ECE 254 S15 Final Exam Solutions

(1A)

Longer time slices reduce the amount of time lost to overhead (the work of switching processes) but make individual processes less responsive to the user (because it takes longer for any individual process to get a turn to run).

(1B)

Remember that slack time is calculated as (deadline - execution): the maximum we can delay starting the task and still meet the deadline. So, calculated for each task:

$A - 2009, B - 191, C - 390, D - 505$.

Thus the schedule order is B, C, D, A.

(1C)

Shortest Job First, as you should recall, is not about the total predicted process time, but the predicted next CPU burst. Yes, that middle column is there as a trap.

The schedule algorithm is just interested in the rightmost column, so the order is A, B, D, C.

(1D)

Two of the following:

1. Fairness - because the I/O-bound process issues requests and gets blocked relatively quickly, its share of the CPU time is disproportionately small.

2. Resource utilization – I/O devices are often slow compared to the CPU so to get maximum system performance it is best to keep them busy all the time. If I/O bound processes are given some priority then they will quickly issue I/O requests and keep the devices busy.

3. Responsiveness: I/O bound processes may be user interactive so we'd like to give the users a better experience by making their programs more responsive.

(1E)

Priority inheritance is the idea of raising the priority of a process temporarily. If low priority process P_2 has a resource locked and high priority process P_1 is waiting, we raise the priority of P_2 to that of P_1 so that P_2 will be scheduled to run and can finish and unlock the resource. Then the priority of P_2 falls back to its previous value. The problem it solves: we do not want a high priority process waiting for a low priority process.

(1F)

1. If each CPU has its own cache (L1, L2, L3...) then processor affinity is a performance increase because it results in fewer cache misses. If a process P_1 ran on CPU-2 then its pages will be in that CPU's cache; if it moves to CPU-7 then there will be numerous cache misses.

2. If the system is NUMA (Non-Uniform Memory Access), then it will be better to keep a process on the CPUs closest to where its memory accesses are fastest.

(1G)

Constant Time Scheduler $O(1)$ (any five of the following, or other relevant points):

- 1 queue per priority level; 140 priority levels
- Constant ($O(1)$) runtime
- Active & Expired Queues
- Highest priority queue is chosen; round-robin scheduling in each queue.
- If a process is preempted before a full time slice it goes back in the ready queue
- If a process reaches the end of its time slice, it goes into the expired queue
- When the ready queue is empty, it swaps places with the expired queue
- Poor performance for interactive processes

Completely Fair Scheduler (CFS) (any five of the following, or other relevant points):

- Red-Black Tree to model ready queue
- Orders processes based on execution time
- Leftmost node is always the one scheduled
- Tracks execution time (`vruntime`)
- Logarithmic $O(\ln(n))$ runtime
- Target latency: specified period in which all tasks should run at least once
- Process priority alters the execution time calculation
- Group scheduling: can designate a group to be scheduled like a process

(2A)

1: A, 2: D, 3: A, 4: E

(2B)

Let h be the cache hit rate. Thus, (1-h) is the miss rate. In the event of a miss, 99% of the time it is in memory so it will be retrieves in 1 000 ns; 1% of the time it is on disk so we need to retrieve it from there at a cost of 10 000 000 ns.

Set up the equation:

Average access time = hit rate $\times$ cache access time + (1 - hit rate) $\times$ (memory hit rate $\times$ memory access time + (1 - memory hit rate) $\times$ disk access time)

Fill in the known values:

$750 = h \times 7 + (1 - h) \times (0.99 \times 1\,000 + 0.01 \times 10\,000\,000)$.

Solve for h: $0.99264 \rightarrow 0.993$

(2C)

First-In First-Out

Step	`cache[0]`	`cache[1]`	`cache[2]`	**Fault?**
1	1	—	—	✓
2	1	2	—	✓
3	1	2	3	✓
4	4	2	3	✓
5	4	2	3	–
6	4	1	3	✓
7	4	1	0	✓
8	6	1	0	✓
9	6	2	0	✓
10	6	2	1	✓
11	6	2	1	–
12	3	2	1	✓
13	3	7	1	✓
14	3	7	6	✓
15	3	7	6	–
16	2	7	6	✓
17	2	1	6	✓
18	2	1	6	–
19	2	1	3	✓
20	6	1	3	✓

FIFO Page Faults: 16

Least Recently Used

Step	`cache[0]`	`cache[1]`	`cache[2]`	**Fault?**
1	1	—	—	✓
2	1	2	—	✓
3	1	2	3	✓
4	4	2	3	✓
5	4	2	3	–
6	4	2	1	✓
7	0	2	1	✓
8	0	6	1	✓
9	0	6	2	✓
10	1	6	2	✓
11	1	6	2	–
12	1	3	2	✓
13	7	3	2	✓
14	7	3	6	✓
15	7	3	6	–
16	2	3	6	✓
17	2	3	1	✓
18	2	3	1	–
19	2	3	1	–
20	2	3	6	✓

LRU Page Faults: 15

(2D) Thrashing is when the system spends so much time swapping virtual memory pages in and out of memory that no useful work is getting done.

