Concurrency

Remainging Chapter 32

OSTEP Chapter 26 - Concurrency: An Introduction

- Threads share the same address space and thus can access the same data
- Each thread has its own private set of registers it uses for computation; thus, if there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a context switch must take place
- register state of T1 must be saved and the register state of T2 restored before running T2
- we'll need one or more **thread control blocks (TCBs)** to store the state of each thread of a process
- **Note:** The address space need not be changed
- Instead of a single stack in the address space, there will be one per thread
- Thus, any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is sometimes called thread-local storage, i.e., the stack of the relevant thread.
- Reasons for using threads:
    - **Parallelism**: A task may be broken into multiple subtasks and each can be performed parallely, thus reducing the overall time taken to complete the task
    - avoid blocking of program due to I/O
    - threads share an address space and thus make it easy to share data, and hence are a natural choice when constructing these types of programs. Processes are a more sound choice for logically separate tasks where little sharing of data structures in mem◁ory is needed
- ```
  counter = counter + 1
  ```

  changes to

  ```
  mov 0x8049a1c, %eax
  add $0x1, %eax
  mov %eax, 0x8049a1c
  ```

  and thus for a function

  ```
  static volatile int counter = 0;
  void *mythread(void *arg) {
      printf("%s: begin\n", (char *) arg);
      int i;
  ```
  ```

```
        for (i = 0; i < 1e7; i++) {
            counter = counter + 1;
        }
        printf("%s: done\n", (char *) arg);
        return NULL;
    }
```

If two threads are created simultaneously, it may not always return 2e7 as the value of counter at the end.

- This is called a **race condition** and the code segment is called **critical section**.
- A critical section is a piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread
- Atmocity is simply **"all or nothing"**. Either everything will take place in one step or without interruption or nothing at all
- Sometimes, the grouping of many actions into a single atomic action is called a transaction, an idea developed in great detail in the world of databases and transaction processing
- what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call synchronization primitives
- there is another common interaction that arises, where one thread must wait for another to complete some action before it continues. This interaction arises, for example, when a process performs a disk I/O and is put to sleep; when the I/O completes, the process needs to be roused from its slumber so it can continue.

OSTEP Chapter 27 - Interlude: Thread API

OSTEP Chapter 28 - Locks

- Programmers annotate source code with locks, putting them around critical sections, and thus ensure that any such critical section executes as if it were a single atomic instruction
- ```
  lock_t mutex; // some globally-allocated lock 'mutex'
  ...
  lock(&mutex);
  balance = balance + 1;
  unlock(&mutex);
  ```
- lock is just a variable
- The lock variable holds the state of lock at a given instant, either it is held (locked) or not held (unlocked)
- Only one thread can hold a lock and thus can be in the critical section
- Other information like which thread holds the lock or a priority queue for threads storing the order in which lock may be given to the threads can be stored in this data structure
  ```

- Calling the routine lock() tries to acquire the lock; if no other thread holds the lock (i.e., it is free), the thread will acquire the lock and enter the critical section; this thread is sometimes said to be the owner of the lock. If another thread then calls lock() on that same lock variable (mutex in this example), it will not return while the lock is held by another thread; in this way, other threads are prevented from entering the critical section while the first thread that holds the lock is in there
- Once this thread calls unlock(), any other thread waiting for the lock may take the lock
- The name that the POSIX library uses for a lock is a **mutex**, as it is used to provide **mutual exclusion** between threads
- instead of one big lock that is used any time any critical section is accessed (a coarse-grained locking strategy), one will often protect different data and data structures with different locks, thus allowing more threads to be in locked code at once (a more fine-grained approach)
- To evaluate if a lock works well, we perform the following checks:
    1. provide mutual exclusion
    2. fairness, does every thread gets an equal share of the time holding the lock
    3. Performance, the time overheads for a single or multiple threads while contending for a lock
    4. How does the lock perform if multiple CPUs are involved

Controlling Interrupts

First kind of lock for single-core CPUs

```
    void lock() {
        DisableInterrupts();
    }
    void unlock() {
        EnableInterrupts();
    }
```

