Operating Systems

CSE 231

Instructor: Sambuddho Chakravarty

(Semester: Monsoon 2020)

Week 6: Oct 26 – Oct 29

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution

- Concurrent access to shared data may result in data inconsistency

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

- Illustration of the problem:
  Suppose that we wanted to provide a solution to the consumer-producer problem that fills ***all*** the buffers. We can do so by having an integer **`counter`** that keeps track of the number of full buffers. Initially, **`counter`** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
   /* produce an item in next produced */

   while (counter == BUFFER_SIZE) ;
      /* do nothing */
   buffer[in] = next_produced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer

```
while (true) {
   while (counter == 0)
      ; /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;  counter--;
   /* consume the item in next consumed */
}
```

Race Condition

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute **`register1 = counter`** {register1 = 5}
S1: producer execute **`register1 = register1 + 1`** {register1 = 6}
S2: consumer execute **`register2 = counter`** {register2 = 5}
S3: consumer execute **`register2 = register2 – 1`** {register2 = 4}
S4: producer execute **`counter = register1`** {counter = 6 }
S5: consumer execute **`counter = register2`** {counter = 4}

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}

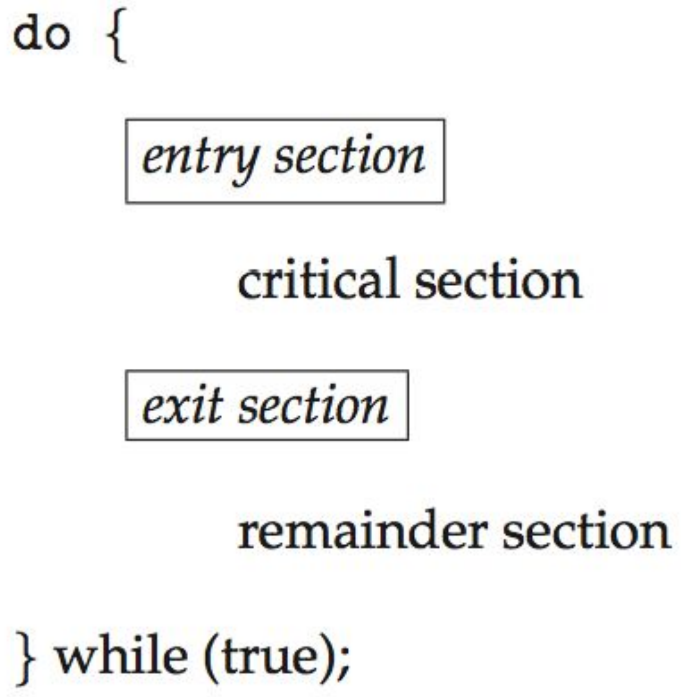
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section

- ***Critical section problem*** is to design protocol to solve this

- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process $\boldsymbol{p_i}$ is



```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```

Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process ***P_i*** is executing in its critical section, then no other processes can be executing in their critical sections

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Solution to Critical-Section Problem

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when doing kernel operations (e.g. execution of system call).
  - Complicated in presence SMP architectures.
- **Non-preemptive** – runs until exits kernel operstion, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Peterson’s Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Algorithm for Process Pi

```
do {
   flag[i] = true;
   turn = j;
   while (flag[j] && turn == j);
      critical section
   flag[i] = false;
      remainder section
} while (true);
```

- Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for critical section code

- All solutions below based on idea of **locking**
    - Protecting critical regions via locks

- Uniprocessors – could disable interrupts
    - Currently running code would execute without preemption
    - Generally too inefficient on multiprocessor systems
        - Operating systems using this not broadly scalable

- Modern machines provide special atomic hardware instructions
        - **Atomic** = non-interruptible
    - Either test memory word and set value
    - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (TRUE);
```

test_and_set Instruction

- Definition:

```
boolean test_and_set (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
 }
```

Solution using test_and_set()

-Shared boolean variable lock, initialized to FALSE

-Solution:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
   /* critical section */
   lock = false;
   /* remainder section */
} while (true);
```

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;                    /* Meets condition 2 and 3 – progress and bounded waiting
*/
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = false;
   else
      waiting[j] = false;
   /* remainder section */
} while (true);
```

Atomic Operations

- Operations that are completed in such a way that that they are not pre-empted or they do not going into a inconsistent state.
- Most operation which solve the critical section problem do so via ``locks’’.
- One may say that it is a clever way to have atomic ``locks’’ .

