ECE 254 Final Exam Solutions

J. Zarnett, C. Moreno

December 10, 2017

(1A)

(a) No deadlock exists and no single resource request would cause a deadlock.

(b) No deadlock exists, but if P1 requests R2 it will cause a deadlock.

(c) A deadlock exists: P1 - P4 - P2 - P3

(d) No deadlock exists and no single request would cause a deadlock.

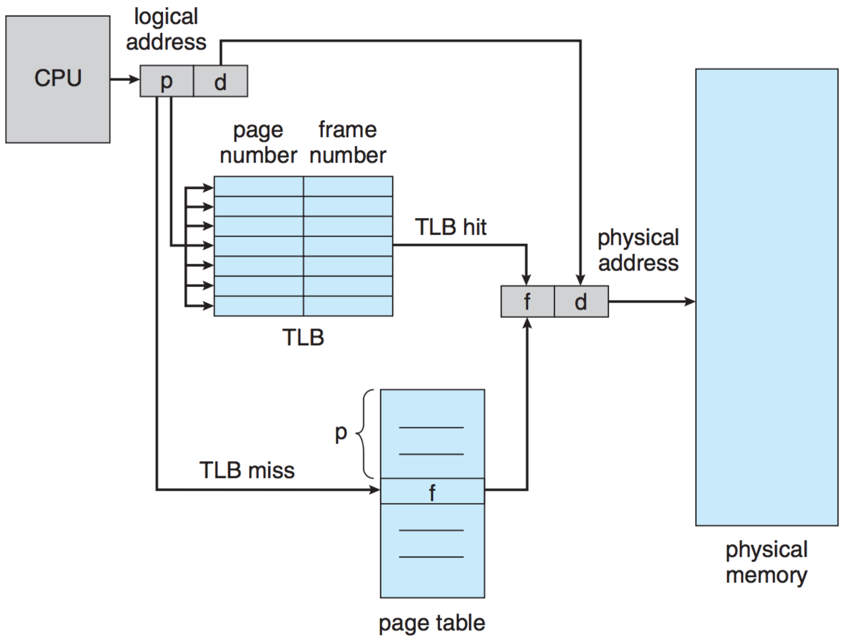
(1B)

1. Data copying: rather than different threads operating on shared data, they each get a copy of the data to work on. This can speed up the code dramatically by removing the need for locking, but it might also not be possible or might lead to some threads getting an out of date view of he data.

2. Lock ordering: assign a correct order in which data elements are locked to prevent a cycle from being formed in the resource allocation graph. It will prevent deadlock and can be tested for using some tools, but it can be hard to enforce when locks don't have the same names everywhere, and adds overhead to writing the code.

3. Two-phase locking: attempt to lock all resources and release those that were locked if not all resources were received. It avoids deadlock and allows programs to do useful work while waiting for a resource to be free, but it is easy to make mistakes and can lead to live-lock.

(2A) A reasonable text description of this diagram gets full marks:



(2B)

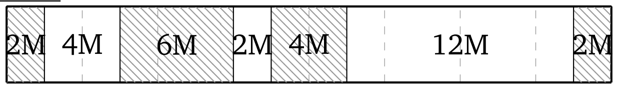
Part 1. Breakdown:

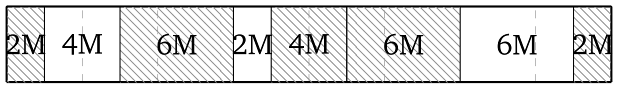
Part 2.

Virtual Address	Physical Address	Seg. Fault?	Page Fault?
0x00000020FF			
0x00000032FF			
0x0001003A00			
0x00010007FF			

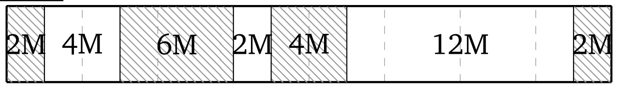
(2C)

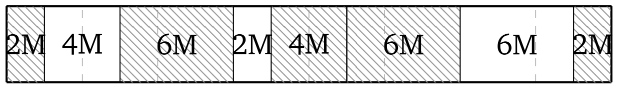
First Fit

1. 

2. 

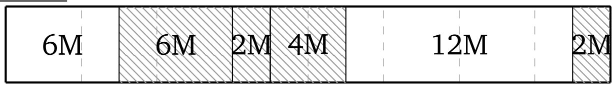
Next Fit

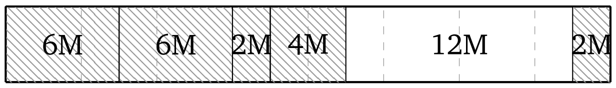
1. 

2. 

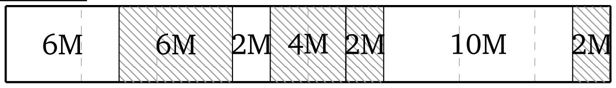
Yes, it is the same as first fit. This happens because the 2MB block is the last allocated one and it's at the very end.

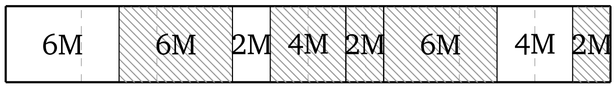
Best Fit

1. 

