CS 180 *Quiz #4* Name GOH WEI ZHE

DigiPen Institute of Technology November 9, 2020

7/20

Do all questions. The total marks is 20. 5 bonus points.

1. The family of computers called X97 supports an atomic instruction known as CAS (Compare and Swap). The following **pseudocode** shows the functionality of CAS.

false, 0, 1

```
int CompareAndSwap(int *a, int old, int new)
{
                  expected
    if(*a==old) {
        *a = new;  ← set new value
        return 1;  ← true
    }
    else
        return 0;  ← false
}
```

In reality, CAS is a single atomic instruction. Probably it is used like this:

```
CAS mem, reg1, reg2
/* mem is the address we are checking i.e., the int * above,
   reg1 is the old value
   reg2 is the new value.
   CAS is an atomic instruction
*/
```

You are required to use the CAS instruction to implement a C++ struct named CASLock. It has the following declaration:

```
struct CASLock
{
private:
    int lock;
public:
    CASLock();
    void lock();
    void unlock();
};
```

A typical use of the CASLock looks like the following:

```
CASLock mtx; /* shared variable */

void foo()
{
   mtx.lock();
   /*Some critical section - mutual exclusion is a must */
   mtx.unlock();
}
```

1

(a) Provide a definition for the constructor of `CASLock`. [1 point]

CASLock() : lock{0}{}

~~CAS to~~

CASLock :: CASLock() : lock(0) {}

Answer:

0

(b) Provide a definition for the `lock` member function of `CASLock`. You may assume that we can call a function called `CompareAndSwap` that will use the CAS instruction. [2 points]

```
void lock ()
{
    lock = 1;
}
```

```
void CASLock :: lock()
{
    while (! CompareAndSwap (&lock, 0, 1));
}
```

Answer:

1

(c) Provide a definition for the `unlock` member function of `CASLock` [1 point]

```
void unlock()
{
    lock = 0;
}
```

```
void CASLock :: unlock()
{
    lock = 0;
}
```

Answer:

2. Assume for a preemptive, multitasking OS, that n processes, $P_1 \ldots P_n$ are using a TestAndSet instruction to achieve mutual exclusion (as shown below) into the critical section. Indicate whether the following statements are true or false. (3 points)

```
1 while(true)
2 {
3    while(TestAndSet(&lock)); // do nothing
4     //critical section
5    lock=0;
6 }
```

3

(a) Every process executing at line 3 eventually gets it's turn to enter the critical section. mutual exclusive F/T

(b) TestAndSet being a function, is unable to guarantee atomicity, hence it is possible for more than 1 process to be in the critical section at the same time. F/T

(c) It is gauranteed that *some* process executing at line 3 eventually get it's turn to enter the critical section. F/T

3. Which of the following is an example of a *race condition?* Choose one answer. (2 points)

(a) A variable is shared between two threads. Its value depends on the particular order in which the access takes place during run-time.

~~(b)~~ The main thread creates a child thread. The main thread may finish before the child thread does.

~~(c)~~ The user runs a program. The user may log out before the program finishes.

(d) The main thread has two child threads. The first thread may finish before the second.

(e) none of the above

A

2

4. Provide a solution for dining philosopher's problem using one mutex (binary semaphore). Philosophers need to acquire the mutex before picking the chopsticks. Assume there are two functions wait(mutex *x) and signal(mutex *x) for mutex. The original solution without considering synchronization is given as follows. Note that you are only required to re-write the code within the while loop. (4 points)

```
mutex m; ///mutex for synch on accessing chopsticks[n]
int chopstick[N]; ///bookkeeping for whether chopsticks are taken
void Dining(int n) //n is index of philosopher
{
    while (true) {
        get_left();
        get_right();
        eat();
        think();
    }
}
```

Answer:

```
void Dining (int n)
{
 while (true)
 {
  wait (mutex x);
  if((n+1 %N) > n)
  {
   wait ( chopstick[N] );
   get -left();
   wait (chopstick[N+1]);
   get-right ();
  }
  else
  {
   wait (chopstick[N+1] );
   get -right();
   wait ( chopstick [N]);
   get-left();
  }
  eat ();
  signal (mutex x);
  think();
}
```

```
mutex m;
int chopstick[N];
void Dining (int n) // n is index of philosopher
{
 while (true)
 {
  wait (&m);
  if (chopstick [n] == 0 && chopstick [(n+1)% N] == 0)
  {
   chopstick[n] = 1;
   chopstick[(n+1)%N] = 1;
   signal (&m);
   get-left ();
   get-right ();
   eat();
  }
  else
  {
   signal (&m);
   Continue;
  }
  wait (&m);
  chopstick[n] = 0
  chopstick [(n+1)%N] = 0;
  signal (&m);
  think();
 }
```

5. Consider the following code where each bank account is associated with a particular semaphore. The semaphores are declared in the following way:

```
sempahore_t bank_sem[MAX_ACCOUNT];
```

Usually, mutual exclusive access to a bank account would be achieved this way.

```
...
     wait(bank_sem[acc]);
     / * process account */
     ...
     signal(bank_sem[acc]);
...
```

Consider the following code that performs the transfer of deposits between two accounts.

```
void transfer(int from_account, int to_account, int amt)
{
     wait(bank_sem[from_account]);
     wait(bank_sem[to_account]);
     ...
     / * transfer amt btw from_account to to_account */
     ...
     signal(bank_sem[to_account]);
     signal(bank_sem[from_account]);
}
```

Now, the mainframe of the bank computer supports multi-threading. Therefore, there's a chance for multiple threads to call the same `transfer` function at the same time. What is wrong with the function above? (2 points)

multiple threads will enter the critical section at the same time, the function must only enable one process to enter the critical section while the second process waits till the first process to finish before executing.

