ECE 254 Final Exam Solutions

J. Zarnett

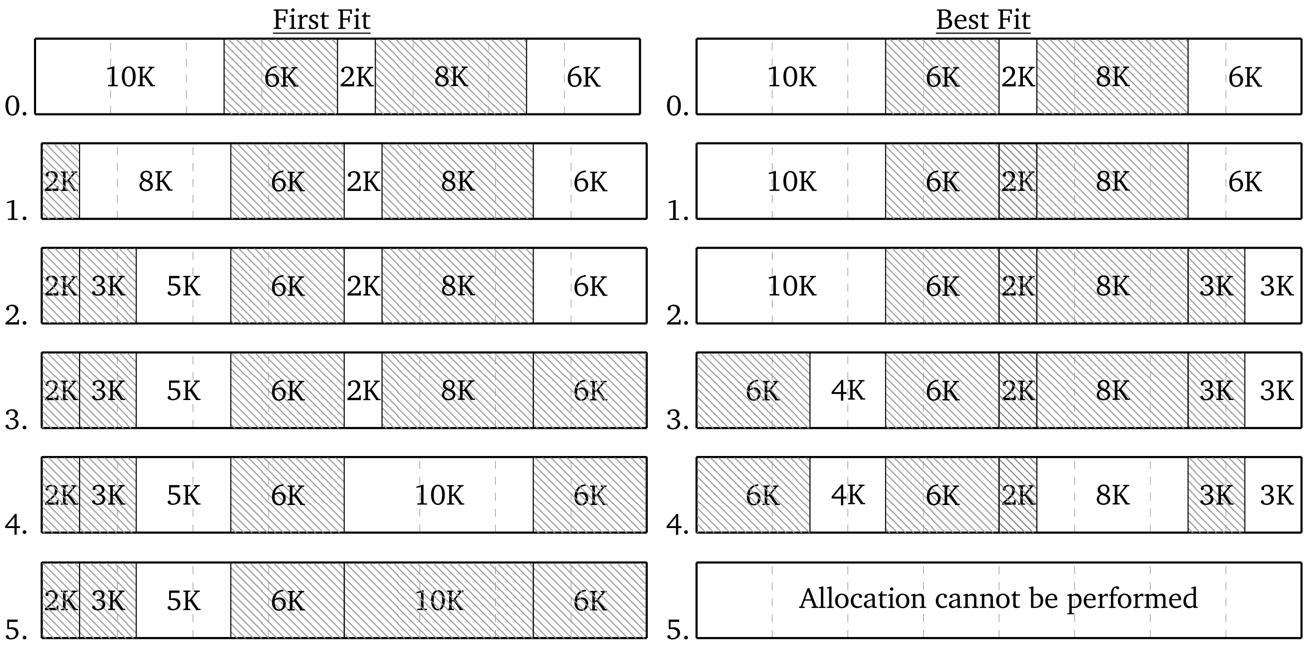
July 27, 2017

(1a)

```
int mode;
int num_frames;
int faults = 0;

typedef struct {
  unsigned long page;
  int updated;
} frame_t;

frame_t* frames;

unsigned long get_page_num( unsigned long address ) {
  return address / (unsigned long) 4096;
}

unsigned long* parse_args( int argc, char** argv ) {
  if ( argc < 4 ) {
    printf("Too␣few␣arguments.\n");
    abort();
  }
  mode = atoi( argv[1] );
  num_frames = atoi( argv[2] );

  unsigned long* addresses = malloc( (argc - 3) * sizeof( unsigned long ) );
  for ( int i = 3; i < argc; ++i ) {
    unsigned long parsed = strtoul( argv[i], NULL, 10 );
    unsigned long page = get_page_num( parsed );
    addresses[i - 3] = page;
  }

  return addresses;
}

void print_state( ) {
  printf("(");
  for ( int i = 0; i < num_frames - 1; i++ ) {
    if (frames[i].updated == -1) {
      printf("-,␣");
    } else {
      printf( "%lu,␣", frames[i].page );
```

```c
    }
  }
  if ( frames[num_frames - 1].updated == -1 ) {
    printf("-)\n");
  } else {
    printf("%lu)\n", frames[num_frames - 1].page );
  }
}

void simulate_fifo( unsigned long* addresses, int length ) {
  int frame_to_replace = 0;
  for( int i = 0; i < length; i++ ) {
    unsigned long p = addresses[i];
    // Check if it's already in there
    // If so then we are done
    bool found = false;
    for (int j = 0; j < num_frames; ++j ) {
      if (frames[j].page == p && frames[j].updated != -1) {
        // Finished -- found the page inside the frame
        found = true;
        break;
      }
    }
    if (!found) {
      ++faults;
      frames[frame_to_replace].page = p;
      frames[frame_to_replace].updated = i;
      // Update frame
      frame_to_replace = (frame_to_replace + 1) % num_frames;
    }
    print_state( );
  }
}

void simulate_lru( unsigned long* addresses, int length ) {
  int frame_to_replace;
  for( int i = 0; i < length; i++ ) {
    unsigned long p = addresses[i];
    // Check if it's already in there
    bool found = false;
    for( int j = 0; j < num_frames; ++j ) {
      if (frames[j].page == p && frames[j].updated != -1) {
        found = true;
        frames[j].updated = i;
        break;
      }
    }
    if (!found) {
      ++faults;
      for( int k = 0; k < num_frames; ++k ) {
        if ( frames[k].updated == -1 ) {
          frame_to_replace = k;
          break;
        }
        if (frames[k].updated < frames[frame_to_replace].updated ) {
          frame_to_replace = k;
        }
      }
      frames[frame_to_replace].page = p;
      frames[frame_to_replace].updated = i;
    }
    print_state( );
  }
}

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

  unsigned long* addresses = parse_args( argc, argv );

  frames = malloc( num_frames * sizeof( frame_t ) );
  for ( int j = 0; j < num_frames; ++j ) {
    frames[j].page = 0;
    frames[j].updated = -1;
  }

  if ( mode == 0 ) {
    simulate_fifo( addresses, argc - 3 );
  } else {
    simulate_lru( addresses, argc - 3 );
  }

  printf("Total␣number␣of␣page␣faults:␣%d.\n", faults );
  free( frames );
  free( addresses );
  return 0;
}
```

