ASSIGNMENT 1 CMPT 300 SOLUTION SPRING 2019

1. **[36 points]** You will write a C program that will require you to practice creating and managing multiple generations of processes. In the description of the program below words in all capital letters should be replaced with the actual values of the process IDs when the line is printed. Use rand() and srand() from stdlib.h to generate random numbers needed below. Please use the printed outputs given below without editing, each of the printed outputs should begin on a new line when printed as output. Be sure to deal with potential errors.

   The parent process (the first process created when the program is started) will execute the following steps
   a) Prompt "enter the seed for the parent process ". Then read the seed. The seed will be used to determining numbers of children and waiting times.
   b) Initialize the random number generator using srand(). Then use rand to determine the number of children the process will generate (5 <= number of process <=9)
   c) Determine the process id of the running process.
   d) Print "My process ID is PARENTPID"
   e) Before each child is generated the parent process will print "MYPROCESSID is about to create a child"
   f) Create the child
   g) After each child process is generated the parent process will print "Parent MYPROCESSID has created a child with process ID CHILDPROCESSID"
   h) After all children have been created consider each child in the same order the children were created.
      i. Print "I am the parent, I am waiting for child CHILDPROCESSID to terminate"
      ii. After the child CHILDPROCESSID has terminated print "I am process MYPROCESSID. My child CHILDPROCESSID" is dead"
   i) After the last child has died the process will print "I am the parent, child CHILDPROCESSID has terminated"
   j) Sleep for 5 seconds
   k) Terminate the process

   Any child created by the child process will be referred to as a grandchild.
   Each child process will
   a) Generate a seed for the child process by adding to the seed for the parent process. Add 0 for the first process generated, 1 for the second process generated, 2 for the third process generated and so on.
   b) Reinitialize the random number generator using the child seed and srand().
   c) Print "I am a new child, my process ID is CHILDPROCESSID, my seed id CHILDSSEED"
   d) Generate a random integer number 1 <= number <= 3. This random number will determine how many children this child process will generate.
   e) Prints "I am child CHILDPROCESSID, I will have NUMCHILDREN children"
   f) Before each grandchild is generated by the child process print "I am child CHILDPROCESSID, I am about to create a child"
   g) The child process creates a grandchild.
   h) After each grandchild is generated the child process will print "I am child CHILDPROCESSID, I just created a child"

i) After all grandchildren have been created print
    1. "I am the child CHILDPROCESSID, I have NUMGRANDCHILDREN children, "
    2. "I am waiting for my children to terminate"
j) After all children have been created consider each child in the same order the children were created. After each child CHILDPROCESSID has terminated print "I am child CHILDPROCESSID. My child GRANDCHILDPROCESSID has been waited". Wait for the first process, when the first process has terminated wait for the second and so on.
k) After all of the children have been waited print "I am child CHILDPROCESSID, I am about to terminate"
l) Sleep for 5 seconds
m) Terminate the process

Each grandchild process will

a) Print "I am grandchild GRANDCHILDPROCESSID, My grandparent is PROCESSID, My parent is PARENTPROCESSID"
b) Generate a random integer number 5 <= sleep number <= 14.
c) Make the grandchild sleep for sleep number seconds
d) After the grandchild wakes up print "I am grandchild GRANDCHILDPROCESSID with parent CHILDPROCESSID, I am about to terminate"
e) Terminate the process

2. Consider a system that uses a 64-bit word. This CPU has a 1 Mbyte (1024*1024 bytes) cache memory. The size of each CPU cache slot is 512 bytes. The system has a 4GB main memory. Cache line 0 holds main RAM memory (byte) addresses 0 to 511, cache line 1 holds main memory (byte) addresses 512 to 1024, and so on. Assume that the cache initially contains the information in the first 550 cache lines of main RAM memory. Also assume that cache line 0 was loaded first, and that the other 549 cache lines were loaded in numerical sequence. After being loaded each of the cache lines was accessed several times, but the last access to cache line N occurred before the first access to cache line N+1. Further assume that the remainder of the cache (cache slots 550 …) are empty. In other words they have not been used to hold cache lines since the OS was started.

   Consider a program that makes a series of memory requests for data and/or instructions. Each request is for information stored in a particular cache line of main RAM memory. The part of the code we are considering consists of two consecutive loops. The first loop will be executed 8 times. The second loop is executed 43 times. To provide the needed instructions and data to the program, as it runs, cache lines are accessed in the following order:

   i. The code and data in the first loop is stored in cache lines numbered 400 to 950. Each time the first loop is executed there will be 5 accesses for each cache line, 5 accesses to cache line 400 then 5 accesses to cache line 401, and so on, …, finishing with 5 accesses to cache line 950.
   ii. The code and data in the second loop is stored in cache lines 4666 to 5028, and lines 16968 – 17314. Each time the second loop is executed there will be 3 accesses to cache line 4666, 3 accesses to cache line 4667, …, finishing the first group of cache lines with 3 accesses to cache line **5028,** then there will be 3 access to cache line 16968, then 3 accesses to 16969, and so on until the 3 accesses to line 17314 are complete.

   You will be asked to consider three different mapping algorithms. For each of these mapping algorithms assume the replacement algorithm is to place the new cache line in the MAPPED cache slot that was accessed least recently. The rules for the replacement algorithm are:

i. If there are MAPPED slots that are empty, choose the empty MAPPED slot with the smallest number
ii. otherwise choose the MAPPED cache slot that was last accessed the longest time ago (last accessed at the earliest time)

a) [15 points]. The system uses direct mapping of the 4GB memory of the system to the cache. Direct mapping means that if there are N cache slots in the cache memory and M cache lines in memory, cache line K in memory will map to cache slot K%N in the cache. When cache line K is loaded in the cache it must always be loaded in cache slot K%N. What are the number of cache hits, the number of cache misses and the hit ratio for the series of accesses above using direct mapping? Give a step by step explanation in your answer which includes a summary of which cache lines are placed in which cache slots in which sequence. Most of the points will be given for your explanation.

HINT: *The cache initially contains the information in the first 550 cache lines of main memory in its first 550 cache slots (Cache line 0 in cache slot 0, cache line 1 in cache slot 1, …, finishing with cache line 549 in cache slot 549).*

HINT: *First consider how many hits and misses for the first time through the first loop, then consider how many hits and misses for the remaining times through the first loop, then consider the first time through the second loop, then the subsequent times through the second loop*

.

The cache has cache size/cache slot size = 1024KB/512B = 1024KB/0.5KB = 2048 cache slots

<u>CONSIDER THE FIRST TIME THROUGH THE FIRST LOOP</u>

The cache initially contains the first 550 cache lines of memory in first 550 slots (slots 0 to 549). Therefore, lines 400 to 549 are already in the cache. Thus, the 5 accesses for each cache line for cache lines 400 to 549 are all hits. (A total of 150 * 5 = 750 hits).

Cache slots 550-2047 are empty, so each time a particular cache line in the range 550-950 is accessed the first access to that cache line is a miss because the cache line is not already in memory. The miss will cause the cache line to be loaded into an empty cache slot. Subsequent accesses to that particular cache line will be hits. Each cache line is accessed 5 times each time through the loop, so for each cache line in this range there will be 1 miss and 4 hits the first time through the first loop. (A total of 401 misses and 4*401=1604 hits). So for the first time through the code in the first loop there are a total of 750+1604=2354 hits and 401 misses.

<u>CONSIDER SUBSEQUENT TIMES THROUGH THE FIRST LOOP</u>

All cache lines needed within the first loop are still in the cache, so all accesses during the last seven passes through the first loop will be hits. Each time through the loop there are 2354+401=2755 hits
The total number of hits and misses for the first loop is 2354 + 2755*7 = 21,639 hits and 401 misses

<u>CONSIDER THE FIRST PASS THROUGH THE SECOND LOOP</u>

The cache slots for cache lines 4666 to 5028 are 4666%2048=570 to 5028%2048=932.
The cache slots for cache lines 16968 to 17314 are 16968%2048=584 to 17314%2048=930

The first time through the loop the accesses for cache lines 4666 to 5028 are all misses. Cache slots 570 to 932 were just used for instructions or code for the first loop (which filled slots 550-950 with cache lines 550 to 950). Therefore, the first access to each of the cache lines in the range 570 to 932 looking for lines 4666 to 5028 respectively is a miss. For the first block of lines in the second loop there are 363 misses and 2*363=726 hits.

