Homework 14

Problem 1

Assume our physical memory can contain at most 4 pages and we are using **clock algorithm** as our replacing mechanism with the following design:

1) the clock pointer points to position 0 initially.

2) we do not reset the clock pointer after we have found an evict page, so next time we start from the position after previous victim.

3) Each page brought in were set as accessed initially.

1. Please complete the following table

Time	1	2	3	4	5	6	7	8	9	10	11	12
reference string	0	1	2	3	4	0	3	1	2	4	3	0
Memory	0	0	0	0	4	4	4	4	4	4	4	4
content	-	1	1	1	1	0	0	0	0	0	3	3
after	-	-	2	2	2	2	2	1	1	1	1	0
reference	-	-	-	3	3	3	3	3	2	2	2	2
remove	-	-	-	-	0	1	--	2	2	--	0	1
bring in	0	1	2	3	4	0	--	1	3	--	3	0

2. Assume the cost of memory is 100ns and the cost of accessing disk is 10ms, please compute the AMAT of the previous access.

AMAT = 100ns*(2/12) + 10ms*(10/12) = 8.33 ms

Problem 2

Consider the code of M&S queue described in class, answer the following questions.

1) After executed which line will the enqueue operation be visible to other dequeue threads? Why?

```
1 void Queue_Enqueue(queue_t *q, int value) {
2    node_t *tmp = malloc(sizeof(node_t));
3    assert(tmp != NULL);
4    tmp->value = value;
5    tmp->next = NULL;
6    pthread_mutex_lock(&q->tail_lock);
7    q->tail->next = tmp;
8    q->tail = tmp;
9    pthread_mutex_unlock(&->tail_lock);
10 }
```

Line 7. Because after line 7, a new node will be added to the end of the queue. Even if tail hasn't been changed, you can always find the new node from the head and therefore your dequeue operation can see the new node.

2) Now we are going to try to remove the lock in our concurrent queue. Read the following dequeue code and try to fill in the blank with CAS operation below.

`CAS(void **ptr, void *expected, void *new)` : if the pointer **pointed by `ptr`** equals to **`expected`**, then replace it with **`new`** and return 1. Return 0 otherwise.

```
int Queue_Dequeue(queue_t *q, int* value) {
   while (1) {
      node_t *tmp = q->head;
      node_t *new_head = tmp->next; // skip the dummy node
      if (head == q->head) {
         if (new_head == NULL) {
            return -1; // queue was empty
         }
         *value = new_head->value;
         if (CAS(q->head, tmp, new_head))
            break;
      }
   }
   free(tmp);
   return 0;
}
```

3) Read the enqueue code with CAS operation below. Is there any concurrent problems? Why?

```
1  void Queue_Enqueu(queue_t *q, int value) {
2     node_t *tmp = malloc(sizeof(node_t));
3     assert(tmp != NULL);
4     tmp->value = value;
5     tmp->next = NULL;
6     while (1) {
7        node_t *tail = q->tail;
8        node_t *next = tail->next;
9        if (tail == q->tail) {
10          if (CAS(&tail->next, next, tmp)
11             break;
12       }
13    }
14   CAS(&q->tail, tail, tmp);
15}
```

Yes. Because the CAS operation of tail->next and tail are not atomic. Assume two threads T1 and T2 are doing T1:enqueue(1) and T2:enqueue(2). If T1 get scheduled out after it inserted 1 into the queue but hasn't modified tail (just before line 14), then T2 will insert 2 at the same position of T1's

1 and the two enqueue operations will resulting in only one new node added.

Problem 3

Read the following lock implementation, what's the problem here? Try to fix it by using set_park()

```
void lock (lock_t *lock) {
   while (TestAndSet(&lock->guard, 1) == 1)
      ; // acquire guard lock by spinning
   if (lock->flag == 0) {
      lock->flag = 1; // lock is acquired
      lock->guard = 0;
   } else {
      queue_add(lock->q, gettid());
      setpark();
      lock->guard = 0;
      park();
   }
}

void unlock (lock_t *lock) {
   while (TestAndSet(&lock->guard, 1) == 1)
      ; // acquire guard lock by spinning
   if (queue_empty(lock->q)) {
      // let go of lock; no one wants it
      lock->flag = 0;
   } else {
      // hold lock (for next thread!)
      unpark(queue_remove(lock->q));
   }
   lock->guard = 0;
}
```

There is a wakeup/waiting race. If a thread acquiring a lock is scheduled out before its operation and is scheduled back after the previous holder's unpark operation finished, then it will never be unparked by others.

This can be solved by adding setpark like code above.