CS241 SP15 Exam 5: Solution Key

Name: Unni, N. **UIN:** 664350897

Exam code: DBAACC **NetID:** `nunni2`

SCROLL TO THE NEXT PAGE TO REVIEW YOUR ANSWERS

A VERSION OF THESE QUESTIONS MAY APPEAR IN A FUTURE QUIZ

1. (1 point.) This question assumes knowledge of the `work4hire` dining philosophers problem previously studied in your discussion section. Which response best describes the following proposed solution to the dining philosophers problem? Assume all mutex locks are correctly initialized.

```
void* work4hire(void*param) {
        CompanyData *company = (CompanyData*)p;
        pthread_mutex_t * a = company->prog_left;
        pthread_mutex_t * b = company->prog_right;

       while(running) {

         pthread_mutex_lock( a );

         while( 1 ) {
             int lock2 =  pthread_mutex_trylock( b ); // nonzero if failed

             if( 0 == lock2)  { // Success - we can do some work

               usleep( 1+ rand() % 3); // do work!
               pthread_mutex_unlock( b );
             } else {
               usleep(1); // Wait a bit and try again
             }

          }
         pthread_mutex_unlock( a );

       }
       return NULL;
  }
```

(A) No deadlock but the solution suffers from livelock (i.e. starvation is still possible).

(B) There is no pre-emption so deadlock is impossible.

(C) Locks are acquired in the same order so circular wait is impossible

(D) Each thread acquires two mutex locks so deadlock is possible

(E) Each thread acquires two mutex locks so deadlock is impossible

2. (1 point.) There are 8 threads and two global arrays: An array of 8 initialized mutex locks, and and array of 8 data values. Each thread concurrently executes the following code. Each thread receives a different starting value of i (valued 0 to 7 inclusive). Which response best describes the following code?

```
int running = 1;
pthread_t threads[8];
double data[8];
pthread_mutex_t mutexlocks[8];
// initialization code not shown

void* threadfunc(void*param){
  int i = (int) param;
  int j = (i+1)  & 7;
  while(running) {
    if( i != j ) {
      pthread_mutex_lock( &mutexlocks[i] );
      pthread_mutex_lock( &mutexlocks[j] );
      swap( data, i , j ); // swaps values[i] & values[j]
      pthread_mutex_unlock( &mutexlocks[i] );
      pthread_mutex_unlock( &mutexlocks[j] );
      usleep(1 + rand() & 3); // sleep for a few microseconds
    }
}
```

(A) Circular wait is impossible but all 8 threads can still deadlock

(B) Circular wait is possible and it is possible for just 2 threads to deadlock

(C) Circular wait is possible and deadlock is possible once all 8 threads are running.

(D) Deadlock is impossible because hold-and-wait is not satisfied

(E) Livelock and starvation are possible.

3. (1 point.) The following code is an attempt to implement the READER-WRITER solution by wrapping an existing map (that has no synchronization support e.g. no mutex locks). Read the code very carefully. Which response best describes this implementation? Note function names have been shortened for clarity (e.g. pthread_mutex_lock) . You may assume variables are correctly initialized where appropriate.

```
map_t* map; // assume points to a valid associative map implementation
// cv = condition variable, m = mutex

double read(int key) {
  lock(&m);
  readers++;
  unlock(&m);

  double r = get(map, key);

  lock(&m);
  readers--;
  cond_broadcast(&cv);
  unlock(&m);
  return r;
}

double write(int key, double score) {
  lock(&m);
  while(readers>0) cond_wait(&cv, &m);

  put(map, key, score);

  cond_broadcast(&cv);
  unlock(&m);
}
```

(A) None of the other responses are correct

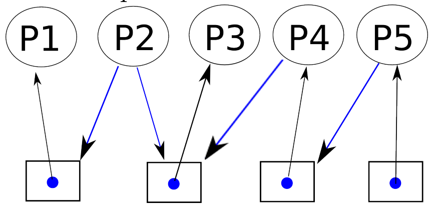
(B) A reader may call `get` at the same time as a writer calls `put`

(C) Suffers from deadlock if both reader and writer are called at the same time

(D) Satisfies many readers, exclusive single writer requirements but a writer may suffer starvation

(E) Multiple writers may call `put` at the same time

4. (1 point.) Use the Resource Allocation Graph below to determine deadlock in the following system. You may assume a process is able to finish if it is not waiting for any resources and that it releases all resources upon completion.



