Lab 5: Synchronization

Sejong Yoon, Ph.D.

References:

- Silberschatz, et al. *Operating System Concepts* (10e), 2018
- Materials from OS courses offered at TCNJ (Dr. Jikai Li), Princeton, Rutgers, Columbia (Dr. Junfeng Yang), Stanford, MIT, UWisc, VT

TCNJ THE COLLEGE OF NEW JERSEY

Agenda

- Lock Implementation
- Mutex Example
- Semaphore Example
- Application

Avoid race conditions

- Critical section (CS): a segment of code that accesses a shared variable (or resource); no more than one thread in critical section at a time
- CS requirements
  - Safety (mutual exclusion)
  - Liveness (progress)
  - Bounded waiting (starvation-free)
- Makes no assumptions about speed & number of CPU
- CS desired properties
  - Efficient (no too much busy wait)
  - Fair
  - Simple

```
// ++ balance
mov    0x8049780,%eax
add    $0x1,%eax
mov    %eax,0x8049780
...

// -- balance
mov    0x8049780,%eax
sub    $0x1,%eax
mov    %eax,0x8049780
...
```

Implementing locks: version 1

- In uniprocessor case: disable / enable interrupts

```
lock()
{
        disable_interrupt();
}
```

```
unlock()
{
        enable_interrupt();
}
```

- Good: Simplicity
- Bad
  - Both operations are privileged; user programs cannot use them
  - Doesn’t work on multiprocessors

Implementing locks: version 2

- Peterson's algorithm: software-based lock implementation

- Good: doesn't require much from hardware
- Assumptions
  - Operations doing load and store are atomic
  - They are executed in order
  - Does not require special hardware instructions

Software-based lock: 1st attempt

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;

lock()
{
        while (flag == 1)
          ; // spin wait
        flag = 1;
}

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

Not atomic!

- Idea: use one flag, test, then set; if unavailable, spin-wait
- Problem
  - Not safe: both threads can be in critical section
  - Not efficient: busy wait, particularly bad on uniprocessor

Software-based lock

- 2nd attempt: use per-thread flags, set then test, to achieve mutual exclusion
  - Not live: can deadlock
- 3rd attempt: strict alternation to achieve mutual exclusion
  - Not live: depends on threads outside critical section
- Final attempt: combine above ideas

Implementing locks: version 3

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;
lock()                                    unlock()
{                                         {
      while(test_and_set(&flag))                flag = 0;
          ;                               }
}
```

- Problem with the test-set approach: test and set are not atomic
- Fix: special atomic operation

```
int test_and_set (int *lock) {
        int old = *lock;
        *lock = 1;
        return old;
}
```

  – Atomically returns *lock and sets *lock to 1

Implementing test_and_set on x86

```
long test_and_set(volatile long* lock)
{
    int old;
    asm("xchgl %0, %1"
        : "=r"(old), "+m"(*lock)  // output
        : "0"(1)                  // input
        : "memory"                // can clobber anything in memory
        );
    return old;
}
```

- xchg reg, addr: atomically swaps *addr and reg
- Most spin locks on x86 are implemented using this instruction
  - xv6: spinlock.h, spinlock.c, x86.h

xchg of x86

- x86 xchg %eax, addr instruction does the following:
  1. Freeze other CPUs' memory activity for address addr
  2. temp = *addr
  3. *addr = %eax
  4. %eax = temp
  5. Un-freeze other memory activity

Spin-wait or block?

- Problem: waste CPU cycles
  - Worst case: previous thread holding a busy-wait lock gets preempted, other thread try to acquire the same lock
- On uniprocessor: should not use spin-lock
  - Yield CPU when lock not available (need OS support)
- On multi-processor
  - Correct action depends on how long it would take before lock release
    - Lock released “quickly” → ?
    - Lock released “slowly” → ?

Problem with simple yield

```
lock()
{
        while(test_and_set(&flag))
            yield();
}
```

- Problem
  - Still a lot of context switches: thundering herd
  - Starvation possible
- Why? No control over who gets the lock next
- Need explicit control over who gets the lock

Implementing locks: version 4

```
lock() {
  while (test_and_set(&flag)))
     add myself to wait queue
     yield
  ...
}
```

