ECE 254 F15 Final Solutions

J. Zarnett

December 16, 2015

(1A)

Temporal locality: a memory location that has been recently accessed is likely to be accessed again in the near future.

Spatial locality: a memory location near one that has been recently accessed is likely to be accessed in the near future.

(1B)

Average access time = hit rate × cache access time + (1 - hit rate) × (memory hit rate × memory access time + (1 - memory hit rate) × disk access time)

If we choose to increase the hit rate:

$a = 0.99 \times 7 + 0.01 \times (0.99 \times 1\,000 + 0.01 \times 10\,000\,000)$. Produces an average access time of 1016.83 ns.

If we choose to increase the hard drive speed:

$a = 0.98 \times 7 + 0.02 \times (0.99 \times 1\,000 + 0.01 \times 9\,000\,000)$. Produces an average access time of 1826.66 ns.

Conclusion: spend the money to increase cache hit rate!

(1C) If free memory is maintained in a linked list it's very inefficient to iterate over a large number of items in the linked list if the resultant blocks are all very small. It would be better to put those small fragments inside blocks and that would speed up the process of finding a free block of appropriate size.

Even if not using a linked list you could make the argument about the amount lost to overhead. Any free block having a header will consume nonzero space for the header and accordingly the more blocks, the bigger the space lost to overhead is.

(1D) Realloc will presumably try to extend the memory block. If the current block is size x and the new size is y (obviously greater than x), if the memory location for x is followed by a free block of size $(y - x)$ or larger, take that $(y - x)$ from the free block and add it to the allocated block for x. If the subsequent block can't be (partially-)merged with the block for x, then a new block of size y must be allocated and the data copied to the new block. A pointer to the new block is returned and the old block will be deallocated by the realloc routine.

(1E) Answers will obviously vary based on how your particular lab was implemented.

(2A)

The criteria are:

1. Turnaround time. (The amount of time between starting a process and when it finishes, including waiting)
2. Response time. (Time between putting in a request and getting answers back)
3. Deadlines.
4. Predictability.
5. Throughput.
6. Processor utilization.
7. Fairness.
8. Enforcing priorities.
9. Balancing resources.

(2B) If the task has a hard deadline and will take so long to execute that the deadline will not be met, the RTOS will simply not schedule it (because what's the point?!).

(2C)

A and D arrive at the same time and have equal response ratios of 1. So you have your choice about what to do first, A or D. Either is fine!

Suppose A goes first. It runs for 1058 units of time, during which time both C and E arrive. Evaluate R for processes C (w=654, R=1.946), D (w=1058, R=11.17), and E (w=963, R= 4.762). So choose D. It runs, bringing the time up to 1162. Evaluate C (w=758, R=2.097) and E (w=1067, R=5.168) again and E will execute for 256, bringing elapsed time up to 1418. B has not arrived yet, so choose C. It runs next; during its run B arrives and it runs last. So the order is: A D E C B.

Suppose D goes first: during this 104 units of time, process E arrives. After 104 time units, D is finished and we have to decide between A and E. A has a waiting time of 104 and E has waited 9 units of time. So A's calculated value is 1.098; E has a ratio of 1.035. So run A. A will finish at time 1162 during which time process C has arrived. Evaluate E (w=1067, R=5.168) and C (w=758, R=2.097) and choose E, which runs for 256, up to 1418. At this point B has not yet arrived so choose C and it runs; during its run B arrives and it runs last. So the order is: D A E C B.

(2D) The auxiliary queue is where processes that got blocked before reaching the end of their time slice go, and processes that are in this queue will have priority over those in the ready queue. Also acceptable as an answer: an extension of round robin scheduling structures to give priority to I/O bound processes to get better resource utilization in the system.

(2E)

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

(3A)

Strategy	Service Order	Total Cylinders Moved	Improvement
First Come First Served	66, 356, 297, 268, 461	217+290+59+29+193=788	1.000
Shortest Seek Time First	297, 268, 356, 461, 66	14+29+88+105+395=631	1.249
SCAN	268, 66, 297, 356, 461	15+202+(66+297)+59+105=744	1.059
LOOK	268, 66, 297, 356, 461	15+202+231+59+105=612	1.288

(3B) The OS maintains data about which files are currently in use, often in a system-wide and per-process table. The open and close system calls are the way a process tells the OS a file is in use and when it's finished with it.