- this approach requires us to allow any calling thread to perform a privileged operation (turning interrupts on and off)
- We have to trust that this facility is not abused
- a greedy program could call lock() at the beginning of its execution and thus monopolize the processor; worse, an errant or malicious program could call lock() and go into an endless loop
- the approach does not work on multiprocessors
- threads will be able to run on other processors, and thus could enter the critical section. As multiprocessors are now commonplace, our general solution will have to do better than this

A Failed Attempt: Just Using Loads/Stores

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}
```

```
void lock(lock_t *mutex) {
    while (mutex->flag == 1) // TEST the flag
    ; // spin-wait (do nothing)
    mutex->flag = 1; // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

- Two problems:
    - Correctness: Thread T1 can process the loop and interrupt and T2 can process the loop and take the flag and then T1 schedules and starts processsing from the point where it also makes flag = 1. Thus this approach does not provide mutual exclusion
    - Performance: Spin waiting, can lead to a lot of time waste on a uniprocessor

Building Working Spin Locks with Test-And-Set

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new; // store ’new’ into old_ptr
    return old; // return the old value
}
```

- Test and Set is an atomic instruction and can be expressed using the above C code

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0: lock is available, 1: lock is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
    ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

- Ensures correctness, implements simple **spin lock**
- Requires preemptive scheduler as a thread spinning will never relinquish the CPU

- This lock is **not fair** and may lead to starvation of some thread
- The performance is not good for a single CPU, in case there are N threads scheduled each contending for the lock, the one that holds the lock may be scheduled after N context switches and thus a lot of wastage of CPU cycles.
- For a multi CPU system, this lock may still work effectively if the number of threads equal the number of CPUs

Compare and Swap

- Also called **compare-and-exchange** on x86. Atomic instruction, compares with the expected value, if found true, swaps with the new one.

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int original = *ptr;
    if (original == expected)
        *ptr = new;
    return original;
}
```

- Only change in lock() implementation

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
    ; // spin
}
```

- Similar to test-and-set

Load-linked and Store-conditional

- simply fetches a value from memory and places it in a register. The key difference comes with the store-conditional, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place. In the case of success, the storeconditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and 0 is returned

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no update to *ptr since LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    }
```

```
        else {
            return 0; // failed to update
        }
    }


    void lock(lock_t *lock) {
        while (1) {
            while (LoadLinked(&lock->flag) == 1)
                ; // spin until it's zero
            if (StoreConditional(&lock->flag, 1) == 1)
                return; // if set-it-to-1 was a success: all done
                // otherwise: try it all over again
        }
    }

    void unlock(lock_t *lock) {
        lock->flag = 0;
    }
```

- The key feature of these instructions is that only one of these threads will succeed in updating the flag to 1 and thus acquire the lock; the second thread to attempt the store-conditional will fail (because the other thread updated the value of flag between its load-linked and store-conditional) and thus have to try to acquire the lock again

Fetch-And-Add

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

Ticket Lock implementation

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
```

```
        ; // spin
}

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

- Instead of a single value, this solution uses a ticket and turn variable in combination to build a lock. The basic operation is pretty simple: when a thread wishes to acquire a lock, it first does an atomic fetch-and-add on the ticket value; that value is now considered this thread's "turn" (myturn). The globally shared lock->turn is then used to determine which thread's turn it is; when (myturn == turn) for a given thread, it is that thread's turn to enter the critical section. Unlock is accomplished simply by incrementing the turn such that the next waiting thread (if there is one) can now enter the critical section
- Ensures fairness

A Simple Approach: Just Yield, Baby

Working of perfomance now!!