```
unlock() {
   flag = 0
   if(any thread in wait queue)
      wake up one wait thread
  ...
}
```

- Idea: add thread to queue when lock unavailable; in unlock(), wake up one thread in queue

Implementing locks: version 4 (code)

```
typedef struct __mutex_t {
    int flag;       // 0: mutex is available,  1: mutex is not available
    int guard;      // guard lock to avoid losing wakeups
    queue_t *q;  // queue of waiting threads
} mutex_t;
```

```
void lock(mutex_t *m) {
    while (test_and_set(m->guard))
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // acquire mutex
        m->guard = 0;
    } else {
        enqueue(m->q, self);
        m->guard = 0;
        yield();
    }
}
```

```
void unlock(mutex_t *m) {
    while (test_and_set(m->guard))
        ;
    if (queue_empty(m->q))
        // release mutex; no one wants mutex
         m->flag = 0;
    else
        // direct transfer mutex to next thread
         wakeup(dequeue(m->q));
    m->guard = 0;
}
```

TCNJ THE COLLEGE OF NEW JERSEY

Exercise 1: Mutex Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define BUF_SIZE 10

int buf[BUF_SIZE];

pthread_mutex_t lock;

void *prod(void *param)
{
	int i;

	/* pthread_mutex_lock(&lock); /* 1 */
	for (i = 0; i < BUF_SIZE; ++i) {
		buf[i] = i;
	}
	/* pthread_mutex_unlock(&lock); /* 1 */

	pthread_exit(0);
}
```

Exercise 1: Mutex Example (cont.)

```
void *cons(void *param)
{
      int i;

      /* pthread_mutex_lock(&lock); /* 1 */
      for (i = 0; i < BUF_SIZE; ++i) {
            printf("Buffer index %d = %d\n", i, buf[i]);
      }
      /* pthread_mutex_unlock(&lock); /* 1 */

      pthread_exit(0);
}
```

Exercise 1: Mutex Example (cont.)

```
int main(int argc, char** argv)
{
      pthread_t t_prod, t_cons;

      /* pthread_mutex_init(&lock, NULL); /* 1 */

      pthread_create(&t_prod, 0, prod, NULL);
      /* sleep(0.5); /* 2 */
      pthread_create(&t_cons, 0, cons, NULL);

      pthread_join(t_prod, NULL);
      pthread_join(t_cons, NULL);

      /* pthread_mutex_destroy(&lock); /* 1 */

      return 0;
}
```

Exercise 1: Mutex Example (cont.)

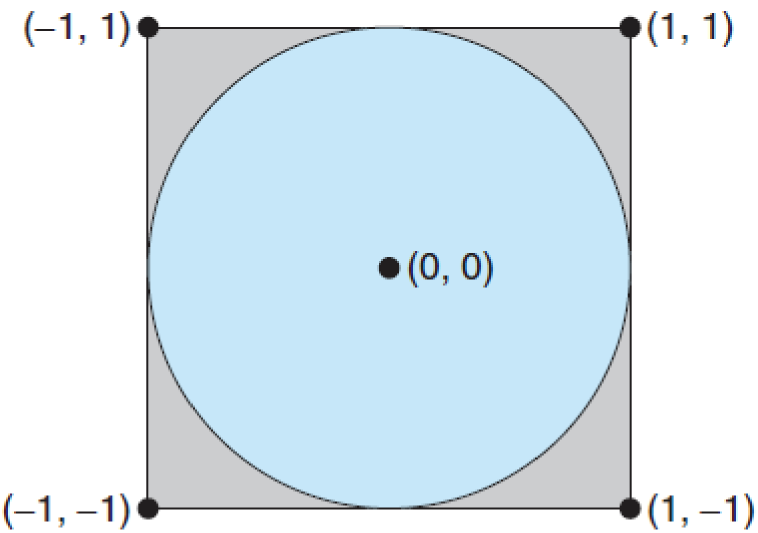
- You need to compile this code with -lpthread option. Run the program.
  - What do you observe?
- Uncomment the lines marked as /* 1 */ and compile again.
  - What do you expect to see?
  - What do you actually get?
  - Compare your result with your friends'. Why do you think you get such results?
- Uncomment the line marked as /* 2 */ and compile again.
  - What do you expect to see?
  - What do you actually get?
  - Why do you think you get such result?
  - Can you provide a solution to this problem?