(A) 3 processes are or will be deadlocked

(B) 2 processes are or will be deadlocked

(C) 4 processes are or will be deadlocked

(D) 5 processes are or will be deadlocked

(E) No process is or will be deadlocked

5. (1 point.) Read the following code very carefully. It represents a multithreaded producer consumer implementation attempt with a queue size of 1. Which response best describes this implementation?

```
double saved;
// Assume counting semaphores {\tt sA} and {\tt sB} are
// initialized with count values 1 and 0 respectively

void insert(double value) {
  sem_post(sB);
  saved = value;
  sem_wait(sA);
}
double remove() {
  sem_post(sA);
  double result  = saved;
  sem_wait(sB);
  return result;
}
```

(A) It is possible to remove a value before any value has been inserted

(B) The queue will suffer from livelock if `insert` and `remove` are called at the same time

(C) The queue will suffer from deadlock if `insert` and `remove` are called at the same time

(D) The queue will suffer from deadlock if `remove` is called twice before another thread calls `insert`

(E) The queue will suffer from deadlock if `insert` is called twice before another thread calls `remove`

6. (1 point.) Review the multi-threaded code below for synchronization errors. The two methods are used to increase or decrease `money`. The `withdraw` method should block until there are sufficient funds in the account `(money>= amount)`, Note `PTHREAD_COND_INITIALIZER` is equivalent to `pthread_cond_init`, and the argument `amount` will always contain a positive value.

```
 int money = 100; /* Must be positive */
 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
 pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

 void deposit(int amount) { /* add money */
  pthread_mutex_lock(&m);
  money =  money + amount;
  pthread_mutex_unlock(&m);
 }
// withdraw should block;
// it will return once there is sufficient money in the account
 void withdraw(int amount) { /* reduce money */
   if(money < amount) pthread_cond_wait(&cv,m);
   else pthread_cond_signal(&cv);
   money = money - amount;
 }
```

Which one of the following is a true statement about the synchronization used in above functions?

(A) The `withdraw` method must call `pthread_mutex_lock` the mutex after `pthread_cond_wait` returns

(B) `pthread_cond_signal` should be wrapped inside a while loop

(C) The `withdraw` method contains no synchronization errors

(D) None of the ofter responses are correct

(E) The `deposit` method needs to call `pthread_cond_wait`

7. (1 point.) Spot the error(s)! 10 threads will call `barrier` just once. The first 9 threads should block i.e. wait until the 10th thread calls `barrier`, then all 10 threads should continue. Review the multi-threaded code below for synchronization errors. Note `PTHREAD_COND_INITIALIZER` is equivalent to `pthread_cond_init`

```
int count=10;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

void barrier() {
  pthread_mutex_lock(&m);
  count--;
  if(count >0) {
      while(count>0) pthread_cond_wait(&m,&cv);
  } else {
       pthread_cond_signal(&cv);
  }
  pthread_mutex_unlock(&m);
}
```

Which one of the following is true for the code above?

(A) When the 10th thread calls `barrier`, no threads will continue executing code after the `barrier` call

(B) When the 10th thread calls `barrier`, 10 threads will continue executing code after the `barrier` call

(C) Two or more threads can continue executing code after the `barrier` call before the 10th thread calls `barrier`

(D) When the 10th thread calls `barrier`, 2 threads will continue executing code after the `barrier` call

(E) One thread can continue executing code after the barrier call before the 10th thread calls `barrier`

8. (1 point.) Which response best describes the additional code required to implement a barrier? Assume the mutex lock (`m`) was correctly initialized and the counting semaphore (`sem`) was initialized with a count of 0. Only when the 10th thread calls barrier() can all 10 threads will (eventually) unblock and continue.

```
int count;

// The first 9 calls to barrier will block
void barrier() {
  pthread_mutex_lock(m);
  count++;
  if(count == 10)
       sem_post(sem);

  ?? Missing Code??
}
```

(A) `pthread_mutex_unlock(m); sem_wait(sem); sem_post(sem);`

(B) `sem_post(sem); pthread_mutex_unlock(m); sem_wait(sem);`

(C) `pthread_mutex_unlock(m); sem_wait(sem);`

(D) `pthread_mutex_unlock(m); sem_post(sem); sem_wait(sem);`

(E) `sem_post(sem); sem_wait(sem); pthread_mutex_unlock(m);`

9. (1 point.) Riddle me this! Which one of the following is the odd-one out?

