
  - Apart from initialization, the integer value is accessed only through 2 standard atomic operations
  - May be initialized to a nonnegative number
  - **Wait** operation decrements the semaphore value
  - **Signal** operation increments semaphore value
  - Originally called P() (Dutch proberen – "to test") and V() (verhogen – "to increment")

Semaphores

<pre>wait (S) {
        while (S <= 0)
                ; // no-op
          S - - ;
}
 	
signal (S) {
      S + +;
}</pre>
	<pre>Semaphore mutex;//initialized to 1
	do {
	          wait (mutex);
	       // Critical Section
	    signal (mutex);
	          // remainder section
	} while (TRUE);</pre>

Semaphores

- Counting semaphore – integer value can range over an unrestricted domain
- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore
- Provides mutual exclusion

Implementation

- Disadvantage of the existing mechanism
  - Busy waiting
    - Process is waiting in the CPU without doing any work. This types of semaphore is also called Spin lock (process spins while waiting for the lock)
    - Spin locks may degrade the performance of Uniprocessor – multi programming machines
    - Spin lock may be advantageous in multiprocessor machines (Context switching time can be high than busy waiting time)
    - If locks are expected to be held for a short time then spin locks are useful.

Semaphore Implementation

- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time
- wait() and signal() implementation will be a critical section problem
  - Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution.

Solution for Busy waiting

- Can modify the definition of wait and signal semaphore operations
- When a process executes wait and finds the semaphore value is not positive, the process can block itself
- The block operation places the process into waiting queue associated with the semaphore, and the state of the process is switch to waiting state
- A process that is blocked, waiting on a semaphore S, would be restarted when some other process executes a signal operation
  - The state of the process changes from waiting to ready

Semaphore with NO busy waiting

- With each semaphore there is an associated waiting queue. Each entry in a semaphore has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - block – place the process invoking the operation on the appropriate waiting queue.
  - wakeup – remove one of processes in the waiting queue and place it in the ready queue.

Semaphore with NO busy waiting

- Each semaphore has an integer value & a list of processes
- When a process must wait on a semaphore, it is added to the list of processes
- A signal operation removes one process from the list and awakens that process

Semaphore with NO busy waiting

```
typedef struct {
    int value ;
    struct process *list ;
}semaphore ;
```

Implementation of wait

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

Implementation of signal

```
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process
        P from S->list;
        wakeup(P);
    }
}
```

Semaphore with NO Busy waiting

- This implementation may have negative semaphore values.
  - If the semaphore value is negative its magnitude is the number of processes waiting on the semaphore.
- Use FIFO queue to satisfy bounded waiting
  - Semaphore contains both head & tail pointers to the queue.
- The semaphores are executed atomically by
  - disabling interrupts for uniprocessor systems
  - using appropriate hardware for multiprocessor systems

Semaphore implementation using TestAndSet

```
wait ( s )  {
  while (TestAndSet(s.flag));
  s.count - - ;
  if ( s.count < 0) {
     place this process in
                s.queue;
      s.flag=0;
     block this process;
       }
  else
     s.flag = 0;
}
```

```
signal ( s )  {
while (TestAndSet(s.flag));
  s.count + + ;
  if ( s.count <= 0) {
     remove a process P from
                s.queue;
     place process P on ready list;
  }
  s.flag = 0;
}
```

Semaphore implementation using Interrupts

```
wait ( s )  {
  inhibit interrupts;
  s.count - - ;
  if ( s.count < 0) {
     place this process in
               s.queue;
      block this process and
     allow interrupts;
       }
  else
     allow interrupts;
}
```

```
signal ( s )  {
inhibit interrupts;
  s.count + + ;
  if ( s.count <= 0) {
     remove a process P from
               s.queue;
     place process P on ready list;
  }
  allow interrupts;
}
```

Deadlock & Starvation

- Deadlock – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

- Starvation – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended
- Priority Inversion - Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Implementation of a Counting Semaphore by using Two Binary Semaphores and a Count variable

- Binary semaphore will have an integer value 0 or 1.
- It is simpler to implement, depending on underlying hardware.
- Data structure needed for counting semaphore implementation
  - binary semaphores S1 , S2 ;
  - int C ;
  - Initialize S1 = 1; S2 = 0 ; C = initial value of counting semaphore S.