2. 

Worst Fit

1. 

2. 

(3A)

1. `vruntime`: The virtual runtime of a task, i.e., and adjusted value for how much CPU time a process has received.
2. Red-Black Tree: The data structure used to store tasks in a sorted manner such that the one with the lowest vruntime value is always the leftmost node.
3. `nice`: the priority of a process which is used as an adjustment factor for vruntime. A high nice value means the vruntime will be larger than the physical runtime; a negative value means the vruntime will be smaller than the physical runtime.
4. Target Latency: A period of time in which every thread (or task) gets to run at least once.

(3B)

(3C)

(4A)

Advantages: 1. It is easier for application developers where they don't have to make such system calls; 2. Better performance: system calls are expensive and instead of an open and then a read, for example, we only do the read and don't have to do two system calls.

Disadvantages: 1. It is no longer possible to specify you want to open something read-only or in shared mode; 2. A long running program might keep a file locked for a very very long time even though other programs want to use that file.

(4B)

Algorithm	**Service Order**	**Cylinders Moved**	**Improvement over FCFS**
FCFS	490, 142, 163, 19, 168, 167, 36	$438 + 142 + 21 + 144 + 149 + 1 + 131 = 1026$	1.000
SSTF	36, 19, 142, 163, 167, 168, 490	$16 + 17 + 123 + 21 + 4 + 1 + 322 = 504$	2.036
SSTF+DB3	142, 163, 490, 168, 167, 19, 36	$90 + 21 + 327 + 322 + 1 + 148 + 17 = 926$	1.108
SSTF+DB5	19, 142, 163, 168, 490, 167, 36	$33 + 123 + 21 + 5 + 322 + 323 + 134 = 961$	1.068
SCAN	142, 163, 167, 168, 490, (499), 36, 19	$90 + 21 + 4 + 1 + 322 + 9 + 463 + 17 = 927$	1.107

(4C)

How NTFS uses journalling to make sure the system is in a consistent state: first the changes are written to the log, then the changes are made to the volume in the cache, then the log file is written to disk, and only after that write to disk is completed then the changes can be made to the volume in the cache. This means that if there is a crash, at the time of the crash there will be 0 or more transactions in the log. If 0, the state is consistent; if more than 0 then there are some partially done operations and then the log file contains enough information to undo those changes and get the system back to a consistent state.

(5A)

```
void init( ) {
  pthread_mutex_init( &searcher_mutex, NULL );
  pthread_mutex_init( &inserter_mutex, NULL );
  pthread_mutex_init( &perform_insert, NULL );
  sem_init( &no_inserters, 0, 1 );
  sem_init( &no_searchers, 0, 1 );
  searchers = 0;
  inserters = 0;
}

void* searcher_thread( void *target ) {
  pthread_mutex_lock( &searcher_mutex );
  searchers++;
  if ( searchers == 1 ) {
    sem_wait( &no_searchers );
  }
  pthread_mutex_unlock( &searcher_mutex );

  search( target );

  pthread_mutex_lock( &searcher_mutex );
  searchers--;
  if ( searchers == 0 ) {
    sem_post( &no_searchers );
  }
  pthread_mutex_unlock( &searcher_mutex );
}
```

```
void* deleter_thread( void* to_delete ) {
  sem_wait( &no_searchers );
  sem_wait( &no_inserters );

  delete( to_delete );

  sem_post( &no_inserters );
  sem_post( &no_searchers );
}


void* inserter_thread( void *to_insert ) {
  pthread_mutex_lock( &inserter_mutex );
  inserters++;
  if ( inserters == 1 ) {
    sem_wait( &no_inserters );
  }
  pthread_mutex_unlock( &inserter_mutex );

  pthread_mutex_lock( &perform_insert );
  insert( to_insert );
  pthread_mutex_unlock( &perform_insert );

  pthread_mutex_lock( &inserter_mutex );
  inserters--;
  if ( inserters == 0 ) {
    sem_post( &no_inserters );
  }
  pthread_mutex_unlock( &inserter_mutex );}
```

(5B)

The problem is deadlock. The scenario occurs if in parallel thread 1 called `swap(a, b)` and thread 2 called `swap(b, a)` and each of them get the first lock and then is blocked on the second lock.

An alternative answer is also a deadlock: a call to `swap(a, a)` which will cause a problem because the mutex locks are not re-entrant but that is not something I'd expect people to know in this course.

(5C)

1. If philosophers choose at random which chopstick they pick up first, it is less likely that deadlock will occur but it can still happen. Imagine the decision as a coin flip. It could happen that all philosophers get heads when they flip the coin, and all choose the leftmost chopstick first and thus all get deadlocked.

2. If an additional chopstick is added to the table, deadlock will not occur. By the pigeonhole principle, if there are n philosophers and $n + 1$ chopsticks and all chopsticks are taken, then at least one philosopher has 2 chopsticks and therefore will be able to eat and then put the chopsticks down, allowing another philosopher to eat and so on.

3. This will prevent deadlock: it knocks down the pillar of "no preemption" which is one of the necessary conditions for deadlock to be possible.

(6A)

(6B)