(A) “Once a process acquires a resource, another process cannot force the original process to release it”

(B) “There exists a set of process P1,P2,... such that there is a process dependency cycle in the Resource Allocation Graph”

(C) “A process is able to execute but never able to acquire all necessary resources to make progress”

(D) “The resources can not be shareable between processes at the same time”

(E) “Once a process acquires a resource, it will retain the resource and wait for the next resource”

10. (1 point.) What are the correct initial values for the three counting semaphores sA,sB,SC so that the following are thread-safe multithreaded stack push and pop functions? Assume `sem_post` is never interrupted by a signal and N = maximum capacity of array.

```
void push(double value) {
  sem_wait(&sA);
  sem_wait(&sC);
  stack[ count++ ] = value;
  sem_post(&sC);
  sem_post(&sB);
}
```

```
double pop() {
  sem_wait(&sB);
  sem_wait(&sC)
  double result = stack[ --count ];
  sem_post(&sC);
  sem_post(&sA);
  return result;
}
```

(A) `sA(1), sB(N) sC(N)`

(B) `sA(N), sB(0) sC(0)`

(C) `sA(N), sB(0) sC(1)`

(D) `sA(0), sB(N) sC(1)`

(E) `sA(0), sB(N) sC(0)`

11. (1 point.) Identify the following definition "There exists a set of process P1,P2,... such that there is a process dependency cycle in the Resource Allocation Graph"

(A) No Pre-emption

(B) Circular Wait

(C) Mutual Exclusion

(D) Hold and Wait

(E) Livelock

12. (1 point.) Which response best describes the following scenario? Alice and Bob race to the toy shed to pick up a bat and ball. Both Alice and Bob will only pick up the bat if they can also pick up the ball at the same time.

(A) Example of livelock

(B) No mutual exclusion

(C) No circular wait

(D) No pre-emption

(E) No hold and wait

13. (1 point.) Which response best describes the following scenario? Alice owns markers and is willing to share them while she is drawing. Bob owns paper and is willing to share to share it while he is drawing. Chris owns pencils and paper. Alice and Bob both want to draw with markers on the paper at the same time.

(A) No circular wait

(B) Example of livelock

(C) No mutual exclusion

(D) No pre-emption

(E) No hold and wait

14. (1 point.) Which response best describes the following implementation attempt at a barrier? The barrier is required to block until the 10th thread has called `barrier`.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;

int count;

void barrier() {
  pthread_mutex_lock(&m);
  while(count < 10) pthread_cond_wait(&cv, &m);
  count ++;
  if(count ==10) pthread_cond_broadcast(&cv);
  pthread_mutex_unlock(&m);
}
```

(A) Threads will continue when the 9th thread calls `barrier`.

(B) The first thread will continue once another thread calls `barrier`

(C) All threads will be blocked indefinitely

(D) The implementation is correct but inefficient

(E) The implementation is correct and uses condition variables efficiently

15. (1 point.) Identify the following definition “A process is *not* deadlocked but is never able to acquire all necessary resources to make progress”

(A) Hold and Wait

(B) Livelock

(C) Circular Wait

(D) No Pre-emption

(E) Mutual Exclusion

16. (1 point.) Identify the following definition “Once a process acquires a resource, it will retain the resource and wait for the next resource”

(A) No Pre-emption

(B) Circular Wait

(C) Mutual Exclusion

(D) Hold and Wait

(E) Livelock

17. (1 point.) Two threads attempt to lock two mutex locks (see pseudo code below) before entering a critical section and become deadlocked. Which response is NOT true? You can assume all locks are released after the critical section.

```
THREAD 1:
  pthread_mutex_lock(m1) // success
  pthread_mutex_lock(m2) // blocks
  // critical section ...
  pthread_mutex_unlock(m2)
  pthread_mutex_unlock(m1)

THREAD 2:
  pthread_mutex_lock(m2) // success
  pthread_mutex_lock(m1) //blocks
  // critical section ...
  pthread_mutex_unlock(m2)
  pthread_mutex_unlock(m1)
```

(A) If the first thread had successfully locked both `m1`, `m2` the given code would not deadlock

(B) Deadlock can still occur even if you swap the order of the two `lock` calls (lock `m1` before `m2`) in thread 2 code

(C) If a third thread locks `m1` or `m2` it will also be deadlocked

(D) The given sequence appears as a circular dependency in the Resource Allocation Graph

(E) Deadlock can still occur even if you swap the order of the two `unlock` calls (unlock `m1` before `m2`) in thread 2 code

18. (1 point.) In this example, which Coffman condition is *NOT* satisfied? Before leaving the bus depot, the bus driver requires two resources: a route and a bus. Once all routes are assigned, a driver may drive a bus on a route that already has one bus assigned.

(A) Circular Wait

(B) No Pre-emption

(C) Hold and Wait

(D) None of the other responses are correct

(E) Mutual Exclusion

19. (1 point.) Identify the following definition “Once a process acquires a resource, another process cannot force the original process to release it”

(A) Livelock

(B) Hold and Wait

(C) No Pre-emption

(D) Mutual Exclusion

(E) Circular Wait

20. (1 point.) This question assumes knowledge of the `work4hire` dining philosophers problem previously studied in your discussion section. Which response best describes the following proposed solution to the dining philosophers problem?