Implementation of a Counting Semaphore

Wait operation

```
wait ( S1 ) ;
C - - ;
if ( C < 0 ){
    signal ( S1 ) ;
    wait ( S2 );
}
signal ( S1 ) ;
```

Signal Operation

```
wait ( S1 ) ;
C + + ;
if ( C < = 0)
    signal ( S2 ) ;
else
    signal ( S1 ) ;
```

SEMAPHORE OPERATIONS IN C

semaphore.h

- int sem_init (sem_t *sem, int pshared, unsigned int value);
  - Initializes the unnamed semaphore at the address pointed to by sem
  - value specifies the initial value of the semaphore (Negative values are not allowed)
  - pshared = 0 (semaphore is shared between the threads of a process)
  - pshared = non-zero (semaphore is shared between the processes)
  - Returns 0 on success; -1 on error
- int sem_destroy(sem_t *sem)
  - Destroys the semaphore (initialized by sem_init) object
  - No threads should be waiting on the semaphore

- sem_t *sem_open (const char *name, int oflag)
- sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)
  - Creates (open) a new (an existing) semaphore
  - Semaphore is identified by name
  - oflag specifies flags that controls operation of the call (O_CREAT, O_EXCL)
  - Mode specifies the permissions to be specified
  - Value specifies the initial value of the semaphore
  - If oflag with O_CREAT is specified and the semaphore with the same name exists then mode and value arguments are ignored
  - Returns the address of the new semaphore on success . This address is used for other semaphore related operations

- int sem_close (sem_t *sem)
  - Closes the named semaphore referred by sem
  - All the open named semaphores are automatically closed on process termination
- int sem_unlink (const char *name)
  - Removes the named semaphore referred to by sem.
  - The semaphore name is removed immediately.
  - The semaphore is destroyed once all other processes that have the semaphore open close it.
  - On success returns 0 else -1

- int sem_wait(sem_t *sem)
  - Decrements the semaphore count by 1.
  - If semaphore value is greater than 0 then decrement proceeds and the function returns immediately
  - If the semaphore currently has the value zero, then the call blocks until either
    - It becomes possible to perform the decrement (semaphore value raises above 0) or
    - A signal handler interrupts the call
- int sem_trywait(sem_t *sem)
  - Same as sem_wait except
    - If the decrement can not be immediately performed then call returns an error (EAGAIN) instead of blocking

- int sem_timedwait(sem_t *sem, __const struct timespec *time)
  - Same as sem_wait except
    - Argument “time” specifies a limit on the amount of time that the call should block if the decrement can not be immediately performed.
    - “time” argument points to a structure that specifies an absolute time out in seconds and nanoseconds since (00:00:00, 1st January 1970)

```
struct timespec {
    time_t tv_sec;      /* Seconds */
    long   tv_nsec;     /* Nanoseconds [0 .. 999999999] */
};
```

    - If the time out has already expired at the time of the call and semaphore could not be locked immediately then this call fails with a timeout error (ETIMEDOUT)

- sem_post (sem_t *sem)
  - Automatically increases the count by 1
  - This function never blocks
- sem_getvalue(sem_t *sem, int *sval)
  - Stores the count of the semaphore to sval
  - If one or more processes / threads are blocked waiting to lock the semaphore with sem_wait POSIX provides two possibilities for the value returned in sval
    - 0 is returned (Linux follows this)
    - A negative number whose absolute value is the count of the number of processes / threads currently blocked
  - Returns 0 on success; -1 otherwise