(3A)

FCFS: Just do them in order.
SSTF: At every stage, compute which is the closest, go to that.
SCAN: Increasing order all the way to the end, then reverse direction. So after servicing 472, the scan goes all the way to 499 and then reverses. C-SCAN: Increasing order, scan goes all the way to the end, at the end go back to 0 (a jump of 500 cylinders).

Strategy	Service Order	Total Cylinders Moved	Improvement over FCFS
FCFS	356, 9, 472, 375, 302	$47 + 347 + 463 + 97 + 73 = 1027$	1.000
SSTF	302, 356, 375, 472, 9	$7 + 54 + 19 + 97 + 463 = 640$	1.605
SCAN	356, 375, 472, 302, 9	$47 + 19 + 97 + (27 + 197) + 293 = 680$	1.510
C-SCAN	356, 375, 472, 9, 302	$47 + 19 + 97 + (27 + 500 + 9) + 293 = 992$	1.035

(3B)

Disk journalling is a technique for ensuring a consistent state of the file system at all times, even if there is a crash or power failure. In journalling, any operation takes place as a transaction: first all of the steps are written to a log file, then the changes are actually carried out, and only if the changes were all completed successfully is the transaction considered complete and removed from the log file. It is desirable because we never want the file system to be in an inconsistent state as that may lead to corrupt or damaged data. User data may still get lost, in this scheme, forever.

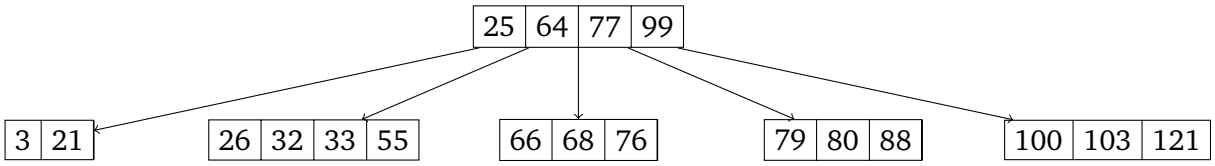
(3C)

B-Trees are used because we want to reduce disk read operations for directory structures. The B-Tree can be sized to equal a disk block, and allows for a lot of information to be retrieved in a single disk operation.

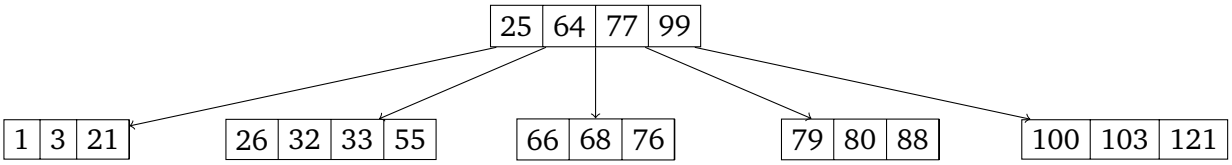
(3D)

Yes, it is possible to reconstruct the free space list. It is an expensive operation because the OS has to scan the disk, loading all the file control blocks, and identify which blocks are allocated. Given that list, all the blocks not in it are free and constitute the free space list.

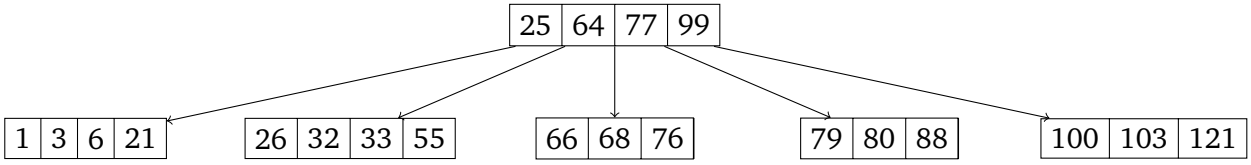
(3E) Initial State:



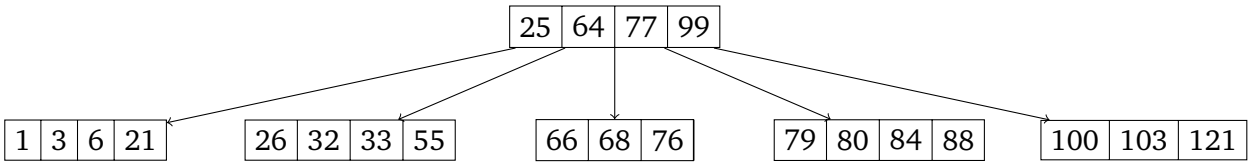
Insert 1: 1 is less than 25, so follow the leftmost pointer. In that block, we are at the leaf level, so insert it in order.