```
void* work4hire(void*param) {
        CompanyData *company = (CompanyData*)p;
        pthread_mutex_t * a = company->prog_left;
        pthread_mutex_t * b = company->prog_right;

       while(running) {
         // Use pointer value to determine lock ordering
         if( a < b) pthread_mutex_lock( a );
         pthread_mutex_lock( b );
         if( b < a)  pthread_mutex_lock( a );

         usleep( 1+ rand() % 3); // do work!
         pthread_mutex_unlock( a );
         pthread_mutex_unlock( b );
       }
       return NULL;
  }
```

(A) Each thread acquires two mutex locks so deadlock is impossible

(B) The solution suffers from livelock once all 5 threads are running.

(C) Each thread acquires two mutex locks so deadlock is possible

(D) Locks are acquired in the same order so circular wait is impossible

(E) There is no pre-emption so deadlock is impossible.

21. (1 point.) Which response best describes the following scenario? Alice and Bob race to the toy shed to pick up a bat and ball. If Alice has already picked up the ball, Bob will let go of the bat so that Alice can use it.

(A) No circular wait

(B) Example of deadlock

(C) No mutual exclusion

(D) No hold and wait

(E) No pre-emption

22. (1 point.) Which response best describes the following code? You may assume that funcA locks the mutex before the second thread.

```
int iteration =0; [SPRING 2015 THIS LINE WAS MISSING]
int x=0;
pthread_cond_t cv1 = PTHREAD_COND_INITIALIZER;
pthread_cond_t cv2 = PTHREAD_COND_INITIALIZER;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
void* funcA(void* p) {

  pthread_mutex_lock(&m);
  while(iteration < 20) {
      while( 0== x) {
          pthread_cond_signal(&cv2);
          pthread_cond_wait(&cv1, &m);
      }
      x --;
      printf("%d\n",x):
      iteration ++;
   }
}
void* funcB(void* p) {
  sleep(1); // assume funcB looses the race to first lock the mutex
  pthread_mutex_lock(&m);
  while(iteration < 20) {
      while(4==x ) {
          pthread_cond_signal(&cv1);
          pthread_cond_wait(&cv2, &m);
      }
      x ++;
      printf("%d\n",x):
      iteration ++;
   }
}
int main() {
 pthread_t tid1,tid2;
 pthread_create(&tid1, NULL, funcA, NULL);
 pthread_create(&tid2, NULL, funcB, NULL);
 pthread_join(tid1,NULL);
 pthread_join(tid2,NULL);
 return 1;
}
```

(A) Prints the following 4 values (one per line) `1 2 3 4` and then stops

(B) Deadlocks and does not print anything

(C) None of the other responses are correct

(D) Prints the following 20 values (one per line) `1 2 3 4 3 2 1 0 1 2 3 4 3 2 1 0 1 2 3 4`

(E) Prints the following 20 increasing values (one per line) `1 2 3 4 5 6 7 8 9 10 ... 20`

23. (1 point.) Identify the following definition "The resources can not be shareable between processes at the same time"

(A) Hold and Wait

(B) No Pre-emption

(C) Mutual Exclusion

(D) Circular Wait

(E) Livelock

24. (1 point.) Which one of the following is NOT true for the Reader Writer Problem?

(A) When there is an active writer the number of active readers must be zero

(B) A writer must wait until the current active readers have finished

(C) The number of active readers must be less than two

(D) If there is an active reader the number of active writers must be zero

(E) The number of active writers plus the number of active readers must be greater than zero

Summary of answers:

Question	Correct Answer	Your Answer	Points
1	A	A	1
2	C	C	1
3	D	D	1
4	E	E	1
5	A	A	1
6	D	D	1
7	D	B	0
8	A	C	0
9	C	C	1
10	C	C	1
11	B	B	1
12	E	E	1
13	C	C	1
14	C	C	1
15	B	B	1
16	D	D	1
17	B	B	1
18	E	A	0
19	C	C	1
20	D	D	1
21	E	D	1
22	D	B	1
23	C	C	1
24	C	C	1
Total			21