PROCESS SYNCHRONIZATION

Background

- A cooperating process is one that can affect or be affected be other processes executing in the system
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Consider our original solution to producer-consumer problem

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

```
item next_consumed;
while(true) {
   while(in == out)
      ;   /* do nothing */
   next_consumed = buffer[out];
   out = (out + 1)% BUFFER_SIZE;
}
```

Background

- 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**
  - Initially, counter is set to 0.
  - It is incremented by the producer after it produces a new buffer
  - And it 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 */

}
```

- counter++ could be implemented as

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

- counter++ could be implemented as

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

- counter-- could be implemented as

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

Race Condition

- Consider this execution interleaving with "counter = 5" initially:

T0: producer execute $register_1$ = counter	{$register_1$ = 5}
T1: producer execute $register_1 = register_1 + 1$	{$register_1$ = 6}
T2: consumer execute $register_2$ = counter	{$register_2$ = 5}
T3: consumer execute $register_2 = register_2 - 1$	{$register_2$ = 4}
T4: producer execute counter = $register_1$	{counter = 6 }
T5: consumer execute counter = $register_2$	{counter = 4}

- This happened because we allowed both processes to manipulate the variable counter concurrently
- A situation like this is called a **race condition**
- Hence we need, **Process Synchronization** and **Coordination** among cooperating processes

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

Critical-Section Problem

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

General structure of process P_i

```
do {

     entry section

          critical section

     exit section

          remainder section

} while (true);
```

Solution to Critical-Section Problem

- A solution to the critical section code must satisfy the following three requirements:

1. **Mutual Exclusion**
2. **Progress**
3. **Bounded Waiting**

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

Solution to Critical-Section Problem

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

Solution to Critical-Section Problem

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

Critical-Section Handling in OS

- Two approaches are used to handle critical sections in Operating Systems:
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

Peterson’s Solution

- Two process solution
- We denote the two process as P_i and P_j

Peterson's Solution

- 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!
  - Initially `flag[i] = flag[j] = false`

Algorithm for Process P_i

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

 flag[i] = false;
    remainder section

 } while(true);
```

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

P_i

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

P_j

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
- 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-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

- All these solutions protect critical section by **locking**

```
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:
    }
```

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

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);
```

compare_and_swap Instruction

Definition:

```
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.

Solution using compare_and_swap

- Shared integer "**lock**" initialized to **0**;
- Solution:

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
         ;  /* do nothing */
     /* critical section */
       lock = 0;
     /* remainder section */
} while (true);
```

- These algorithms satisfy the mutual-exclusion requirement, but they do not satisfy bounded-waiting requirement
- So another algorithm is designed using **`test_and_set()`** instruction that satisfies all he critical-section requirements
- The common data structures are:

```
boolean waiting[n];
boolean lock;
```

- Both initialized to **`false`**

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

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

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable **`available`** indicating if lock is available or not

Mutex Locks

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

acquire() and release()

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


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

Mutex Locks

- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions

- Main disadvantage of this solution is, it requires **busy waiting**
  - This lock therefore called a **spinlock** because the process "spins" while waiting for the lock to become available.

SEMAPHORE

Semaphore

- 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()`**

Semaphore

- Definition of the `wait() operation`

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

- Definition of the `signal() operation`

```
signal(S) {

     S++;
}
```

Semaphore Usage

- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- **Counting semaphore** – integer value can range over an unrestricted domain
- Can solve various synchronization problems

Semaphore Usage

- Consider ***P**₁* and ***P**₂* that require ***S**₁* to happen before ***S**₂*

P1

```
……;

……;

……;

S1  ;

……;
```

P2

```
……;

……;

……;

S2  ;

……;
```

Semaphore Usage

- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "`synch`" initialized to 0

P1

```
……;
……;
S1 ;
signal(synch);
……;
```

P2

```
……;
……;
wait(synch);
S2 ;
……;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

Semaphore Implementation

- Like mutex locks, semaphores too have the problem of **busy waiting** in critical section implementation
- To overcome this, we modify the wait() and signal() and make the process to block itself.

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- 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

- we define a semaphore as follows:

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

Implementation with no Busy waiting (Cont.)

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

Implementation with no Busy waiting (Cont.)

```
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

P_0 P_1

```
        wait(S);                    wait(Q);
         wait(Q);                   wait(S);
 ...           ...
         signal(S);                signal(Q);
          signal(Q);                signal(S);
```

Deadlock and Starvation

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
  - Can happen if queue is LIFO

Priority Inversion

- Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**