08. Synchronization II

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Mutex Locks

- Previous solutions are generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
  - The simplest is **mutex lock**
    - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **acquire()** a lock
  - Then **release()** the lock
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap

Solution to CS Problem Using Mutex Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Deadlocks with Mutexes

- Consider the following scenario

Thread A	**Thread B**
1. *pthread_mutex_lock(mutex1);*	1. *pthread_mutex_lock(mutex2);*
2. *pthread_mutex_lock(mutex2);* blocks	2. *pthread_mutex_lock(mutex1);* blocks

- A deadlock may occur!
- Lesson: threads should always acquire locks in the same order

Pthread Mutexes

- Data type:
  - pthread_mutex_t
- Common operations:
  - int pthread_mutex_lock(pthread_mutex_t *mutex);
  - int pthread_mutex_unlock(pthread_mutex_t *mutex);
- Typical use:

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *thread_func(void *arg)
{
  pthread_mutex_lock(&mtx); // get the lock
  // access shared data
  pthread_mutex_unlock(&mtx); // release the lock
}
```

Locking the Same Lock?

- What if the same thread tries to obtain the same mutex multiple times?
  - The result depends on how the mutex was initialized
- Types of pthread mutexes:
  - PTHREAD_MUTEX_DEFAULT or PTHREAD_MUTEX_NORMAL
    - Results in a deadlock if the same pthread tries to lock it a second time using the pthread_mutex_lock subroutine without first unlocking it. This is the default type
  - PTHREAD_MUTEX_ERRORCHECK
    - Avoids deadlocks by returning a non-zero value if the same thread attempts to lock the same mutex more than once without first unlocking the mutex
  - PTHREAD_MUTEX_RECURSIVE
  - Allows the same pthread to recursively lock the mutex using the pthread_mutex_lock subroutine without resulting in a deadlock or getting a non-zero return value from pthread_mutex_lock. The same pthread has to call the pthread_mutex_unlock subroutine the same number of times as it called pthread_mutex_lock subroutine in order to unlock the mutex for other pthreads to use

Producer-Consumer Problem

- A canonical example: the producer-consumer problem
  - Producer threads produce elements
  - Consumer threads consume the elements produced by the producer threads
- A lock alone isn’t a good solution:
  - It only ensures mutual exclusion
  - Consider the case where a consumer wants to run but there are no elements available:
    - Obtain lock
    - Check for elements
    - Release lock
    - Sleep
- Condition variables to the rescue!

Condition Variables

- A condition variable allows one thread to inform other threads about changes in the state of a shared variable (or other shared resource) and allows the other threads to wait (block) for such notification
- Common operations:
  - int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
  - int pthread_cond_signal(pthread_cond_t *cond);
  - int pthread_cond_broadcast(pthread_cond_t *cond);

Solution to Producer-Consumer Problem

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t cond_empty = PTHREAD_COND_INITIALIZER;

void *producer_func(void *arg)
{
  pthread_mutex_lock(&mtx);
  while (num_avail >= MAX_SIZE)
    pthread_cond_wait(&cond_empty, &mtx);
  num_avail++;
  pthread_mutex_unlock(&mtx);
  pthread_cond_signal(&cond_full);
}
```

```
void *consumer_func(void *arg)
{
  pthread_mutex_lock(&mtx);
  while (num_avail <= 0)
    pthread_cond_wait(&cond_full, &mtx);
  // consumer data and process
  num_avail--;
  pthread_mutex_unlock(&mtx);
  pthread_cond_signal(&cond_empty);
}
```

Mutex + Condition Variable

- The mutex associated with a condition variable is for mutual exclusion
- The condition variable is for signaling
- Important: always check the condition in a while loop!

```
void *consumer_func(void *arg)
{
  pthread_mutex_lock(&mtx);
  while (num_avail <= 0)
    pthread_cond_wait(&cond_full, &mtx);
  // consumer data and process
  num_avail--;
  pthread_mutex_unlock(&mtx);
  pthread_cond_signal(&cond_empty);
}
```

This atomically:
1. Unlocks the mutex
2. Waits on the condition variable

When execution reaches here, you have obtained the mutex, so you must unlock it

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities
- Semaphore S – integer variable
  - Can only be accessed via two indivisible (atomic) operations
    - wait() and signal()
    - Originally called P() and V()
      - Dutch: Probeer (try) and Verhoog (increment) in Dijkstra's original paper
  - Counting semaphore – integer value can range over an unrestricted domain
  - Binary semaphore – integer value can range only between 0 and 1
    - Same as a mutex lock

Semaphore Implementation

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

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

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

Semaphore Usage Example

- With semaphores we can solve various synchronization problems
- Solution to the critical section problem
  - Create a semaphore “mutex” initialized to 1

```
wait(mutex);
   CS
signal(mutex);
```

- Consider P1 and P2 that with two statements S1 and S2 and the requirement that S1 to happen before S2
  - Create a semaphore “synch” initialized to 0

```
P1:                         P2:
    S1;                         wait(synch);
    signal(synch);              S2;
```