Deadlock. As thread 1 that waits for bank_sem[from_account] after acquiring bank_sem[to_account] may not be able to enter into CS when bank_sem[from_account] has been acquired by other thread (eg. thread 2) that also waits for the resource (bank_sem[to_account]), which has been acquired by thread 1 -

Answer: a circular wait.

6. Give an alternative code that fixes the above code. [3 points]

```
void transfer ( int from-accant , int to-accont , int amt )
{
  /* wait according to the number from low to high or high to low one
     point for comparision of from-accont and to_account one point for two
     functions of wait() */

  if (from_accont > to-accont)
  {
    wait (back_sem[from_accont]);
    wait (back-sem [to-accont]);
  }
  else
  {
    wait(bank_sem[to-accont]);
    wait(back_sem[from-accont]);
  }
  ....
  /* transfer amt between from_accont to to-accont */
  ....
  signal (bank_sem[to-accont]);
  signal (bank-sem [from-accont]);
}
```

O

Answer:

7. For the following questions assume that we have three processes P_0, P_1, P_2 ordered in *increasing* priorities. You **must** use the following set of statements in your answer to the questions. It is possible for a statement to be used multiple times in your answer.

i Statements indicating which processes are in the **ready queue initially**.

  i. P_0 and P_2 in ready queue.

  ii. P_0, P_1 and P_2 in ready queue.

ii Statements indicating which processes are **sharing the critical section**.

  i. P_0 and P_2 share a critical section.

  ii. P_0, P_1 and P_2 share a critical section.

iii Statements indicating **which process is running**.

  i. P_0 runs.

  ii. P_1 runs.

  iii. P_2 runs.

iv Statements indicating which process is **blocking/sleeping** to go into critical section.

  i. P_0 blocks on waiting to go into critical section.

  ii. P_1 blocks on waiting to go into critical section.

  iii. P_2 blocks on waiting to go into critical section.

v Statements indicating which process is **busy–waiting** to go into critical section.

  i. P_0 busy waits to go into critical section.

  ii. P_1 busy waits to go into critical section.

  iii. P_2 busy waits to go into critical section.

vi Statements about **processes entering critical section**.

  i. P_0 enters critical section.

  ii. P_1 enters critical section.

  iii. P_2 enters critical section.

vii Statements about **processes leaving the critical section**.

  i. P_0 leaves critical section.

  ii. P_1 leaves critical section.

  iii. P_2 leaves critical section.

viii Statements about processes entering waiting state to **wait for I/O**.

  i. P_0 enters waiting state for I/O event.

  ii. P_1 enters waiting state for I/O event.

  iii. P_2 enters waiting state for I/O event.

ix Statements about processes' **I/O arriving** and processes going to **ready state**.

  i. P_0's I/O event arrives and becomes ready

  ii. P_1's I/O event arrives and becomes ready

  iii. P_2's I/O event arrives and becomes ready

x Statements about processes **never entering critical section** and the **reasons why not**.

i. P_1 never enters critical section because P_0 doesn't get to run.
ii. P_2 never enters critical section because P_0 doesn't get to run.
iii. P_2 never enters critical section because P_1 is running.
iv. P_2 never enters critical section because P_0 is running.

xi **Conclusions** on why the current state is kind of a deadlock/livelock.

i. As long as P_0 is running, both P_1 and P_2 cannot progress.
ii. As long as P_1 is running, both P_0 and P_2 cannot progress.
iii. As long as P_2 is running, both P_0 and P_1 cannot progress.

- (2 points) Assuming that the OS is employing preemptive priority scheduling and the critical sections are protected by busy waiting mechanisms. Show a sequence that leads to priority inversion. (Hint: Only 2 processes are involved so far)

i) i
ii) i
iii) i
iv) iii
v) iii
vi) i
vii) i
viii) iii
ix) iii
x) iv
xi) i

i) i
ii) i
iii) iii
viii) iii
iii) i
vi) i
ix) iii
iii) iii
v) iii
x) ii

Answer:

0

- (2 bonus points) Assume now that the critical sections are now protected by sleep and wakeup mechanism instead of busy waiting. Show a similar sequence that avoids the problem described in the previous question. (Hint: Still only 2 processes are involved so far)

i) i	i) i
ii) i	ii) i
iii) i	iii) iii
iv) iii	viii) iii
v) iii	ii) i
vi) i	vi) i
vii) i	ix) iii
viii) iii	iii) iii
ix) iii	iv) iii
x) iv	iii) i
xi) i	vii) i

Answer:

- (3 bonus points) Assume now that the critical sections are now protected by sleep and wakeup mechanism. Give a sequence to show a sequence of execution where a higher priority process is prevented from entering its critical section because of a lower priority process. (Hint: 3 processes involved).

i) ii	i) ii
ii) ii	ii) i
iii) i	iii) iii
iv) ii	viii) iii
v) iii	ii) ii
vi) i	viii) ii
vii) i	ii) i
viii) ii	vi) i
ix) ii	ix) iii
x) iii	iii) iii
xi) ii	ix) ii
	iv) iii
	iii) ii
	x) ii
	xi) ii

Answer: