ECE254 Final F2017

1A. A) No deadlock, no single request would cause deadlock.

B) No deadlock, if P1 requests R2 then deadlock.

C) Deadlock: P1 - P4 - P2 - P3

D) No deadlock, no single request would cause deadlock.

1B. 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by removing the need for locking, but it might also not be possible or might lead to some threads getting an out of date view of he data.

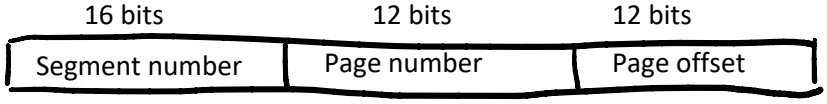
Lock ordering: assign a correct order in which data elements are locked to prevent a cycle from being formed in the resource allocation graph. It will prevent deadlock and can be tested for using some tools, but it can be hard to enforce when locks don't have the same names everywhere, and adds overhead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

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. See official solution for diagram.

The logical address is composed of a page number and an offset. We search the TLB for the page number - search happens on all entries simultaneously. If there is a TLB hit, we have obtained the frame number. If there is a TLB miss, we look in the page table for the frame number. Now that we have obtained the frame number, we combine it with the offset to form the physical address.

2B. Part 1. Since the page size is 4KB = 2^12 bytes, we need 12 bits to represent the page offset. Since each segment is at most 16MB = 2^24 bytes and a page is 4KB = 2^12 bytes, we need (24-12 = 12) bits for our page number. Then we have

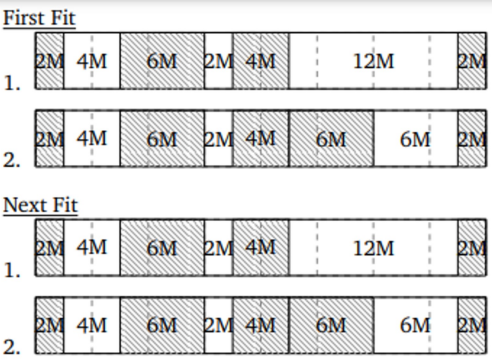


Part 2. [Not sure on this one]

Virtual Address	Physical Address	Seg. Fault?	Page Fault?
0x00000020FF Segment 0 Page 2 Offset 0x0FF	0x7000 + 0x0FF = 0x000070FF	No	No
0x00000032FF Segment 0 Page 3 Offset 0x2FF	0x4000 + 0x2FF = 0x000042FF	No	No
0x0001003A00 Segment 1 Page 3 Offset 0xA00	0x5000 + 0xA00 = 0x00005A00	No	No
0x00010007FF Segment 1 Page 0 Offset 0x7FF		No	Yes

2C.

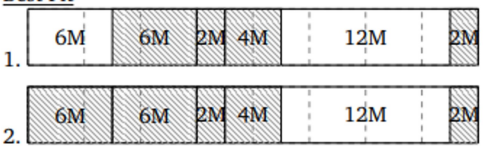
First Fit



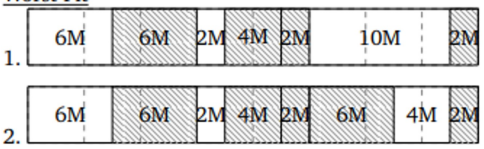
Next Fit

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



Worst Fit



3A.

vruntime: The virtual runtime is the adjusted value for how much CPU time a task has received.

Red-black tree: The red-black tree stores tasks so that the one with the lowest vruntime is always the left-most node.

nice value: The nice value is the priority of a process, which is used as an adjustment factor for vruntime.

A high nice value means vruntime will be larger than the actual execution time, and a negative value means the vruntime will be smaller than the actual execution time.

Target latency: The amount of time in which all threads (tasks) should have gotten to run at least once.

3B. This algorithm essentially selects the highest priority ready task. The first problem is that lower priority processes could starve, if there are always higher priority processes that are ready. The second problem is that if all tasks are blocked, the dispatcher will dispatch task 0 anyways (??).

3C. Case 1: Yes. Task 2 has higher priority, so it will run when ready. It has I/O operations (connecting to the server and reading data), so when this task is blocked on I/O, task 1 will get to run.

Case 2: No. Task 1 has higher priority and never gets blocked or terminates, so it will run indefinitely and task 2 will never get a chance to run.

Case 3: No. Since the two tasks have the same priority, task 1 will run first. But it never gets blocked or terminates, so unless there is time slicing, it will run indefinitely and task 2 will never get to run.

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 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	1026	1.000
SSTF	36, 19, 142, 163, 167, 168, 490	504	2.036
SSTF + Double Buffering Buffer size 3	142, 163, 490, 168, 167, 19, 36	926	1.108
SSTF + Double Buffering Buffer size 5	19, 142, 163, 168, 490, 167, 36	961	1.068
SCAN	142, 163, 167, 168, 490, (499), 36, 19	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 disk. 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_searchers, 0, 1);
        sem_init(&no_inserters, 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.

We could have deadlock in the scenario below. Let x and y both be of type container_t*.

Thread A calls swap(x, y)
Lock x
Process switch: thread B is selected to run
Thread B calls swap(y, x)
Lock y

Now thread A is blocked because y is locked, and thread B is blocked because x is locked.

5C.

1. No. It makes deadlock less likely but still possible. If philosophers choose at random, there is still the possibility that all the philosophers choose to pick up the left chopstick at the same time, and we are deadlocked.
2. Yes. By the pigeonhole principle, if there are n philosophers and n+1 chopsticks, then at least one philosopher will have 2 chopsticks and will get to eat, put the chopsticks down, allowing another philosopher to eat and so on.
3. Yes. One of the requirements for deadlock is 'no preemption'. If we introduce preemption, no deadlock can occur: a philosopher can steal a chopstick from another, eat, and put the chopsticks down, allowing another philosopher to eat.

6A. [Not covered]

1. Yes. Encapsulation breaks up the application into more isolated modules. This means that each object contains its own data and functionality and hides its internal implementation from other objects. This can help minimize the impact of an attack on one object on the rest of the application, as the attack may be contained within the affected object.
2. No. A bug in one thread may affect other threads, since threads share a memory space with other threads in the same process.
3. Yes. Since each process has its own memory space and set of resources, which can help prevent one process from accessing or modifying resources of other processes if attached. A fault or crash in one process is less likely to affect other processes and bring down the whole system.

6B. [Not covered]