CS4411 Operating Systems Final Solutions
Spring 2019

1. **[20 points]** Five processes *A*, *B*, *C*, *D* and *E* arrived in this order at the same time with the following CPU burst and priority values. A smaller value means a higher priority.

	CPU Burst	*Priority*
A	3	3
B	7	5
C	5	1
D	2	4
E	6	2

Fill the entries of the following table with waiting time and average turn around time for each indicated scheduling policy. Ignore context switching overhead. Fill out the following table and provide a detailed elaboration. **Without a detailed elaboration, you risk to receive a very low or even 0 points.**

	Waiting Time					*Average*	
Scheduling Policy	*A*	*B*	*C*	*D*	*E*	*Waiting Time*	*Turn Around Time*
First-Come-First-Served							
Non-Preemptive Shortest-Job First							
Priority							
Round-Robin (time quantum=2)							

Answer: For each process, the following has the FCFS's start time, CPU burst (given), end time, waiting time and turnaround time:

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
A	0	3	3	0	3
B	3	7	10	3	10
C	10	5	15	10	15
D	15	2	17	15	17
E	17	6	23	17	23

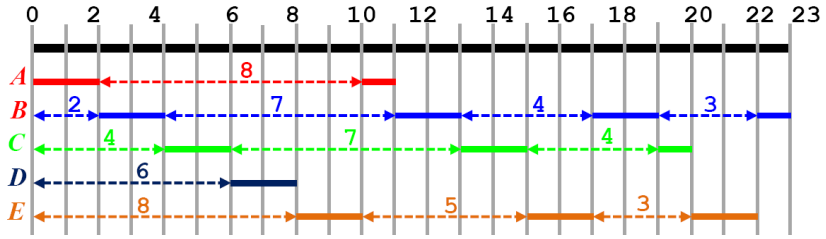
For each process, the following shows SJN's start time, CPU burst (given), end time, waiting time and turnaround time. Note that the order of processes is based on the length of CPU bursts, from the shortest CPU burst to the longest CPU burst.

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
D	0	2	2	0	2
A	2	3	5	2	5
C	5	5	10	5	10
E	10	6	16	10	16
B	16	7	23	16	23

For each process, the following is the Priority's start time, CPU burst (given), end time, waiting time and turnaround time. Note that the order of processes is based on the priority of each process, from highest to lowest.

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
C	0	5	5	0	5
E	5	6	11	5	11
A	11	3	14	11	14
D	14	2	16	14	16
B	16	7	23	16	23

The following figure shows the RR scheduling of these five processes. It is not difficult to see that A, B, C, D and E starts at time 0, 2, 4, 6 and 8, respectively, and has waiting times 8, $16 = 2+7+4+3$, $15 = 4+7+4$, 6, and $16 = 8+5+3$, respectively.



From this diagram, we are able to generate the following table:

	Start	*Burst*	*End*	*Waiting*	*Turnaround*
A	0	3	11	8	11
B	2	7	23	16	23
C	4	5	20	15	20
D	6	2	8	6	8
E	8	6	22	16	22

The following table is a summary of our findings:

	Waiting Time					*Average*	
Scheduling Policy	*A*	*B*	*C*	*D*	*E*	*Waiting Time*	*Turn Around Time*
First-Come-First-Served	0	3	10	15	17	9.0=45/5	13.6 = 68/5
Non-Preemptive Shortest-Job First	2	16	5	0	10	6.6 = 33/5	11.2 = 56/5
Priority	11	16	0	14	5	9.2 = 46/5	13.8 = 69/5
Round-Robin (time quantum=2)	8	16	15	6	16	12.2 = 61/5	16.8 = 84/5

2. **[10 points]** Consider the following snapshot of a system:

	Allocation				*Max*				*Need*				*Available*			
	U	*V*	*W*	*X*	*U*	*V*	*W*	*X*	*U*	*V*	*W*	*X*	*U*	*V*	*W*	*X*
A	1	1	0	0	2	3	1	1	1	2	1	1	1	2	0	0
B	1	1	1	1	1	5	1	4	0	4	0	3				
C	0	1	0	1	2	4	0	3	2	3	0	2				
D	2	0	2	0	5	4	2	4	3	4	0	4				
E	0	0	1	2	0	2	1	2	0	2	0	0				