- Initialize a spin lock
  - int pthread_spin_init (pthread_spinlock_t *lock, int pshared)
- Destroy a spin lock
  - int pthread_spin_destroy (pthread_spinlock_t *lock)
- Try locking a spin lock
  - int pthread_spin_trylock (pthread_spinlock_t *lock)
- Lock a spin lock
  - int pthread_spin_lock (pthread_spinlock_t *lock)
- Unlock a spin lock
  - int pthread_spin_unlock (pthread_spinlock_t *lock)

- Initialize a mutex
  - int pthread_mutex_init (pthread_mutex_t *mutex, pthread_mutexattr_t *mutexattr)
- Destroy a mutex
  - int pthread_mutex_destroy (pthread_mutex_t *mutex)
- Try locking a mutex
  - int pthread_mutex_trylock (pthread_mutex_t *mutex)
- Lock a mutex
  - int pthread_mutex_lock (pthread_mutex_t *mutex)
- Timed Lock
  - int pthread_mutex_timedlock (pthread_mutex_t *mutex, struct timespec *time)

- int pthread_mutex_unlock (pthread_mutex_t *mutex)
- int pthread_mutexattr_init (pthread_mutexattr_t *attr)
- int pthread_mutexattr_destroy (pthread_mutexattr_t *attr)
- int pthread_mutexattr_getpshared (pthread_mutexattr_t *attr, int *pshared)
- int pthread_mutexattr_setpshared (pthread_mutexattr_t *attr, int pshared)
  - PTHREAD_PROCESS_SHARED
  - PTHREAD_PROCESS_PRIVATE (Default)

- int pthread_mutexattr_gettype (pthread_mutexattr_t *attr, int *kind)
- int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int kind)
- “kind” value can be of 4 types
  - PTHREAD_MUTEX_NORMAL
    - Does not detect deadlock
    - A thread attempting to relock this mutex without unlocking it shall deadlock
    - Attempting to unlock a mutex locked by a different thread results in undefined behavior
    - Attempting to unlock an unlocked mutex results in undefined behavior.

- PTHREAD_MUTEX_RECURSIVE
  - A thread attempting to relock this mutex without first unlocking it shall succeed in locking the mutex.
  - Deadlock because of relocking cannot occur
  - Multiple locks of this mutex shall require the same number of unlocks to release the mutex before another thread can acquire the mutex
  - A thread attempting to unlock a mutex which another thread has locked shall return with an error
  - A thread attempting to unlock an unlocked mutex shall return with an error.
- PTHREAD_MUTEX_DEFAULT
  - Attempting to recursively lock a mutex of this type results in undefined behavior.
  - Attempting to unlock a mutex of this type which was not locked by the calling thread results in undefined behavior
  - Attempting to unlock a mutex of this type which is not locked results in undefined behavior

- PTHREAD_MUTEX_ERRORCHECK
  - Provides error checking
  - A thread attempting to relock this mutex without first unlocking it shall return with an error
  - A thread attempting to unlock a mutex which another thread has locked shall return with an error
  - A thread attempting to unlock an unlocked mutex shall return with an error.
- int pthread_mutexattr_getprotocol (pthread_mutexattr_t *attr, int *protocol)
- int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr, int protocol)
  - PTHREAD_PRIO_NONE, PTHREAD_PRIO_INHERIT, PTHREAD_PRIO_PROTECT

- PTHREAD_PRIO_NONE
  - Thread's priority and scheduling shall not be affected by its mutex ownership
- PTHREAD_PRIO_INHERIT
  - Inherit the priority of the highest priority thread waiting on any of the mutexes owned by this thread and initialized with this protocol
- PTHREAD_PRIO_PROTECT
  - The thread shall execute at the higher of its priority or the highest of the priority ceilings of all the mutexes owned by this thread and initialized with this attribute (regardless of whether other threads are blocked on any of these mutexes or not)