(1b)



(2A)

This scheduling algorithm heavily favours CPU-bound processes; that is an advantage for CPU bound processes and a disadvantage for I/O bound processes. This means it is likely to use resources inefficiently because slow I/O devices may be left idle for long periods of time. Furthermore this scheme has a lower limit for timeslice length but not an upper limit. Thus a program with an infinite loop could get a time slice length that approaches infinity.

(1 mark for recommendation) The algorithm should not be used due to its possibility of a process having unreasonably long time slices.

(2B)

A	A	A	A	B	B	B	A	–	C	D	D	D	D	C	E	E	F	F	–
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

(2C)

1. Round Robin with an infinite length timeslice is FCFS.

2. Shortest Process Next is about total length of execution and Shortest Job First is about next CPU burst time. No real relation in terms of parameters can be established.

3. Lottery can work out effectively like Round Robin in the long term if all processes get an equal number of tickets and random number generation is sufficiently random – but it's not the same because in lottery the same process can run multiple times and thus configuration does not quite make them identical. Although in theory one could have Round Robin behaviour by moving all the tickets from one process to the next so that each process gets 100% of the tickets, briefly, then loses them all to the next one.

4. The multilevel feedback queue can look like Highest Priority Period if we always take from the highest queue and the feedback process does not move a process down to a lower queue.

5. Lottery and Guaranteed Scheduling work out the same if lottery tickets are re-allocated on every run to make things as even as possible (i.e. processes that have run a lot will lose some to those who have run infrequently).

(2D)

Approach	**Advantage**	**Disadvantage**
OS Guesses Randomly	It's certainly easy...	But high chance of being wrong!
Developer Specifies at Compile-Time	Developers know better than users or random guessing what the program does.	Not always possible to know how the program will be used at run-time
User Specifies at Launch-Time	Poor system performance is the user's fault! (We like that)	Users are really bad at knowing these things... Do you trust them?
OS Scans Binary and Counts System Calls (many = I/O Bound)	Fair and potentially effective in determining	Inaccurate: a lot of system calls in the binary does not account for the run-time behaviour (1 call in a loop could run more than 50 individual calls)
OS Assumes CPU-Bound; Modifies if High I/O Usage	Most accurate way of determining, can change with time.	Most overhead and accounting

(3A)

If the value of C is too small the behaviour resembles FCFS – and we lose performance. If the value of C is too large then we risk starvation as a request might have to wait unreasonably long before execution.

(3B)

In theory you don't have to allocate these at the start of free space, but:

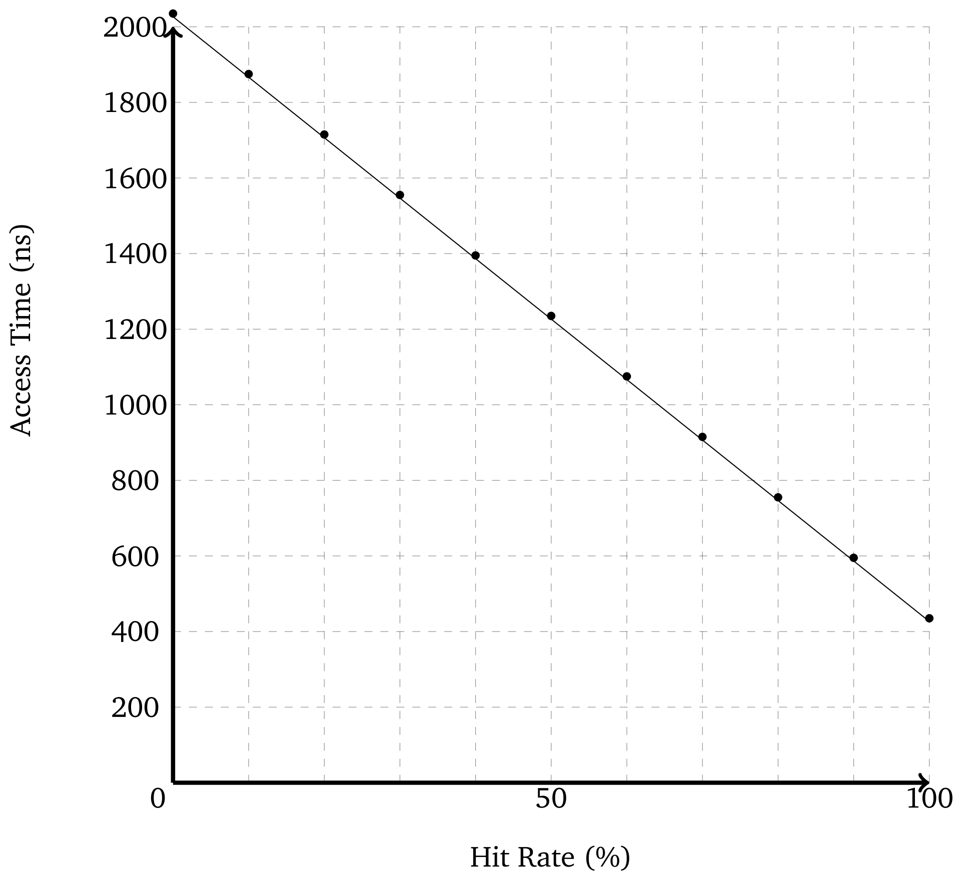
File Allocation Method	**Allocated Blocks for File F**	**Allocated Blocks for File G**
Contiguous Allocation	5, 6, 7, 8	9, 10
Contiguous Allocation, Best Fit	23, 24, 25, 26	14, 15
Linked Allocation	1, 5, 6, 7	8, 9

(3C)

1. The effective access time is 2026.66 ns.

2. With the modification for 20% of the time then the effective access time is 1706.66 ns.

3. The graph should begin at 0 but we will forgive if it is not. Not all labels need to be shown, just enough that we understand.



(4A)

No, this does not work. One possible scenario: `lock++;` statement on line 2 is not atomic so a thread switch can take place in between its steps. Remember that an increment statement requires reading the variable, increasing its value by 1, and writing the changed value back. So imagine this order:

1. Thread A gets to line 2, the `lock++` statement.
2. Thread A reads 0.
3. Thread A increments `lock` (value now 1).
4. Untimely thread switch: Thread B gets chosen to run.
5. Thread B gets to line 2, the `lock++` statement.
6. Thread B reads 0.
7. Thread B increments `lock` (value now 1).
8. Thread B writes 1 to `lock`.
9. Thread B evaluates the if statement on line 3 and enters the critical section.
10. Untimely thread switch back to A.
11. Thread A writes 1 to `lock`.
12. Thread A evaluates the if statement on line 3 and enters the critical section.
13. Threads A and B are now in the critical section!

(4B)

Global variables:

```
sem_t turnstile;
sem_t room_empty;
pthread_mutex_t mutex;
int readers;

void init() {
  readers = 0;
  sem_init( &turnstile, 0, 1 );
  sem_init( &room_empty, 0, 1 );
  pthread_mutex_init( &mutex, NULL );
}

void cleanup() {
  sem_destroy( &turnstile );
  sem_destroy( &room_empty );
  pthread_mutex_destroy( &mutex );
}

void* writer( void* ignore ) {
  sem_wait( &turnstile );
  sem_wait( &room_empty );
  update_data( );
  sem_post( &turnstile );
  sem_post( &room_empty );
}

void* reader( void* ignore ) {
  sem_wait( &turnstile );
  sem_post( &turnstile );
  pthread_mutex_lock( &mutex );
  readers++;
  if ( readers == 1 ) {
    sem_wait( &room_empty );
  }
  pthread_mutex_unlock( &mutex );
  read_data( );
  pthread_mutex_lock( &mutex );
  readers--;
  if ( readers == 0 ) {
    sem_post( &room_empty );
  }
  pthread_mutex_unlock( &mutex );
}
```

(4C) Joining a thread has two behaviours: (1) waiting for the joined thread to finish and (2) collecting a return value. Waiting can be simulated by use of a semaphore: simply wait on a semaphore that the thread to be joined signals on. The return value can be passed in other ways such as global variables or updating a pointer.