Exercise 2: Semaphore Example

```
#include <unistd.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>

int main(int argc, char** argv)
{
      sem_t sem;

      if (sem_init(&sem,0,1) == -1)
            printf("%s\n",strerror(errno));

      if (sem_wait(&sem) != 0)
            printf("%s\n",strerror(errno));

     printf("*** Critical Section ***\n");

      if (sem_post(&sem) != 0)
            printf("%s\n",strerror(errno));

     printf("*** Non-Critical Section ***\n");

      if (sem_destroy(&sem) != 0)
            printf("%s\n",strerror(errno));

      return 0;
}
```

Review: Exercise 4.22

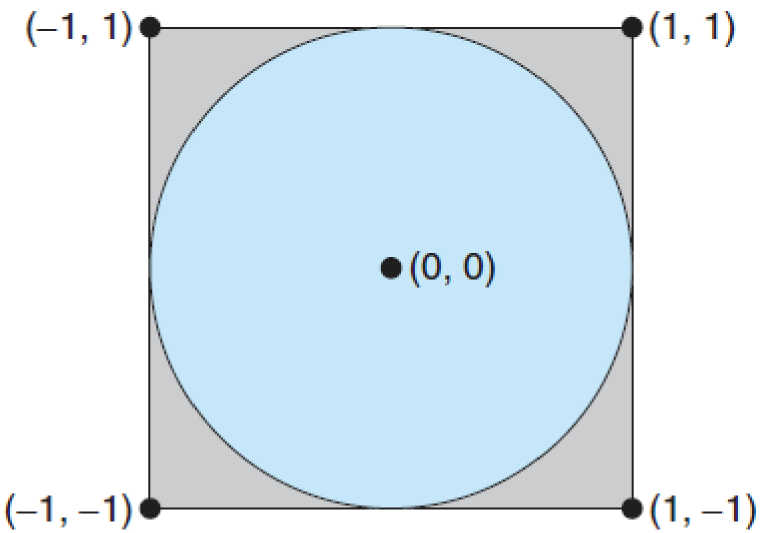
- An interesting way of calculating pi is to use a technique known as Monte Carlo, which involves randomization. This technique works as follows: Suppose you have a circle inscribed within a square, as shown in figure below: (Assume that the radius of this circle is 1.)



Review: Exercise 4.22 (cont.)

- First, generate a series of random points as simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square. Of the total number of random points that are generated, some will occur within the circle. Next, estimate pi by performing the following calculation:

  pi = 4× (number of points in circle) / (total number of points)



Exercise 3: Exercise 5.39

- Exercise 4.22 asked you to design a multithreaded program that estimated pi using the Monte Carlo technique. In that exercise, you were asked to create a single thread that generated random points, storing the result in a global variable. Once that thread exited, the parent thread performed the calculation that estimated the value of pi. Modify that program so that you create several threads, each of which generates random points and determines if the points fall within the circle. Your program need to satisfy following two conditions.
  - (a) Each thread will have to update the global count of all points that fall within the circle.
  - (b) Protect against race conditions on updates to the shared global variable by some synchronization method.

Exercise 3: Exercise 5.39 (cont.)

- Your program should get total number of random trials from the command line
- Each thread will run 1/N of the total number of random trials above, where N is the number of threads you will create; make sure there is no remainders and you actually generated the given number of random points
- Your program will NOT get number of threads from the command line
  - You may decide how many processes to create; 4 is a good choice

Lab 5 assignment

- Add a comment to the beginning of your source code containing your name, the name of the course, and the title of the assignment:

  /** John Smith

  CSC345-01 (or CSC345-02)

  Lab 5 Exercise 1 */
- Rename your source file into lab05_ex1.c, lab05_ex2.c, lab05_ex3.c
- Prepare Makefile that compiles your source codes into object code lab05_ex1, lab05_ex2, lab05_ex3
- Zip your source file into lab05.zip
- Submit your **zip** file via Canvas