Is this system in a safe state? **Show your computation step-by-step and your safe sequence explicitly; otherwise, you will receive no credit.**

Answer: Because E's $Need = [0,2,0,0] \leq Available = [1,2,0,0]$, we run E and reclaim its allocation $[0,0,1,2]$. Therefore, the new $Available = E$'s $Allocation[0,0,1,2] + Available[1,2,0,0] = [1,2,1,2]$

Because A's $Need = [1,2,1,1] \leq Available = [1,2,1,2]$, we run A and reclaim its allocation $[1,1,0,0]$. Hence, the new $Available = A$'s $Allocation[1,1,0,0] + Available[1,2,1,2] = [2,3,1,2]$.

Then, we run C and reclaim its $Allocation = [0,1,0,1]$, and we have new $Available = C$'s $Allocation[0,1,0,1] + Allocation[2,3,1,2] = [2,4,1,3]$.

Now, because B's $Need = [0,4,0,3] \leq Available = [2,4,1,3]$, we run B and reclaim its allocation $[1,1,1,1]$. The new $Allocation = B$'s $Allocation[1,1,1,1] + Available[2,4,1,3] = [3,5,2,4]$.

This $Available = [3,5,2,4]$ is larger than the remaining process D's $Need = [3,4,0,4]$. Therefore, we run D.

Because all process can be run this way and every process finishes. we have a safe sequence $< E,A,C,B,D >$. The system is safe. ∎

3. **[10 points]** A paging system uses 16-bit address and 4K pages. The following shows the page tables of two running processes, Process 1 and Process 2. Translate the logical addresses in the table below to their corresponding physical addresses, and fill the table entries with your answers.

	Process 1
0	6
1	3
2	0
3	2

	Process 2
0	7
1	1
2	4
3	5

Process	*Address*	*Page #*	*Offset*	*Physical Address*
Process 1	16,000			
Process 2	9,000			

Answer: Process 1 has a virtual address 16,000. The page number and offset are the quotient and remainder of dividing 16,000 by 4K = 4096, respectively. Thus, we have "page number" = 16000/4096 = 3 and "offset" = 16000 % 4096 = 3712. From the page table of Process 1, the corresponding page frame of page 3 is page frame 2. The physical address is simply 11,904$= 2 \times 4096 + 3712$.

Process 2 has a virtual address 9000. We have "page number" = 9000/4096 = 2 and "offset" = 9000 % 4096 = 808. Because page 2 of Process 2 is in page frame 4, the corresponding physical address of 9000 is 17,192$= 4 \times 4096 + 808$.

Process	*Address*	*Page #*	*Offset*	*Physical Address*
Process 1	16,000	3	3712	11,904
Process 2	9,000	2	808	17,192

∎

4. **[25 points]** Let the page trace be 3, 1, 2, 4, 3, 2, 5, 2, 1, 3, 2, 4. The computer on which this page trace is run has three page frames. Run this page trace with the FIFO, LRU and MIN the Optimal algorithms. Initially, the memory is empty. Fill in the tables with the content *after* each reference and compute the number of page faults, miss ratio and hit ratio. **You should follow the stack algorithm convention for the LRU, MIN and Working Set.**

(a) **[5 points]** FIFO algorithm:

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory												
Page faults												
Miss ratio												
Hit ratio												

(b) **[5 points]** LRU algorithm:

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory												
Page faults												
Miss ratio												
Hit ratio												

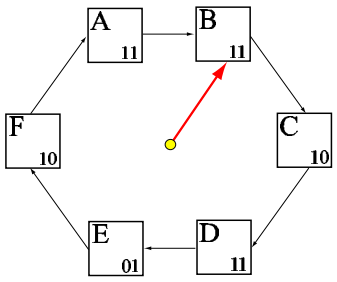
(c) **[5 points]** The MIN Optimal algorithm:

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory												
Page faults												
Miss ratio												
Hit ratio												

(d) **[5 points]** Working set of window size $\theta = 3$:

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory												

(e) **[5 points]** Suppose the CLOCK page-replacement algorithm is used to approximate the LRU algorithm. Currently a system has six page frames with RM (Reference/Modified) bits as shown below (*i.e.*, the first and second bits are the reference and modified bits, respectively):



If the clock pointer is at the page frame as shown and a page fault occurs, after running the CLOCK algorithm, what are the new RM bits? Which page frame will be selected as a victim? Is a page-out required in this case? Why? Which page frame will be selected if a new page fault occurs immediately?