(3C)

The contiguous allocation strategy means that a file occupies a set of contiguous blocks on disk. So a file is allocated, starting at block b and is n blocks in size, the file takes up blocks $b, b+1, b+2, ..., b+(n-1)$. This is advantageous, because if we want to access block b on disk, accessing $b+1$ requires no head movement, so seek time is nonexistent to minimal (if we need to move to another cylinder). If we need a memory block of size N: (1) can we find a contiguous block of N or greater to meet that allocation?

Linked allocation: maintain a linked list of the blocks, and the blocks themselves may be located anywhere on the disk. The directory listing just has a pointer to the first and last blocks (head and tail of the linked list).

If a new file is created, it will be created with size zero and the head and tail pointers are null. When a new block is needed, it can come from anywhere and will just be added to the linked list. Thus, compaction and relocation are not really an issue. Unfortunately, however, accessing block i of a file is no longer as simple as computing an offset from the first block; it requires following i pointers (a pain).

(4A)

1. Incorrect use of the API is anything where we use the functions incorrectly. So this might be something like calling the mutex unlock function on a mutex that is not currently locked.

2. Lock ordering problems is just a way of saying a potential deadlock. If thread A acquires locks in the order L1 then L2 and thread B acquires them in the order L2 then L1, there is the possibility of a deadlock, and Helgrind will detect this and report it.

3. A data race is just uncontrolled access to shared data. So if threads A and B are both writing to a global variable without that access being controlled through any synchronization constructs.

(4B)

The problem here is that the code doesn't run in parallel. In each iteration of the loop a new thread is created and starts, but the parent thread will immediately "join" it - wait for it to complete before continuing. Thus, each thread runs one after the other, so it is not parallel at all.

(4C) There are multiple acceptable answers to this question, if sufficiently well reasoned. One potential answer is yes, in the “base case” where there is exactly one process allowed to run at a time and they all run sequentially (and threads aren’t an issue) then you can’t get a deadlock because the definition of deadlock requires two or more processes to be stuck. But this is very much a special case and rarely would we want an operating system that works like that. But the question did not rule that out, so it’s a valid answer.

You most likely should argue no - no amount of scheduling can save bad programmers from themselves. There are simply too many ways for people to cause a deadlock to be able to categorically rule it out through scheduling. Remember that deadlock requires mutual exclusion, no pre-emption, hold-and-wait, and a cycle in the resource allocation graph. The question is if scheduling can prevent that final condition from coming to pass (it does nothing to stop the others). Ultimately, outside of very specific cases like the one in the previous paragraph, scheduling can’t prevent all deadlocks.

(4D)

```
pthread_mutex_t mutex;

pthread_t threads[NUM_WORKERS];
void *fetchAndExecute( void *void_arg );

/* When SIGINT received, cancel all pthreads */
void interruptHandler( int ignore ) {

  for (int i = 0; i < NUM_WORKERS; ++i) {
    pthread_cancel( threads[i] );
  }

  pthread_mutex_destroy( &mutex ); // Not strictly needed but good practice
}

int main ( int argc, char** argv ) {

    /* Define the interruptHandler function to catch SIGINT */
    signal(SIGINT, interruptHandler);

    pthread_mutex_init( &mutex, NULL );

    for ( int i = 0; i < NUM_THREADS; ++i ) {
       pthread_create( threads[i], NULL, fetchAndExecute, NULL);
       // Any argument is okay, even a null ptr, since we don't need it.
    }

    pthread_exit( 0 );
}

void fetchAndExecute( void *void_arg ) {
    // May safely ignore argument, not needed

    while( 1 ) { // or while true
        pthread_mutex_lock( &mutex );
        workunit* wu = (workunit*) pop_front( );
        pthread_mutex_unlock( &mutex );

        if ( wu == 0 ) {
            sleep( 1000 );
        } else {
            int result = execute( wu );

            if ( result != 0 ) {
                enqueue_workunit( wu ); // Can also do it manually, but no need.
            }
        }
    }
}


void enqueue_workunit( workunit* wu ) {
    pthread_mutex_lock( &mutex );
    push_end( wu );
    pthread_mutex_unlock( &mutex );

}
```