Similarly, the first time through the loop the access for cache lines 16968 to 17314 are all misses. Cache slots 584 to 930 were used by cache lines 4696 to 5028 in the first half of the second loop, and are now need to have cache lines 16969 to 17314 loaded into them. Therefore, the first access for each cache line is a miss. For the second block of lines in the second loop there are 347 misses and 2*347=694 hits

So the first time through the second loop there are 726+694=1420 hits and 710 misses

<u>CONSIDER THE REMAINING 42 TIMES THROUGH THE SECOND LOOP</u>

The first block of cache lines 4666 to 5028 are stored in cache slots

4666%2048=570 to 5028%2048=932.

The second block of cache lines 16968 to 17314 are stored in cache slots

16968%2048=584 to 17314%2048=930

The second block of cache lines (see above) uses a subset (584-930) of the cache slots used by the first block of cache lines (570-932). Therefore, when we re-execute the code in the second loop each cache line in this subset of the first block of lines (4680 - 5026, slots 584 - 930) will have a miss the first time it is accessed. Similarly, when we have completed the execution of the first block of cache lines, each line in the second block of lines will have 1 miss since it has been overwritten by a cache line from the first block.

Lines 4666-4679 and 5027-5028 are loaded the first time through the loop and never overwritten. All accesses to these lines in later iterations through the loop are hits. So there are 15 + 2 additional hits each time through the loop 16 = 363-347

Each subsequent time through the second loop there are

2*347=694 misses 2*2*347+3*(363-347)= 1436 hits

Next let's add up all the hits and misses

First loop first time + Inner loop first time through outer loop + Outer loop subsequent time *2 +

2 * inner loop subsequent times

HITS: firstloop + secondloop = (21639) + (1420 + 42*1436) = 21639 + 61732 = 83371

MISSES: firstloop + secondloop = 401 + (710 + 42*694) = 30259

Hit ratio = 83371 / (112519 + 1111) = 83371 / 113630 = 0.7337

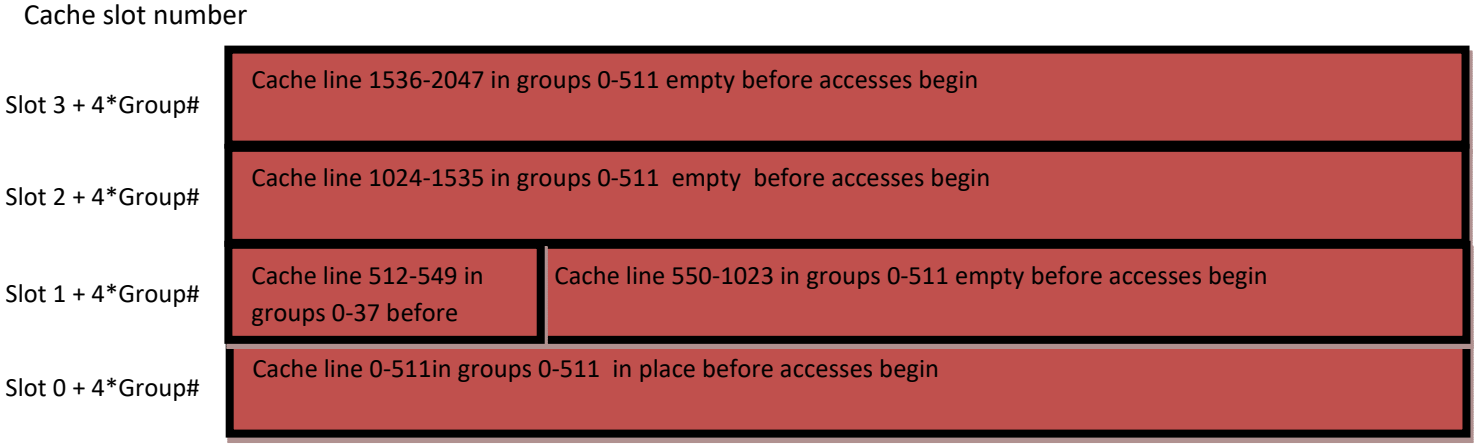
b) [12 points] Assume that all cache lines in the cache when the code begins to execute (lines 0 to 549) were loaded into the cache in numerical order using 4 way associative mapping. When we use M way set associative mapping (for this example M=4) we divide the available cache slots into sets of M slots. If there are N cache slots in the cache memory we can divide them into P groups of M slots (P*M=N).

