Parallel and Concurrent Programming CS3S666

Semaphores

Last Time

- Looked at Shared Memory.
- Looked at race conditions.

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- 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 **count** that keeps track of the number of full buffers. Initially, count 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 and put in nextProduced  */
      while (counter == BUFFER_SIZE);
   // do nothing
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer

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

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

//  consume the item in nextConsumed
}
```

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 {count = 6 }

  S5: consumer execute counter = register2 {count = 4}

Critical Section Problem

- Consider system of n processes {p_0, p_1, … 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
- The critical section problem is to design a protocol to control 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 p_i is

```
do
{
  entry section
  Critical section
  remainder
}while (it needs to);
```

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 processes that wish to enter their critical section, then the selection of the process that wants to 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

Semaphore

- Synchronization tool that does not require busy waiting – what is this??

- In software engineering **busy-waiting** or **spinning** is a technique in which a process repeatedly checks to see if a condition is true.

- In general busy-waiting/spinning is considered an anti-pattern and should be avoided, as processor time that could be used to execute a different task is instead wasted on useless activity.

Semaphore

- Designed E. W. Dijkstra in the late 1960s
- Semaphore *S* – integer variable
- Two standard operations modify S: wait() and signal()

Semaphore

- S can only be accessed via two indivisible (atomic) operations

```
wait (S)
{
        while (S <= 0){}; // no-op
        S--;
}
signal (S)
{
         S++;
}
```

Semaphores as General Synchronization Tools

- **Counting** semaphore – integer value can range over an unrestricted domain
- **Binary** semaphore – integer value can range only between 0 and 1;
  - can be simpler to implement
  - sometimes known as **mutex locks**

Semaphore

- Provides mutual exclusion

```
Semaphore mutex;    //  initialized to 1
do {
        wait (mutex);
         // Critical Section
          signal (mutex);
        // remainder section
} while (TRUE);
```

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

- Applications may spend lots of time in critical sections and therefore this may not be a good solution

Semaphore Implementation

- Better (?) to implement Semaphores 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

Implementation of wait:

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

Implementation of signal:

```
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);

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when a lower-priority process holds a lock needed by higher-priority process

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 itemAvailable initialized to the value 0
- Semaphore spaceAvailable initialized to the value N

Bounded Buffer Problem (Cont.)

- The structure of the producer process:

```
do  {
//   produce an item in nextp
             wait (spaceAvailable);
              wait (mutex);

              //  add the item to the  buffer

              signal (mutex);
              signal (itemAvailable);
} while (TRUE);
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do
{
        wait (itemAvailable);
        wait (mutex);

              //  remove an item from  buffer to nextc

        signal (mutex);
        signal (spaceAvailable);

             //  consume the item in nextc
} 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 mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait (wrt) ;

            //    writing is performed

        signal (wrt) ;
} while (TRUE);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
        wait (mutex) ;
        readcount ++ ;
        if (readcount == 1)
            wait (wrt) ;
        signal (mutex)

            // reading is performed

        wait (mutex) ;
        readcount  - - ;
        if (readcount  == 0)
            signal (wrt) ;
        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

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

Using a Semaphore

```
1   #include<cstdlib>
2   #include<iostream>
3   #include<unistd.h>
4   #include <sys/wait.h>
5   #include <sys/types.h>
6   #include <sys/stat.h>
7   #include <fcntl.h>
8   #include <sys/ipc.h>
9   #include <sys/shm.h>
10  #include<semaphore.h>
11
12  Use these headers and compile with
13  g++ -pthread -o etc.
```

Using a Semaphore

```
12  int main()
13  {
14    //spawn a child process
15    int p = fork();
16    if (p > 0)
17    {
18      cout <
19      sleep(
20      cout <
21      wait(N
22    }
23    else if
24    {
25      cout <
26      sleep(2);
27      cout << "Child: Done sleeping" << endl;
28    }
29  }
30
```

```
simon@simon-VirtualBox:~$ g++ -pthread -o Reader Reader.cpp
simon@simon-VirtualBox:~$ ./Reader
Parent: Sleeping for 2 seconds
Child: Sleeping for 2 seconds
Parent: Done Sleeping
Child: Done Sleeping
simon@simon-VirtualBox:~$
```

```cpp
int main() {
  // create shared memory region
  key_t segid = ftok(".", 'r');

  int shmid;
  shmid = shmget(segid, sizeof(sem_t), IPC_CREAT | 0660);
  void* sh_mem = shmat(shmid, (void*)0, 0);

  //create POSIX semaphore
  sem_t* sem = (sem_t*)sh_mem;
  int sts = sem_init(sem, true, 1);

  //spawn a child process
  int p = fork();
  if (p > 0) {
    sem_wait(sem);
    cout << "Parent: Sleeping for 2 seconds" << endl;
    sleep(2);
    cout << "Parent: Done sleeping" << endl;
    sem_post(sem);
    wait(NULL);
  }
  else if (p == 0) {
    sem_wait(sem);
    cout << "Child: Sleeping for 2 seconds" << endl;
    sleep(2);
    cout << "Child: Done sleeping" << endl;
    sem_post(sem);
  }
}
```

Initialisation

Indicates available across multiple processes.

Memory for semaphore

Decreases counter

Increases counter

Next Week

- Petri Nets to model Concurrent Systems.