POSIX Semaphores (Unnamed)

- Data type:
  - sem_t   (not <pthread.h> but <semaphore.h>)
- Common operations:
  - int sem_init(sem_t *sem, int pshared, unsigned int value);
  - int sem_wait(sem_t *sem);
  - int sem_post(sem_t *sem);
  - int sem_getvalue(sem_t *sem, int *valp);
  - int sem_destroy(sem_t *sem);

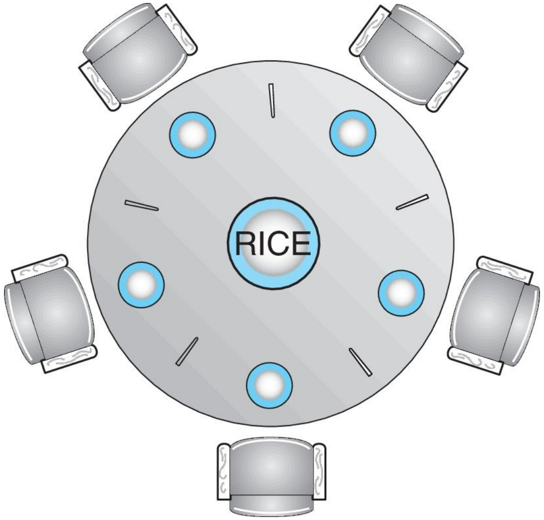
Semaphore Example

```c
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>

sem_t mutex;

void *
foo(void *data)
{
  sem_wait(&mutex);

  //critical section

  sem_post(&mutex);
}
```

```c
int
main()
{
  pthread_t thd;

  sem_init(&mutex, 0, 1);

  pthread_create(&thd, NULL, foo, NULL);
  pthread_join(thd, NULL);

  sem_destroy(&mutex);
  return 0;
}
```

Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowl of rice in the middle
  - They spend their lives alternating thinking and eating
  - They do not interact with their neighbors
  - Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
    - Need both to eat, then release both when done
  - In the case of 5 philosophers, the shared data
    - Bowl of rice (data set)
    - Semaphore chopstick[5] initialized to 1



What Can Go Wrong?

- What is the problem with the following solution?

```
while (true){
  wait(chopstick[i]);
  wait(chopstick[(i + 1) % 5]);

  /* eat for a while */

  signal(chopstick[i]);
  signal(chopstick[(i + 1) % 5]);

  /* think for a while */

}
```

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - Atomic integers
  - Spinlocks
    - Busy waiting!
- On single-CPU system, spinlocks replaced by enabling and disabling kernel preemption
  - Why?

Atomic Variables

- atomic_t is the type for atomic integer
    - atomic_t counter;

Atomic Operation	*Effect*
`atomic_set(&counter,5);`	`counter = 5`
`atomic_add(10,&counter);`	`counter = counter + 10`
`atomic_sub(4,&counter);`	`counter = counter - 4`
`atomic_inc(&counter);`	`counter = counter + 1`
`value = atomic_read(&counter);`	`value = 12`

Linux Implementation of Mutexes

- Locking:
  - Bit 31: indicates if lock is taken
    - 1: taken
    - 0 free
  - Remaining bits: number of waiters
  - Line 7: don't return from here until we get the lock
  - Lines 8-10: check lock is free and decrement # of waiters if so
    - Decrement because we incremented
  - Line 15: check if lock is taken
  - Line 17: futex system call
    - Put calling process on queue

```
1  void mutex_lock (int *mutex) {
2    int v;
3    /* Bit 31 was clear, we got the mutex (the fastpath) */
4    if (atomic_bit_test_set (mutex, 31) == 0)
5      return;
6    atomic_increment (mutex);
7    while (1) {
8        if (atomic_bit_test_set (mutex, 31) == 0) {
9            atomic_decrement (mutex);
10           return;
11       }
12       /* We have to waitFirst make sure the futex value
13          we are monitoring is truly negative (locked). */
14       v = *mutex;
15       if (v >= 0)
16         continue;
17       futex_wait (mutex, v);
18   }
19 }
20
21 void mutex_unlock (int *mutex) {
22   /* Adding 0x80000000 to counter results in 0 if and
23      only if there are not other interested threads */
24   if (atomic_add_zero (mutex, 0x80000000))
25     return;
26
27   /* There are other threads waiting for this mutex,
28      wake one of them up.  */
29   futex_wake (mutex);
30 }
```

Credit: Operating Systems: Three Easy Pieces by Arpaci-Dusseau and Arpaci-Dusseau