Atomic Operations inside the Linux Kernel

- atomic_t defined in <sys/types.h>

```
typedef struct {
volatile int counter;
} atomic_t;

atomic_t v; /* define v */
atomic_t u = ATOMIC_INIT(0); /* define u and initialize it to zero */
atomic_set(&v, 4); /* v = 4 (atomically) */
atomic_add(2, &v); /* v = v + 2 = 6 (atomically) */
atomic_inc(&v); /* v = v + 1 = 7 (atomically) */
```

Spin Locks

- Lock that is held by *atmost* one thread of execution and continuously checks if the lock is being used or not – aka **busy wait.**
- Best to be held and used for only a short duration of time.

```
<linux/spinlock.h>

DEFINE_SPINLOCK(mr_lock);
spin_lock(&mr_lock);
/* critical section ... */
spin_unlock(&mr_lock);
```

Spin Locks

- Can be used in interrupt handlers, unlike other primitives that cannot be used because they may be pre-empted.
- Always disable local interrupts (interrupt requests on the current processor) before obtaining the lock.
- Otherwise, it is possible for an interrupt handler to interrupt kernel code while the lock is held and attempt to reacquire the lock. The interrupt handler spins, waiting for the lock to become available. **The lock holder, however, does not run until the interrupt handler completes!**

  (deadlock scenario)

Reader Writer Locks

- Some process may only want to read a shared resource without any other process simultaneously modifying it.
- Reader/writer spin lock.
- **“**When a data structure is neatly split into reader/writer or consumer/producer usage patterns, it makes sense to use a locking mechanism that provides similar semantics.”

Reader Writer Locks

Warning: One cannot have a read lock being ``upgraded'' to a write lock.

```
DEFINE_RWLOCK(mr_rwlock);
```

The reader code path:

```
read_lock(&mr_rwlock);
/* critical section (read only) ... */
read_unlock(&mr_rwlock);
```

The writer code path:

```
write_lock(&mr_rwlock);
/* critical section (read and write) ... */
write_unlock(&mr_lock);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore ***S*** – integer variable
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Originally called **`P()`** and **`V()`**
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
     while (S <= 0)
         ; // busy wait
     S--;
}
signal (S) {
     S++;
}
```

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Then a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

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

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time

- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section
  - 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

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue 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 Implementation with no Busy waiting (Cont.)

```
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);
   }
}
```

Deadlock and 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

```
          P0                      P1
                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
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
       ...
   /* produce an item in next_produced */
       ...
   wait(empty);
   wait(mutex);
       ...
   /* add next produced to the buffer */
       ...
   signal(mutex);
   signal(full);
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
do {
        ...
   /* produce an item in next_produced */
        ...
   wait(empty);
   wait(mutex);
        ...
   /* add next produced to the buffer */
        ...
   signal(mutex);
   signal(full);
} while (true);
```

```
do {
   wait(full);
   wait(mutex);
        ...
   /* remove an item from buffer
to next_consumed */
        ...
   signal(mutex);
   signal(empty);
        ...
   /* consume the item in next
consumed */
        ...
} while (true);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
   wait(full);
   wait(mutex);
      ...
   /* remove an item from buffer to next_consumed */
      ...
   signal(mutex);
   signal(empty);
      ...
   /* consume the item in next consumed */
      ...
} while (true);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do ***not*** perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
   wait(rw_mutex);

      ...
   /* writing is performed */

      ...

   signal(rw_mutex);

} while (true);
```

Readers-Writers Problem (Cont.)