Cache line K in memory will map to any cache slot in group Q (0<=Q<P) if K%P = Q. When the cache is empty cache line 0 in stored cache slot 0 (first slot of group 0), cache line 1 is stored in cache slot 4 (first slot of group 1), cache line 2 is stored in cache slot 8 (first slot of group 2), and so on. Then cache line 512 is stored in cache slot 1 (second slot of group 0), cache line 513 is stored in cache slot 5 (second slot of group 1) and so on. When all slots in a group are full, then the slot in the group that was last accessed the longest time ago will be replaced. What is the hit ratio for the series of accesses above using 4 way set associative mapping of the cache? Give a step by step explanation in your answer which includes a summary of which cache lines are placed in which cache slots at what times

The cache has cache size/cache slot size = 1MB/512B = 2048 cache slots N=2048
Using 4 way associative mapping there will be P=512 groups of 4 slots.
The cache initially contains cache lines 0 to 549. These cache lines were loaded using 4 way associative mapping into an empty cache.

For lines 400 to 511	***400 % P = 400 % 512 = 400***	***511 % 512 = 511***
For lines 512 to 549	***512 % 512 = 0***	***549 % 512 = 37***

The cache initially contains cache lines distributed as shown in the diagram below.



<u>CONSIDER THE FIRST PASS THROUGH THE FIRST LOOP</u>
The mapping algorithm gives

For lines 400 to 511	***400 % P = 400 % 512 = 400***	***511 % 512 = 511***
For lines 512 to 549	***512 % 512 = 0***	***549 % 512 = 37***
For lines 550 to 950	***550 % 512 = 38***	***950 % 512 = 438***

- ***The first time through the first loop, all accesses to cache lines 400 to 549 are all hits. (A total of 150* 5 = 750 hits) because they are already in the cache.***
- ***The mapping algorithm indicates that cache lines 550 to 950 should be placed in groups 38 to 438 respectively. The replacement algorithm indicates they should be placed in the lowest numbered empty cache slot in their group (if any slots are empty) . For groups 38 to 438 slots 1, 2 and 3 are empty, so we will load cache lines 550 to 950 into slot one of groups 38 to 438 respectively.***
- ***The first accesses for cache lines 550 to 950 are all misses because slots 38 to 438 are empty.***

438-38+1=401 misses.

- ***The second through fourth accesses inclusive will all be hits because the miss on the first access will have caused the cache line to be loaded into the cache slot.***
- ***For the first time through the first loop there are 401 misses and 750 + 4*401 = 2354 hits***

<u>CONSIDER THE FIRST PASS THROUGH THE SECOND LOOP</u>

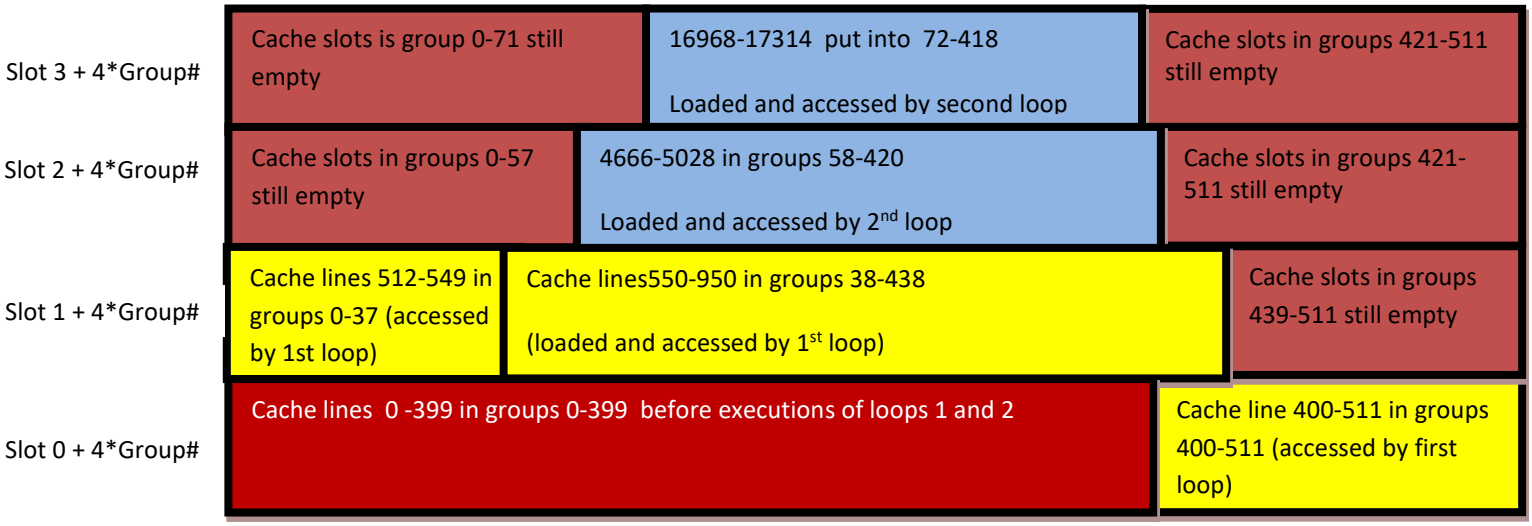
The mapping algorithm gives:

- ***The cache slots for cache lines 4666 to 5028 are 4666%512=58 to 5028%512=420.***
- ***The cache slots for cache lines 16968 to 17314 are 16968%2048=72 to 17314%2048=418***

Cache lines 4666-5028 are accessed next, as we start stepping through the code inside the second loop. Cache lines 4666-5028 are loaded into slot 2 of the groups 58-420 because slot 0 and 1 are of these groups are already in use by loop 1.
Cache lines 16968 to 17314 are accessed next as we continue stepping through the code inside the second loop. Cache lines 16968 to 17314 are loaded into the first available slots of groups 72 to 418. The first and second slot (slot 1) in each group contains data/code from the execution of the first loop, the third slots (slot 2) in each group contains the cache lines loaded for the first block of code inside the second loop. Slot 3 of all these groups is available for cache lines 16968 to 17314.

The first time through the second loop there are
(5028-4666+1) + (17314-16968+1) =363+347=710 misses and 2*710=1420 hits.
For the subsequent times through the second loop all accesses are hits because all required cache lines are stored in the cache.
So for each remaining execution of the second loop there are 3*710=2130 hits.

HITS: firstloop + secondloop = (21639) + (1420 + 42*2130) = 21639 + 93010 = 112519
MISSES: firstloop + secondloop = 401 + 710 = 1111
Hit ratio = 112519 / (112519 + 1111) = 112519 / 113630 = 0.9902



3. Three CPU bound jobs A through C and one I/O bound job D arrive at a computer center at the times shown in the table below. All times in the table are in ms. Ignore context switches. The system follows the following rules:
   a. In the priority column higher numbers indicate higher priorities
   b. The I/0 bound job runs for 3ms of CPU time then prints a value, this pair of actions repeat until the job finishes.
   c. The print hardware takes 2ms to print one value.
   d. The quanta for an I/O bound jobs is cumulative. Therefore:
      i. When an I/O bound job leaves the CPU the remaining time in its quanta is recorded
      ii. When an I/O bound job re-enters the CPU after completing a write the length of the quanta given to it is the remaining time in the quanta recorded before the job left the CPU to enter the print queue
   e. When a job leaves the print queue it is placed in the ready queue
      i. If the job is starting a new quanta it is placed at the end of the ready queue
      ii. If it is continuing an unfinished quanta it is placed at the front of the ready queue
   f. When a pre-empted job leaves the CPU it is placed at the front of the ready queue
   g. If two actions take place at the "same" time the order they are completed is
      i. place a newly arrived process in the queue
      ii. check the queues to see which process is loaded into the CPU next
      iii. place a process leaving the CPU into the ready queue or the print queue
      iv. place a process leaving the print queue into the ready queue
   h. Each process is allowed a quantum of 6 ms of CPU. At the end of the quantum the process will be placed at the end of the appropriate ready queue
      i. For CPU bound jobs this quantum will be used in one visit to the CPU.
      ii. For I/O bound jobs this quantum may be distributed over several visits to the CPU.

Process	Arrival time (ms)	Running Time (ms)	Priority
A	12	28	4
B	19	34	7
C	2	36	4
D	7	9	7

Determine the turnaround time for each process and the average turnaround time for all processes for each of the following scheduling algorithms. Explain how you arrived at your answers. In particular, your explanation should include a COMPLETE list of steps showing the order in which the processes execute. The duration of each step, and the elapsed time at the end of each step should be shown. The description may be presented as a bulleted list of steps OR a table, with one column per process and an entry for each step.

a) **[13 points]** round-robin scheduling (no priorities)
b) **[13 points]** round-robin scheduling (with pre-emptive priorities). ***When a job arrives in the high priority queue, the running job is checked to see if it has a lower priority. If the running job has a lower priority it will be immediately replaced with the higher priority job***
c) **[6 points]** first come first served (with non pre-emptive priorities).

