Operating System Design

Processes Synchronization

Neda Nasiriani
Fall 2018

Process Synchronization

Get Help from Hardware for Locks

- What was the problem here?!?

```
do {
        while (lock);
        lock = 1;
                critical section
        lock = 0;
                remainder section
 } while (true);
```

How to have a working lock?

- Can this be fixed if we were able to **test and set** the value of lock in one atomic (uninterruptible) operation?

```
do {
        while (lock);
        lock = 1;
                critical section
        lock = 0;
                remainder section
 } while (true);
```

No interrupts

Test and Set Instruction

- There is hardware support for such instructions
- The whole instruction will be executed as one uninterruptible unit of operation
- One example of such instructions: Test_and_Set

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Atomically

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.

Lock using Test and Set

- lock initialized to false
- Let's use the test and set operation for implementing our lock!

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Mutual Exclusion: Pass

Bounded Waiting: ?!?

Compare and Swap Instruction

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.

Lock using Compare and Swap

- lock initialized to 0

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

Mutual Exclusion: Pass

Bounded Waiting: ?!?

Is this a Valid CS solution? Groups of 3

- Two shared variables
  - boolean waiting[n] = false
  - boolean lock = false
  - Note: key is local variable
- Does this solution satisfy
  - Mutual Exclusion
  - Progress
  - Bounded Waiting

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = false;
   else
      waiting[j] = false;

      /* remainder section */
} while (true);
```

Mutex (**Mut**ual **Ex**clusion) Locks

- Solutions seen so far are complicated!
- So Operating Systems designers build software tools to solve CS problem
- A process should **acquire** the lock in the entry section then is allowed to enter its CS
- After the process is done, it should **release** its lock in the exit section

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

Mutex Lock

- The function acquire is a blocking operation
- Called also **spinlock**

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;;
}
```

```
release() {
   available = true;
}
```

Atomically

Counting Locks

- What if
  - we have **more than one copy** of the resource?
  - Or want to allow up to *n* processes into the critical section?
- We need a counting lock…

Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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()`**

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

```
signal(S) {
    S++;
}
```

Semaphores Continued

- **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**
- Can solve various synchronization problems
- Consider $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ that require $\boldsymbol{S_1}$ to happen before $\boldsymbol{S_2}$

Create a semaphore "**synch**" initialized to 0

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

Can we avoid busy waiting?!?

```
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;;
}
```

```
release() {
    availab... true;
}
```

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

```
signal(S) {
    S++;
}
```

```
 {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   waiting[i] = false;

      /* critical section */

     = (i + 1) % n;
     ile ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)

               alse;

               ection */
```

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

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

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

- 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

No Busy Waiting!

- 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

```
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);
          }
}
```

Back to Producer Consumer Problem ☺

Producer-Consumer: Semaphores

Does using a lock on counter resolve the issue?

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

      while (counter == BUFFER_SIZE)
         ; /* do nothing */

      buffer[in] = next_produced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

```
while (true) {
      while (counter == 0)
         ; /* do nothing */

      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;

      /* consume the item in next_consumed */
}
```

Producer-Consumer: Semaphores

Candid for semaphores!

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

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

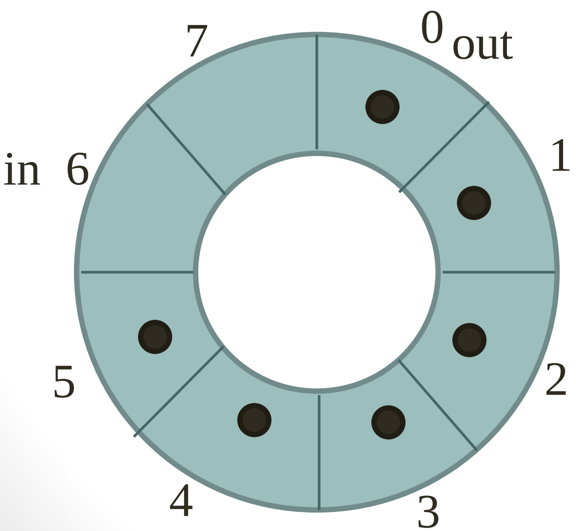
```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

Producer-Consumer with Semaphores!

- Here we have BUFFER_SIZE number of resources (empty slots) and we want the producer to wait for an empty slot while the consumer waits for a full slot
  - We can use a counting semaphore for empty slots
  - We also need to check if there is any full slots in the buffer



Producer-Consumer with Semaphores!

```
sem_t empty, full;

sem_init(&full, 0, 0);
sem_init(&empty, 0, BUFFER_SIZE);
```

Does this satisfy mutual exclusion?

```
while(true){
    /* produce an item */
    sem_wait(&empty);

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;

    sem_post(&full);
}
```

```
while(true){
    /* consume an item */
    sem_wait(&full);

    next_consumed = buffer[out];
    in = (in + 1)% BUFFER_SIZE;

    sem_post(&empty);
}
```

Producer-Consumer with Semaphores!

```
sem_t empty, full;

sem_init(&full, 0, 0);
sem_init(&empty, 0, BUFFER_SIZE);
```

Now lets assume 2 processes are producing items and putting it into buffer concurrently!!!
On board!

```
while(true){
    /* produce an item */
    sem_wait(&empty);

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;

    sem_post(&full);
}
```

```
while(true){
    /* consume an item */
    sem_wait(&full);

    next_consumed = buffer[out];
    in = (in + 1)% BUFFER_SIZE;

    sem_post(&empty);
}
```

Producer-Consumer with Semaphores!

```
sem_t empty, full, mutex;

sem_init(&full, 0, 0);
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&mutex, 0, 1);
```

How can we fix this?

Does this work?

```
while(true){
    /* produce an item */
    sem_wait(&mutex);
    sem_wait(&empty);

    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;

    sem_post(&full);
    sem_post(&mutex);
}
```

```
while(true){
    /* consume an item */
    sem_wait(&mutex);
    sem_wait(&full);

    next_consumed = buffer[out];
    in = (in + 1)% BUFFER_SIZE;

    sem_post(&empty);
    sem_post(&mutex)
}
```

Producer-Consumer with Semaphores!

```
sem_t empty, full, mutex;

sem_init(&full, 0, 0);
sem_init(&empty, 0, BUFFER_SIZE);
sem_init(&mutex, 0, 1);
```

Working Version!

```
while(true){
    /* produce an item */
    sem_wait(&empty);

    sem_wait(&mutex);
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    sem_post(&mutex);

    sem_post(&full);

}
```

```
while(true){
    /* consume an item */
    sem_wait(&full);

    sem_wait(&mutex);
    next_consumed = buffer[out];
    in = (in + 1)% BUFFER_SIZE;
    sem_post(&mutex);

    sem_post(&empty);

}
```

Producer-Consumer with Semaphores! SGG Book version!

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

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

```
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);
```