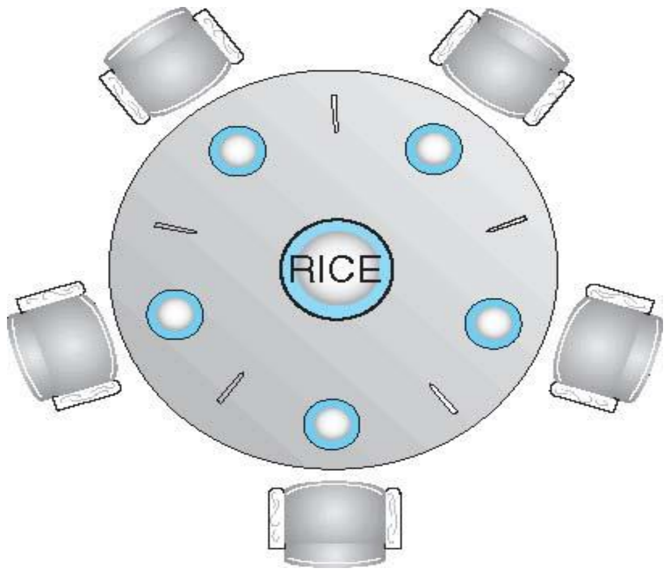
- The structure of a reader process

```
do {
   wait(mutex);
   read count++;
   if (read_count == 1)

      wait(rw_mutex);

   signal(mutex);

   ...
   /* reading is performed */

   ...

   wait(mutex);
   read count--;
   if (read_count == 0)

      signal(rw_mutex);

   signal(mutex);

} while (true);
```

Readers-Writers Problem Variations

- ***First*** variation – no reader kept waiting unless writer has permission to use shared object
- ***Second*** variation – once writer is ready, it performs write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem



- Philosophers spend their lives thinking and eating
- Don’t 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
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do  {
        wait (chopstick[i] );
        wait (chopStick[ (i + 1) % 5] );

                //  eat

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

                //  think

} while (TRUE);
```

- What is the problem with this algorithm?

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

Pthreads Synchronization

- Pthreads API is OS-independent

- It provides:
  - mutex locks
  - condition variables

- Non-portable extensions include:
  - read-write locks
  - spinlocks

Pthreads Synchronization

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

pthread_t tid[2];
int counter;
pthread_mutex_t lock;
void* doSomeThing(void *arg) {
 pthread_mutex_lock(&lock);
 unsigned long i = 0;
counter += 1;
printf("\n Job %d started\n", counter);

for(i=0; i<(0xFFFFFFFF);i++);
printf("\n Job %d finished\n", counter);
pthread_mutex_unlock(&lock); return NULL; }
```

Pthreads Synchronization

```
int main(void)
{
int i = 0;
int err;
if (pthread_mutex_init(&lock, NULL) != 0)
{ printf("\n mutex init failed\n"); return 1; }

while(i < 2)
{ err = pthread_create(&(tid[i]), NULL, &doSomeThing, NULL);
 if (err != 0)
    printf("\ncan't create thread :[%s]", strerror(err));
 i++; }

pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
pthread_mutex_destroy(&lock);
return 0; }
```

POSIX Semaphores

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,  unsigned int
value);
```

"If *pshared* has the value 0, then the semaphore is shared between the threads of a process, and should be located at some address that is visible to all threads (e.g., a global variable, or a variable allocated dynamically on the heap)."

"If *pshared* >0 , 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), etc"

POSIX Semaphores

`int sem_wait (sem_t *sem);`

- Decrements the value of the semaphore. If the value is < 0 then the current process blocks.

Same as semaphore wait()/down() operation.

`int sem_post(sem_t *sem);`

`-Increments the value of the semaphore and wakes up a process that is blocked due to the current semaphore that is being referenced.`

Same as semaphore signal()/up() operation.

int sem_getvalue(sem_t *sem, int *valp);

-Obtains the current value of the semaphore and saves it into the memory location pointed to by valp.

int sem_destroy(sem_t *sem);

-Destroys the semaphore; no thread should be waiting on this semaphore if the destruction is to succeed.

POSIX Semaphores

```
sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```