NOTE: NO TIME SHARING (not round robin)

SOLUTION

a) [13 points] Table shows completion time of each time slice (under the process that ran that time slice) Round-robin no priorities

C arrives at 2 (takes 36)
D arrives at 7 (takes 9)
A Arrives at 12 (takes 28)
B arrives at 19 (takes 34)
Completion times A=99 B=109 C=89 D=47 avg = 86
Turnaround times A=87 B=90 C=87 D=40 avg = 76
Process order: C, D, C, D, A, C, B, D, A, D, C, B, A, C, B, A, C, B, A, B, B

Actions	*printqueue*	*queue*	*Running*	*Running time*	*Time used A*	*Time used B*	*Time Used C*	*Time used D*
C arrives				*2*				
C runs			*C*	*2 to 7*	*0*	*0*	*5*	*0*
D arrives				*7*	*0*	*0*	*5*	*0*
D enters queue		*D*	*C*	*7*	*0*	*0*	*5*	*0*
C finishes quanta		*D*	*C*	*7 to 8*	*0*	*0*	*6*	*0*
D enters the CPU			*D*	*8*	*0*	*0*	*0*	*0*
C enters the queue		*C*	*D*	*8*	*0*	*0*	*0*	*0*
D runs to IO		*C*	*D*	*8 to 11*	*0*	*0*	*6*	*3*
C enters CPU			*C*	*11*	*0*	*0*	*6*	*3*
D enters IO queue	*D*		*C*	*11*	*0*	*0*	*6*	*3*
C runs till 17ms			*C*	*11 to 12*	*0*	*0*	*7*	*3*
A arrives				*12*	*0*	*0*	*7*	*3*
A enters queue	*D*	*A*	*C*	*12*	*0*	*0*	*7*	*3*
C continues running	*D*	*A*	*C*	*12 to 13*	*0*	*0*	*8*	*3*
D leaves IO queue		*A,D*	*C*	*13*	*0*	*0*	*8*	*3*
C finishes quanta		*A,D*	*C*	*13 to 17*	*0*	*0*	*12*	*3*
D enters the CPU		*A*	*D*	*17*	*0*	*0*	*12*	*3*

C enters the queue		C,A	D	17	0	0	12	3
D runs till B arrives		C.A	D	17 to 19	0	0	12	5
B arrives				19	0	0	12	5
B enters queue		B,C,A	D	19	0	0	12	5
D runs to IO		B,C,A	D	19 to 20	0	0	12	6
A enters CPU		B,C	A	20	0	0	12	6
D enters IO queue	D	B,C	A	20	0	0	12	6
A runs til 26ms	D	B,C	A	20 to 22	2	0	12	6
D leaves IO queue (quanta done)		D,B,C	A	22	2	0	12	6
A runs to end of quanta		D,B,C	A	22 to 26	6	0	12	6
C to CPU, A to queue		A,D,B	C	26	6	0	12	6
C runs 1 quanta		A,D,B	C	26 to 32	6	0	18	6
B to CPU, C to queue		C,A,D	B	32	6	0	18	6
B runs 1 quanta		C,A,D	B	32 to 38	6	6	18	6
D to CPU, B to queue		B,C,A	D	38	6	6	18	6
D runs to IO		B,C,A	D	38 to 41	6	6	18	9
A to CPU, D to IO queue	D	B,C	A	41	6	6	18	9
A runs till 43 ms	D	B,C	A	41 to 43	8	6	18	9
D enters IO queue		B,C,D	A	43	8	6	18	9
A finishes quanta		B,C,D	A	43 to 47	12	6	18	9
D to CPU, A to queue		A,B,C	D	47	12	6	18	9
D done		A,B,C		47	12	6	18	9
C to CPU, runs for quanta		A,B	C	47 to 53	12	6	24	-----
B to CPU, C to queue, B runs for quanta		C,A	B	53 to 59	12	12	24	-----
A runs for 1 quanta		B,C	A	59 to 65	18	12	24	-----