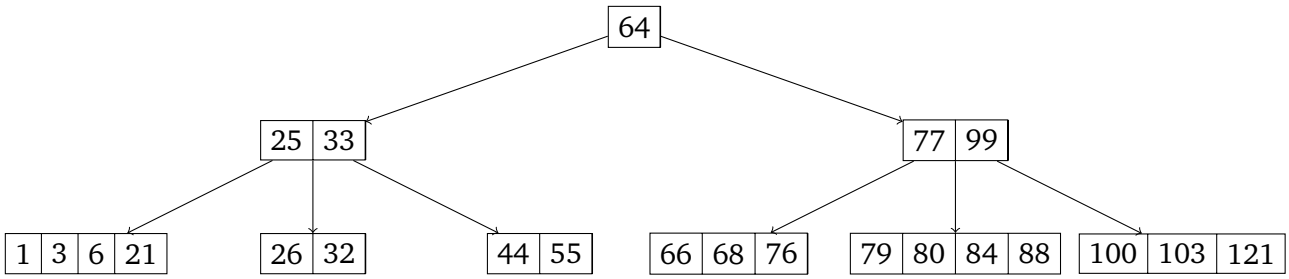
Insert 6: 6 is less than 25, follow the leftmost pointer. At that leaf, insert it in order.



Insert 84: 84 is greater than 25, greater, than 64, greater than 77, less than 99. So follow the pointer between 77 and 99. At this leaf node, insert 84.



Insert 44: 44 is between 25 and 64. Follow that pointer. That node is full, however, so we will have to split it. After the insertion the set would be {26, 32, 33, 44, 55}, making 33 the middle element that would get promoted up to the root node. But the root node is also full so we will have to split it as well. The root node will have {25, 33, 64, 77, 99}, so 64 is promoted up a level and the tree now has three levels.



(4A) Any four relevant points will do. Some samples:

In both shared memory and message passing, the operating system is invoked to set up communication: either it designates memory as shared, or it sets up the message queues.

Shared memory means both/all processes operate on the same data (shared region); with message passing, every process gets its own copy (message).

With shared memory, the processes are responsible for managing the shared area in its entirety; with message passing the OS plays a role in the message structure.

Shared memory is faster because it requires a lot fewer system calls.

Shared memory works only if the two processes are running at the same time (synchronously); with message passing the sender and receiver are indirectly connected via the operating system and can run asynchronously.

(4B)

1. Elimination of mutual exclusion is not possible. We discussed in class how this would be a cure worse than the disease, so to speak. But a chopstick is a physical item and only one person can be using it at a time.

2. Hold-and-wait: this can be eliminated; no philosopher may hold onto a chopstick while awaiting another. This is just like two phase locking (as was on the midterm). If a philosopher picks up a chopstick but cannot acquire the other he or she needs to eat, he or she should put down the one acquired.

3. No pre-emption: this can be eliminated. In the original construction of the problem, philosophers were polite and would not snatch the chopstick out of the hands of a colleague. If that is no longer the case, then a philosopher who "steals" a chopstick from his or her neighbour will eat and deadlock is avoided.

(4C)

```
/* Declare global variables here */
int pindex;
int cindex;
pthread_mutex_t mutex;
sem_t items;
sem_t spaces;

void *producer( void *void_arg ) {
    /* Implement producer code */
    int item = produce_item();

    sem_wait( &spaces );
    pthread_mutex_lock( &mutex );

    BUFFER[pindex] = item;
    pindex = (pindex + 1) % BUFFER_SIZE;

    pthread_mutex_unlock( &mutex );
    sem_post( &items );
    pthread_exit(0);
}

void *consumer( void *void_arg ) {
    /* Implement consumer code */

    sem_wait( &items );
    pthread_mutex_lock( &mutex );

    int item = BUFFER[cindex];
    BUFFER[cindex] = -1;
    cindex = (cindex + 1) % BUFFER_SIZE;

    pthread_mutex_unlock( &mutex );
    sem_post( &spaces );

    consume_item ( item );
    pthread_exit(0);
}

int main( int argc, char** argv ) {
/* Initialize global variables here */
pindex = 0;
    cindex = 0;

    pthread_mutex_init( &mutex, NULL);
    sem_init( &items, 0, 0 );
    sem_init( &spaces, 0, BUFFER_SIZE );

   /* Initialization and creation of the producer / consumer pthreads
      is not shown, but there can be arbitrarily many of each */
    pthread_exit(0);
}
```