Linux Implementation of Mutexes

- Unlocking:
    - Lines 24-25: if *mutex == 0 after adding 0x80000000, then nobody was waiting
    - Line 29: invoke the futex system call
        - Argument of FUTEX_WAKE
- futex() is a multiplexed system call
    - Different arguments change the behavior
    - FUTEX_WAIT: go to sleep until mutex is available
    - FUTEX_WAKE: wake up someone waiting on this mutex
- Described in paper “Futexes are Tricky” by Ulrich Drepper

```
1  void mutex_lock (int *mutex) {
2    int v;
3    /* Bit 31 was clear, we got the mutex (the fastpath) */
4    if (atomic_bit_test_set (mutex, 31) == 0)
5      return;
6    atomic_increment (mutex);
7    while (1) {
8        if (atomic_bit_test_set (mutex, 31) == 0) {
9            atomic_decrement (mutex);
10           return;
11       }
12       /* We have to waitFirst make sure the futex value
13          we are monitoring is truly negative (locked). */
14       v = *mutex;
15       if (v >= 0)
16         continue;
17       futex_wait (mutex, v);
18   }
19 }
20
21 void mutex_unlock (int *mutex) {
22   /* Adding 0x80000000 to counter results in 0 if and
23      only if there are not other interested threads */
24   if (atomic_add_zero (mutex, 0x80000000))
25     return;
26
27   /* There are other threads waiting for this mutex,
28      wake one of them up. */
29   futex_wake (mutex);
30 }
```

Credit: Operating Systems: Three Easy Pieces by Arpaci-Dusseau and Arpaci-Dusseau

Homework

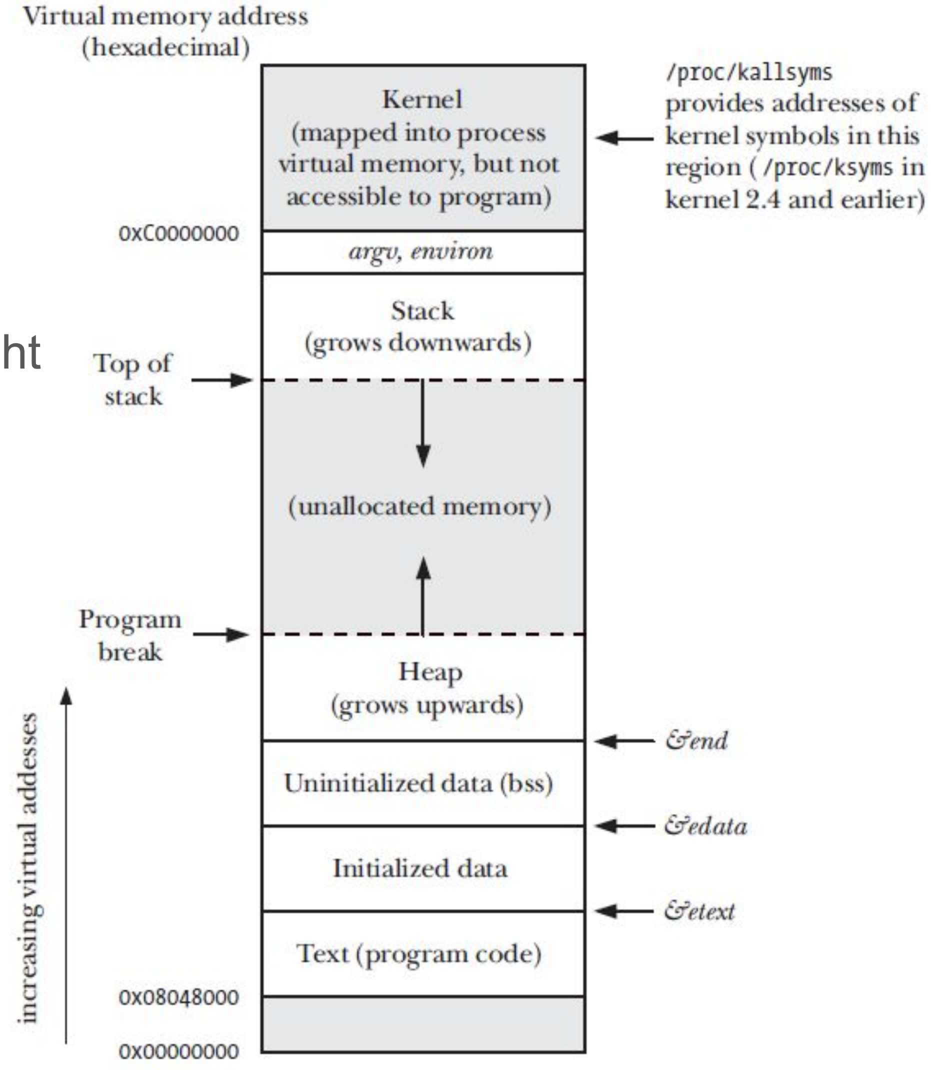
- Start reading Chapters 9 & 10
- Assignment 3 deadline is changed to 10/09

Next Lecture

Let’s start looking at memory management!

Miniproject-3 Q&A

- Typical address space layout of a user-level process is shown on the right
- These addresses are all virtual
  - It’s the job of the hardware (MMU) + OS to translate these to physical addresse



Check Your Answer

- /proc/<pid>/maps
    - It gives each virtual memory address region in a process
        - Address range
        - Permission
        - Offset in a file
            - Where file locate (device, inode, path)
            - You can use objdump -D to find all sections
            - Anonymous ones are all 0 in the following fields (no backing files)