C runs for 1 quanta		*A,B*	*C*	*65 to 71*	*18*	*12*	*30*	*-----*
B runs for 1 quanta		*C,A*	*B*	*71 to 77*	*18*	*18*	*30*	*-----*
A runs for 1 quata		*B,C*	*A*	*77 to 83*	*24*	*18*	*30*	*-----*
C runs for 1 quanta		*A,B*	*C*	*83 to 89*	*24*	*18*	*36*	*-----*
Cdone				*89*			*-----*	*-----*
B runs for 1 quanta		*A*	*B*	*89 to 95*	*24*	*24*	*-----*	*-----*
A runs for 1 quata		*B*	*A*	*95 to 99*	*28*	*24*	*-----*	*-----*
Adone				*99*	*28*		*-----*	*-----*
			B	*99 to*	*-----*	*30*	*-----*	*-----*
			B	*105 to*	*-----*	*34*	*-----*	*-----*
Bdone				*109*	*-----*	*34*	*-----*	*-----*

b) [15 points] round-robin scheduling (pre-emptive priorities). When a job arrives in the high priority queue will cause the running job to be checked to see if it has a lower priority. If the running job has a lower priority it will be immediately replaced with the higher priority job
When pre-emptive priorities are used new jobs arriving in the queue pre-empt jobs of lower priority that are already running. Assume that when process X finishes the next process to run, process Y, is chosen from the highest priority queue containing processes, then the process X is placed at the end of the appropriate queue.

Table shows completion time of each time slice (under the process that ran that time slice)

Arrivals	*Print queue*	*Low priority queue*	*High priority queue*	*Running*	*Running time*	*Time used A (CPU)*	*Time used B (CPU)*	*Time Used C (CPU)*	*Time used D (CPU)*
C arrives					*2*				
C runs				*C*	*2 to 7*	*0*	*0*	*5*	*0*
D arrives					*7*	*0*	*0*	*5*	*0*
D enters queue			*D*	*C*	*7*	*0*	*0*	*5*	*0*
Pre-empt, D enters CPU, C enters low queue, 1ms remains in C's quanta		*C*		*D*	*7*	*0*	*0*	*5*	*0*
D runs to IO		*C*		*D*	*7 to 10*	*0*	*0*	*5*	*3*
C enters CPU, D enters print queue	*D*			*C*	*10*	*0*	*0*	*5*	*3*
C runs remaining 1ms of quanta	*D*			*C*	*10 to 11*	*0*	*0*	*6*	*3*
C leaves CPU	*D*	*C*			*11*				
C starts another quanta	*D*			*C*	*11*	*0*	*0*	*6*	*3*
C runs	*D*			*C*	*11-12*	*0*	*0*	*7*	*3*
A arrives					*12*	*0*	*0*	*7*	*3*
A enters queue		*A*		*C*	*12*	*0*	*0*	*7*	*3*
D enters ready queue			*D*	*C*	*12*				
Pre-empt, D enters CPU, C enters low queue, 5ms remains in C's quanta		*A,C*		*D*	*12*	*0*	*0*	*7*	*3*
Run D to IO		*A,C*		*D*	*12 to 15*	*0*	*0*	*7*	*6*