Answer: In the following, pages that cause a page fault are shown in boldface with *. Note that pages in memory for LRU, MIN and Working Set must be shown in the stack algorithm convention.

(a) **[5 points]** FIFO algorithm: In the following table, pages are ordered from new to old based on the time a page is loaded into memory.

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory	**3***	**1***	**2***	**4***	**3***	3	**5***	**2***	**1***	**3***	3	**4***
		3	1	2	4	4	3	5	2	1	1	3
			3	1	2	2	4	3	5	2	2	1
Page faults	10											
Miss ratio	83.3%											
Hit ratio	16.7%											

(b) **[5 points]** LRU algorithm:

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory	**3***	**1***	**2***	**4***	**3***	2	**5***	2	**1***	**3***	2	**4***
		3	1	2	4	3	2	5	2	1	3	2
			3	1	2	4	3	3	5	2	1	3
Page faults	9											
Miss ratio	75%											
Hit ratio	25%											

(c) **[5 points]** The MIN Optimal algorithm:

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory	**3***	**1***	**2***	**4***	3	2	**5***	2	**1***	3	2	**4***
		3	3	3	2	3	2	5	3	1	3	2
			1	2	4	4	3	3	2	2	1	3
Page faults	7											
Miss ratio	58.3%											
Hit ratio	41.7%											

(d) **[5 points]** Working set of window size $\theta = 3$:

Page	3	1	2	4	3	2	5	2	1	3	2	4
Memory	**3***	**1***	**2***	**4***	**3***	2	**5***	2	**1***	**3***	2	**4***
		3	1	2	4	3	2	5	2	1	3	2
			3	1	2	4	3		5	2	1	3

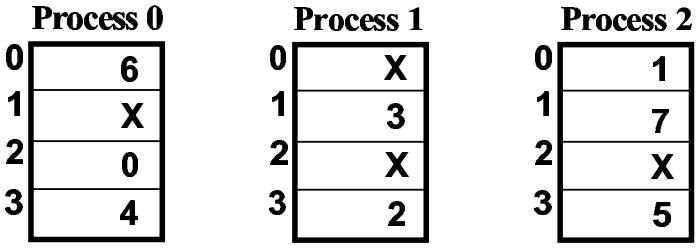
(e) **[5 points]** In the CLOCK algorithm, we scan each page frame and examine its RM bits. If the referenced bit is set, then reset it and move to the next page frame. If the referenced bit is not set, this is the page frame to be evicted.

- Because *B*'s RM = 11, it will be reset to 01.
- Because *C*'s RM = 10, it will be reset to 00.
- Because *D*'s RM = 11, it will be reset to 01.
- Because *E*'s RM = 01, it is the victim. Because its modified bit is set, a page-out is needed.

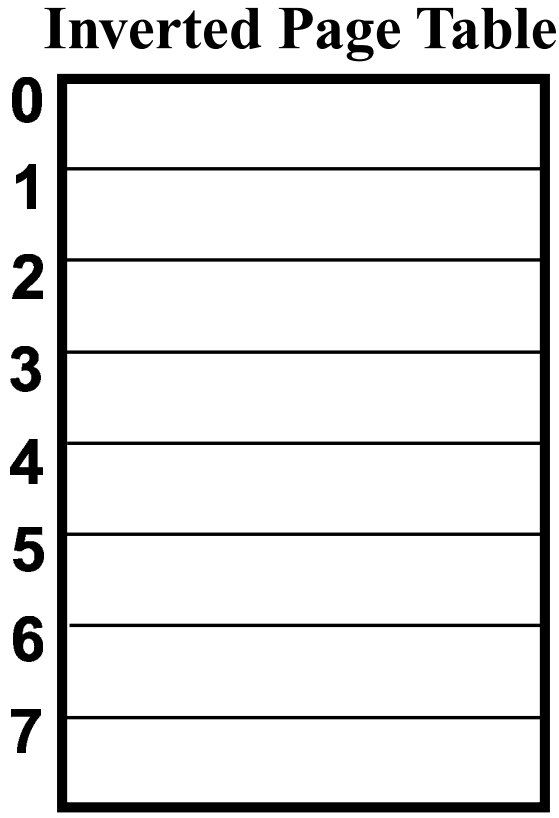
If a page fault occurs, *F*'s RM is reset to 00, *A*'s RM bit is reset to 01, and *B*'s RM bit was reset to 01 with a referenced bit 0. Therefore, *B* is the next victim and a page-out is required.