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
    yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

- Thus, the yielding thread essentially deschedules itself
- moves the process from running state to ready state
- Still doesn't overcome the excessive context switch overheads faced while handling say 100 threads
- Also fairness is not yet guaranteed

Using Queues: Sleeping Instead Of Spinning

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
```

```
    m->flag = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
        // (for next thread!)
    m->guard = 0;
}
```

- we will use the support provided by Solaris, in terms of two calls: park() to put a calling thread to sleep, and unpark(threadID) to wake a particular thread as designated by threadID
- First, we combine the old test-and-set idea with an explicit queue of lock waiters to make a more efficient lock.
- Second, we use a queue to help control who gets the lock next and thus avoid starvation.
- A thread might be interrupted while acquiring or releasing the lock, and thus cause other threads to spin-wait for this one to run again.
- **However, the time spent spinning is quite limited (just a few instructions inside the lock and unlock code, instead of the user-defined critical section), and thus this approach may be reasonable.**
- Note: you can't call park() before setting guard to 0, because if you do, the thread goes to sleep and never wakes up because it has the guard lock and thus no thread can set flag to 0, and can wake up the sleeping thread thus the code gets stuck
- Also the flag is never set to 0, if there is a thread waiting in sleeping queue, because once the sleeping thread gets scheduled, it wakes up as if it holds the lock and setting flag to 0 in between may result in it having lock, even though flag is set to 0

- Another problem, after setting guard to 0, and before calling park(), the thread descedules and another thread releases the lock, also checks the sleep queue and thus finds nothing, the original thread then calls park() and may go to sleep forever called **wakeup/waiting race**
- Solaris solves this problem by adding a third system call: setpark(). By calling this routine, a thread can indicate it is about to park. If it then happens to be interrupted and another thread calls unpark before park is actually called, the subsequent park returns immediately instead of sleeping. The code modification, inside of lock(), is quite small:

```
queue_add(m->q, gettid());
setpark(); // new code
m->guard = 0;
```

Linux based futex lock

```
void mutex_lock (int *mutex) {
    int v;
    /* Bit 31 was clear, we got the mutex (the fastpath) */
    if (atomic_bit_test_set (mutex, 31) == 0)
        return;
    atomic_increment (mutex);
    while (1) {
        if (atomic_bit_test_set (mutex, 31) == 0) {
            atomic_decrement (mutex);
            return;
    }
    /* We have to waitFirst make sure the futex value
        we are monitoring is truly negative (locked). */
    v = *mutex;
    if (v >= 0)
        continue;
    futex_wait (mutex, v);
    }
}

void mutex_unlock (int *mutex) {
    /* Adding 0x80000000 to counter results in 0 if and
        only if there are not other interested threads */
    if (atomic_add_zero (mutex, 0x80000000))
        return;

    /* There are other threads waiting for this mutex,
        wake one of them up. */
    futex_wake (mutex);
}
```

Miscellaneous

- priority inversion: lower priority thread gets the lock and once higher priority thread is scheduled, it may never gets to run and thus lock is never released
- Dekker's algorithm:

```
int flag[2];
int turn;
void init() {
    // indicate you intend to hold the lock w/ 'flag'
    flag[0] = flag[1] = 0;
    // whose turn is it? (thread 0 or 1)
    turn = 0;
}
void lock() {
    // 'self' is the thread ID of caller
    flag[self] = 1;
    // make it other thread's turn
    turn = 1 - self;
    while ((flag[1-self] == 1) && (turn == 1 - self))
    ; // spin-wait while it's not your turn
}
void unlock() {
    // simply undo your intent
    flag[self] = 0;
}
```

OSTEP Chapter 31 - Semaphores

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

> - The second argument to sem init() will be set to 0 in all of the examples we'll see; this indicates that the semaphore is shared between threads in the same process.
> See the man page for details on other usages of semaphores (namely, how they can be used to synchronize access across different processes)
> If pshared is nonzero, then the semaphore is shared between processes, and should be located in a region of shared memory (see shm_open(3), mmap(2), and shmget(2)). (Since a child created by fork(2) inherits its parent's memory mappings, it can also access the semaphore.) Any process that can access the shared memory region can operate on the semaphore using sem_post(3), sem_wait(3), and so on

- definitions of sem_wait and sem_post

```
int sem_wait(sem_t *s) {
    decrement the value of semaphore s by one
    wait if value of semaphore s is negative
```

