CS241 SP15 Exam 5: Solution Key

Name: Geng, Y. **UIN:** 656934543

Exam code: AAABEB **NetID:** yangeng2

SCROLL TO THE NEXT PAGE TO REVIEW YOUR ANSWERS

A VERSION OF THESE QUESTIONS MAY APPEAR IN A FUTURE QUIZ

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

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

(B) "Once a process acquires a resource, another process cannot force the original process to release it"

(C) "The resources can not be shareable between processes at the same time"

(D) "Once a process acquires a resource, it will retain the resource and wait for the next resource"

(E) "A process is able to execute but never able to acquire all necessary resources to make progress"

2. (1 point.) Which one of the following is true for the Reader Writer Problem?

(A) Reader must must block if the queue is empty

(B) A writer must block if the queue is full

(C) Readers cannot be in the critical section at the same time

(D) Only one reader and one writer can be in the critical section at the same time

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

3. (1 point.) Identify the following definition “The resources can not be shareable between processes at the same time”

(A) No Pre-emption

(B) Mutual Exclusion

(C) Circular Wait

(D) Hold and Wait

(E) Livelock

4. (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) Livelock and starvation are possible.

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

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

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

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

5. (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) Mutual Exclusion

(C) Hold and Wait

(D) Circular Wait

(E) Livelock

6. (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 hold and wait

(D) No mutual exclusion

(E) No pre-emption

7. (1 point.) Identify the following definition “Once a process acquires a resource, it will retain the resource and wait for the next resource”

(A) Mutual Exclusion

(B) Circular Wait

(C) Hold and Wait

(D) Livelock

(E) No Pre-emption

8. (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? Function names have been shortened for clarity (e.g. `pthread_mutex_lock` is written as `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++;
  while(writers>0) cond_wait(&cv, &m);
  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 || writers) cond_wait(&cv, &m);
  writers++;
  unlock(&m);

  put(map, key, score);

  lock(&m);
  writers--;
  cond_broadcast(&cv);
  unlock(&m);
}
```

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

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

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

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

(E) None of the other responses are correct

9. (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(N), sB(0) sC(1)`

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

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

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

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

10. (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) Deadlocks and does not print anything

(B) 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`

(C) None of the other responses are correct

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

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

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

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

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

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

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

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

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

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

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

13. (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 counts 1 and 1 respectively
void insert(double value) {
  sem_wait(sA);
  saved = value;
  sem_post(sB);
}
double remove() {
  sem_wait(sB);
  double result  = saved;
  sem_post(sA);
  return result;
}
```

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

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

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

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

(E) It is possible to remove a value before `insert` is called

14. (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) None of the ofter responses are correct

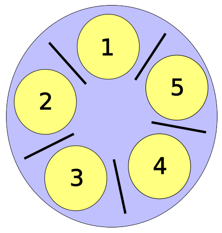
(B) The `withdraw` method contains no synchronization errors

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

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

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

15. (1 point.)



Five philosophers are invited to sit at a circular table. Each philosopher's brain has only two states: Either thinking or hungry! Around the table I place five chopsticks (one chopstick between each philosopher). A philosopher can eat once they are exclusively holding the two chopsticks that are nearest to them. If a philosopher is unable to pick up the other chopstick after 60 seconds, then they will put their first chopstick down and think for 60 seconds before trying again. Unfortunately being too smart they all use the timers on their smart-watches and all attempt to pick up one chopstick on their left at the same time, then wait for another chopstick, give up, release their first chopstick, and try again, and again and again for ever... This scenarios is best described as

(A) Starvation due to livelock

(B) Starvation due to deadlock

(C) Deadlock due to pre-emption

(D) A solution to the Dining Philosophers problem

(E) Deadlock due to hold and wait

16. (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) Mutual Exclusion

(B) None of the other responses are correct

(C) Circular Wait

(D) No Pre-emption

(E) Hold and Wait

17. (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) No mutual exclusion

(B) No pre-emption

(C) No circular wait

(D) Example of livelock

(E) No hold and wait

18. (1 point.) Which response best describes the additional code required to implement the following barrier? Every 2nd thread to call `barrier` unblocks the previous thread and both threads continue. Assume the mutex lock (`m`) is correctly initialized and the counting semaphore (`sem`) is initialized with a count of 0.

```
int count;

void barrier() {
  pthread_mutex_lock(m);
  int result = (++count) & 1; // 1,0,1,0,1...
  ?? Missing code ??
}
```

(A) `pthread_mutex_unlock(m); if( result ) sem_wait(sem); else sem_post(sem);`

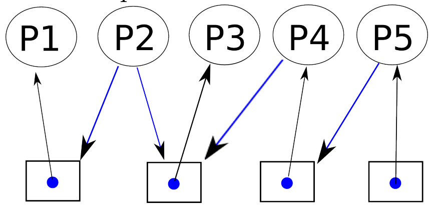
(B) `sem_wait(sem); pthread_mutex_unlock(m); if( result ) { sem_post(sem); }`

(C) `if( result ) sem_post(sem); else sem_wait(sem); pthread_mutex_unlock(m);`

(D) `if( result ) sem_wait(sem); else sem_post(sem); pthread_mutex_unlock(m);`

(E) None of the other responses are correct.

19. (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) 2 processes are or will be deadlocked

(B) 4 processes are or will be deadlocked

(C) No process is or will be deadlocked

(D) 3 processes are or will be deadlocked

(E) 5 processes are or will be deadlocked

20. (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) All threads will be blocked indefinitely

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

(C) The implementation is correct but inefficient

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

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

21. (1 point.) Identify the following definition "A process is *not* deadlocked but is never able to acquire all necessary resources to make progress"

(A) Mutual Exclusion

(B) Hold and Wait

(C) Livelock

(D) Circular Wait

(E) No Pre-emption

22. (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 pre-emption

(B) No mutual exclusion

(C) No circular wait

(D) Example of livelock

(E) No hold and wait

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

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

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

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

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

24. (1 point.) Identify the following definition “Once a process acquires a resource, another process cannot force the original process to release it”

(A) No Pre-emption

(B) Circular Wait

(C) Mutual Exclusion

(D) Livelock

(E) Hold and Wait

Summary of answers:

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