C moves to CPU *D moves to IO queue*	*D*	*A*		*C*	*15*	*0*	*0*	*7*	*6*
D completes IO, C runs	*D*	*A*		*C*	*15 to 17*	*0*	*0*	*9*	*6*
D enters ready queue		*A*	*D*	*C*	*17*				
Pre-empt, D enters CPU, *C enters low queue, 3ms* *remains in C's quanta*		*A,C*		*D*	*17*	*0*	*0*	*9*	*6*
Run D to IO		*A,C*		*D*	*17 to 19*	*0*	*0*	*9*	*8*
B arrives		*A,C*			*19*	*0*	*0*	*9*	*8*
B enters queue		*A,C*	*B*	*D*	*19*	*0*	*0*	*9*	*8*
Run D		*A,C*	*B*	*D*	*19 to 20*	*0*	*0*	*9*	*9*
B moves to CPU *D moves to IO queue*	*D*	*A,C*		*B*	*20*	*0*	*0*	*9*	*9*
D complets IO, B runs	*D*	*A,C*		*B*	*20 to 22*	*0*	*2*	*9*	*9*
D goes to queue		*A,C*	*D*	*B*	*22*	*0*	*2*	*9*	*9*
B runs to end of quanta		*A,C*	*D*	*B*	*22 to 26*	*0*	*6*	*9*	*9*
D to CPU, B to queue		*A,C*	*B*	*D*	*26*	*0*	*6*	*9*	*9*
D done		*A,C*	*B*		*26*	*0*	*6*	*9*	*9*
B to CPU		*A,C*		*B*	*25 to 26*	*0*	*6*	*9*	-----
B runs		*A,C*		*B*	*26 to 32*	*0*	*12*	*9*	-----
B to queue, B to CPU,		*A,C*		*B*	*32*	*0*	*12*	*9*	-----
B runs for 1 quanta		*A,C*		*B*	*32 to 38*	*0*	*18*	*9*	-----
B to queue, B to CPU,		*A,C*		*B*	*38*	*0*	*18*	*9*	-----
B runs for 1 quanta		*A,C*		*B*	*38 to 44*	*0*	*24*	*9*	-----
B to queue, B to CPU,		*A,C*		*B*	*44*	*0*	*24*	*9*	-----
B runs for 1 quanta		*A,C*		*B*	*44 to 50*	*0*	*30*	*9*	-----
B to queue, B to CPU,		*A,C*		*B*	*50*	*0*	*30*	*9*	-----
B runs to end of process		*A,C*		*B*	*50 to 54*	*0*	*34*	*9*	-----
B done					*54*	*0*	*34*	*9*	-----
C to CPU, C runs *remaining 3 ms in quanta*		*A*		*C*	*54 to 57*	*0*	-----	*12*	-----

A to CPU, C to queue *A runs 1 quanta*		*C*		*A*	*57 to 63*	*6*	-----	*12*	-----
C runs 1 quanta		*A*		*C*	*63 to 69*	*6*	-----	*18*	-----
A runs 1 quanta		*C*		*A*	*69 to 75*	*12*	-----	*18*	-----
C runs 1 quanta		*A*		*C*	*75 to 81*	*12*	-----	*24*	-----
A runs 1 quanta		*C*		*A*	*81 to 87*	*18*	-----	*24*	-----
C runs 1 quanta		*A*		*C*	*87 to 93*	*18*	-----	*30*	-----
A runs 1 quanta		*C*		*A*	*93 to 99*	*24*	-----	*30*	-----
C runs to completion		*A*		*C*	*99to 105*	*24*	-----	*36*	-----
C complete					*105*	*24*	-----	*36*	-----
A runs to completion				*C*	*105to109*	*28*	-----	-----	-----
A done					*109*	*24*	-----	-----	-----

C arrives at 2 (takes 36)

D arrives at 7 (takes 9)

A Arrives at 12 (takes 28)

B arrives at 19 (takes 34)

Completion times A=109 B=54 C=105 D=26 avg = 73.5

Turnaround times A=97 B=35 C=103 D=19 avg = 63.5

Process order: C D C C D C D B D B B B B B C A C A C A C A C C

c) [9 points] non pre-emptive priority based scheduling (no time sharing)

Arrivals	*printqueue*	*Low priority queue*	*High priority queue*	*running*	*Running time*	*Time used A*	*Time used B*	*Time Used C*	*Time used D*
C					*2*				
				C	*2 to 7*	*0*	*0*	*5*	*0*
D					*7*	*0*	*0*	*5*	*0*
			D	*C*	*7 to 12*	*0*	*0*	*10*	*0*
A					*12*	*0*	*0*	*10*	*0*
		A	*D*	*C*	*12 to 19*	*0*	*0*	*17*	*0*
B					*19*	*0*	*0*	*17*	*0*
		A	*B,D*	*C*	*19 to 38*	*0*	*0*	*36*	*0*
		A	*B*	*D*	*38 to 41*	*0*	*0*		*3*
	D	*A*		*B*	*41 to 43*	*0*	*2*		*3*
		A	*D*	*B*	*43 to 75*	*0*	*34*		*3*
		A		*D*	*75 to 78*	*0*			*6*
	D			*A*	*78 to 80*	*2*			*6*
			D	*A*	*80-106*	*28*			*6*
				D	*106-111*				*9*
	D				*109-111*				*9*

Process C D B A
38 111 75 106

C arrives at 2 (takes 36)
D arrives at 7 (takes 9)
B arrives at 19 (takes 34)
A arrives at 12 (takes 28)

Completion times A=106 B=75 C=38 D=111 avg= 82.5
Turnaround times: A=94 B=56 C=36 D=104 avg=72.5
Process order: C, D, B, D, A, D