```
}

int sem_post(sem_t *s) {
    increment the value of semaphore s by one
    if there are one or more threads waiting, wake one
}
```

- sem wait() will either return right away (because the value of the semaphore was one or higher when we called sem wait()), or it will cause the caller to suspend execution waiting for a subsequent post. Of course, multiple calling threads may call into sem wait(), and thus all be queued waiting to be woken
- sem post() does not wait for some particular condition to hold like sem wait() does. Rather, it simply increments the value of the semaphore and then, if there is a thread waiting to be woken, wakes one of them up
- Third, the value of the semaphore, when negative, is equal to the number of waiting threads. Though the value generally isn't seen by users of the semaphores, this invariant is worth knowing and perhaps can help you remember how a semaphore functions

Binary Semaphores: Locks

- problem is to implement a lock sem_t m; sem_init(&m, 0, X); // initialize to X; what should X be?

```
sem_wait(&m);
// critical section here
sem_post(&m);
```

- For locks the value of X = 1

Semaphores for ordering

- problem is to ensure ordering of certain events, example in the below case child should complete its execution before parent sem_t s;

```
void *child(void *arg) {
    printf("child\n");
    sem_post(&s); // signal here: child is done
    return NULL;
}

int main(int argc, char *argv[]) {
    sem_init(&s, 0, X); // what should X be?
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL);
    sem_wait(&s); // wait here for child
    printf("parent: end\n");
```

```
    return 0;
}
```

- Initial value of X = 0, so that it can be made sure that child runs before parent

Producer/Consumer problem

- Producers generate data items and place them in a buffer; consumers grab said items from the buffer and consume them in some way
- The put() and get() routines

```
int buffer[MAX];
int fill = 0;
int use = 0;

void put(int value) {
    buffer[fill] = value; // Line F1
    fill = (fill + 1) % MAX; // Line F2
}

int get() {
    int tmp = buffer[use]; // Line G1
    use = (use + 1) % MAX; // Line G2
    return tmp;
}
```

- The first attempt:

```
sem_t empty;
sem_t full;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty); // Line P1
        put(i); // Line P2
        sem_post(&full); // Line P3
    }
}

void *consumer(void *arg) {
    int i, tmp = 0;
    while (tmp != -1) {
        sem_wait(&full); // Line C1
        tmp = get(); // Line C2
        sem_post(&empty); // Line C3
        printf("%d\n", tmp);
    }
```

```
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX are empty
    sem_init(&full, 0, 0); // 0 are full
    // ...
}
```

- works fine for MAX = 1
- if MAX is not 1, and two producer threads are there, see that in in put() routine, it may not update fill, and two threads may write to the same location

- Adding mutual exclusion(incorrectly):

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex); // Line P0 (NEW LINE)
        sem_wait(&empty); // Line P1
        put(i); // Line P2
        sem_post(&full); // Line P3
        sem_post(&mutex); // Line P4 (NEW LINE)
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&mutex); // Line C0 (NEW LINE)
        sem_wait(&full); // Line C1
        int tmp = get(); // Line C2
        sem_post(&empty); // Line C3
        sem_post(&mutex); // Line C4 (NEW LINE)
        printf("%d\n", tmp);
    }
}
```

- The problem here is that, the consumer may run before and thus have the mutex lock and may never let producer run, leading to a deadlock

- Adding it correctly:

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty); // Line P0
        sem_wait(&mutex); // Line P1 (NEW LINE)
        put(i); // Line P2
```

```
        sem_post(&mutex); // Line P3 (NEW LINE)
        sem_post(&full); // Line P4
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full); // Line C0
        sem_wait(&mutex); // Line C1 (NEW LINE)
        int tmp = get(); // Line C2
        sem_post(&mutex); // Line C3 (NEW LINE)
        sem_post(&empty); // Line C4
        printf("%d\n", tmp);
    }
}
```

Reader-Writer Locks

- admits that different data structure accesses might require different kinds of locking. For example, imagine a number of concurrent list operations, including inserts and simple lookups.
- lookups simply read the data structure; as long as we can guarantee that no insert is on-going, we can allow many lookups to pro⊴ceed concurrently
- Reader-Writer Lock implementation:

```
typedef struct _rwlock_t {
    sem_t lock; // binary semaphore (basic lock)
    sem_t writelock; // allow ONE writer/MANY readers
    int readers; // #readers in critical section
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
    rw->readers = 0;
    sem_init(&rw->lock, 0, 1);
    sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers++;
    if (rw->readers == 1) // first reader gets writelock
        sem_wait(&rw->writelock);
    sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
    sem_wait(&rw->lock);
    rw->readers--;
    if (rw->readers == 0) // last reader lets it go
```

```
            sem_post(&rw->writelock);
        sem_post(&rw->lock);
    }

    void rwlock_acquire_writelock(rwlock_t *rw) {
        sem_wait(&rw->writelock);
    }

    void rwlock_release_writelock(rwlock_t *rw) {
        sem_post(&rw->writelock);
    }
```

- More interesting is the pair of routines to acquire and release read locks. When acquiring a read lock, the reader first acquires lock and then increments the readers variable to track how many readers are currently inside the data structure. The important step then taken within rwlock acquire readlock() occurs when the first reader acquires the lock; in that case, the reader also acquires the write lock by calling sem wait() on the writelock semaphore, and then releasing the lock by calling sem post().
- Thus, once a reader has acquired a read lock, more readers will be allowed to acquire the read lock too; however, any thread that wishes to acquire the write lock will have to wait until all readers are finished; the last one to exit the critical section calls sem post() on “writelock” and thus enables a waiting writer to acquire the lock.
- This approach works (as desired), but does have some negatives, especially when it comes to fairness. In particular, it would be relatively easy for readers to starve writers.

> Think of how to manage the starve condition for writer

Dining Philosopher's problem

- Read it from the book, an easy one, simply break the cycle by using the forks for the fourth one in different order

```
    int left(int p) { return p; }
    int right(int p) { return (p + 1) % 5; }
    void get_forks(int p) {
        if (p == 4) {
            sem_wait(&forks[right(p)]);
            sem_wait(&forks[left(p)]);
        } else {
            sem_wait(&forks[left(p)]);
            sem_wait(&forks[right(p)]);
        }
    }
    void put_forks(int p) {
        sem_post(&forks[left(p)]);
        sem_post(&forks[right(p)]);
    }
```

Thread Throttling

- Assume there is a memory intensive region which requires a lot of memory to run
- You can figure out how many threads can simultaneously enter such a region and thus give this value to your semaphore, essentially limiting the number of threads that can access this piece of code
-

Implementing Semaphores

```
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

- Reader/Writer lock with no starvation

Lets develop no starvation solution with a requirement that no thread should be blocked to acquire the lock by any other thread which is ready to acquire the lock at later point in time. To develop no starvation locks we are following lock request order. Lets consider a lock request order as shown below: …. R1, R2, R3, R4, R5, W1 ….. R stands for read request and W stands for write request. In this scenario, the order in which reader threads: “R1, R2, R3, R4” acquires shared lock and get access to shared object has no impact on the shared object and also note that one reader thread can not starve another reader thread. The only restriction is all the reader threads: “R1, R2, R3, R4” must acquire shared lock (in any order) before the writer thread W1 attempts to acquires the exclusive lock. To avoid this before a writer thread puts lock request there should not be any reader thread waiting to acquire the shared lock.

Lets consider another lock request sequence as shown below: … W1, W2, W3, R1 … When writer thread W1 has acquired the exclusive lock, if we allow both writers W2 & W3 to proceed with the lock request, both these writers will compete with each other to get the exclusive lock (kernel thread scheduling will decide which one of them will be unblocked once W1 gives up the exclusive lock.) Similarly when W2 has acquired the exclusive lock and if we allow both W3 & R1 to proceed with the lock request, both W3 and R1 will compete for the shared object access. To avoid this situation till a writer thread doesn't give up the exclusive lock we must not allow other threads to put the lock request.

The strict ordering is based on two rules:

1. Before allowing a writer thread to compete for exclusive lock, one must make sure there are no readers waiting to acquire the shared lock.
2. When a writer has granted the exclusive lock no other lock request should be put till the writer thread gives up exclusive lock.

- Acquire other lock whenever a writer arrives which blocks the requests for user and put them to sleep