■

5. **[10 points]** A system has 8 page frames and each process has a page table of four entries in which a "X" indicates the corresponding page is not in physical memory. Suppose a snapshot of the page tables at certain moment is shown below.



Convert the content of these three page tables to an inverted page table below. Fill out the following table and provide a detailed elaboration. **Without a detailed elaboration, you risk to receive a very low or even 0 points.**



Answer: Each entry in an inverted page table has two fields: the owner (*i.e.*, process ID) and the page number of the owner. From the two given page tables, the corresponding inverted page table is

	Process ID	*Page Number*
0	0	2
1	2	0
2	1	3
3	1	1
4	0	3
5	2	3
6	0	0
7	2	1

■

6. **[10 points]** Consider the two-dimensional array `A`:

```
int A[][] = new int[100][100];
```

where `A[0][0]` is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0. For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement and assuming that page frame 1 contains the process and the other two are initially empty?

(a)

```
for (int j = 0; j < 100; j++)
   for (int i = 0; i < 100; i++)
      A[i][j] = 0;
```

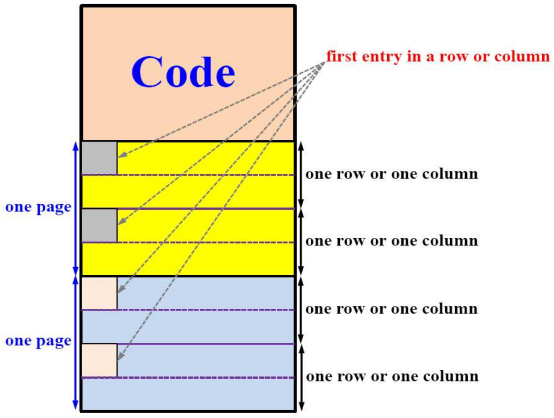
(b)

```
for (int i = 0; i < 100; i++)
   for (int j = 0; j < 100; j++)
      A[i][j] = 0;
```

Fill out the following table and provide a detailed elaboration. **Without a detailed elaboration, you risk to receive a very low or even 0 points.**

Problem	*(a)*	*(b)*
Answer		

Answer: The system has three page frames for this process. The first is for code and the second and third are for the array. Note that these three page frames do not have to be consecutive, even though the diagram below shows they are. The problem statement does not state the page size unit (*i.e.*, byte or something else) and the size of an `int`. For simplicity, we just assume each unit of the size 200 page fits an `int`. In this case, each page can fit two rows or two columns, depending on how the compiler stores a 2-dimensional array. If the compiler produces all entries of a row consecutively (*i.e.*, row-major), then each page contains two rows. Otherwise, the compiler stores all entries of a column consecutively (*i.e.*, column-major), then each page contains two columns. Because the code is in C style, the array is stored row-by-row (*i.e.*, row-major).



Because each entry in the array is only visited once, FIFO and LRU have no difference. In Part (a), the initialization goes column-by-column. Therefore, when a page is loaded into memory, it is only access twice, once per row. Refer to the diagram above for the details. Because the two available page frames are initially empty, once the initialization procedure starts, the first page is loaded in and accessed twice for row 0 and row 1. When the initialization goes to row 2, a page fault occurs and the second page is loaded, which is accessed twice for row 2 and row 3. In this way, just for column 0 for every two rows there is a page fault. To complete initialization for column 0, there will be 50 page faults. Because we have 100 columns, the number of page faults is 5,000$= 50 \times 100$.

Problem	*(a)*	*(b)*
Answer	5,000	50

For Part (b), the initialization is done row-by-row. Consequently, we have one page faults for every two rows. Because we have 100 rows, the total number of page faults is $50 = 100/2$.

If you interpret the size of 200 as 200 bytes and each `int` requires 4 bytes, then each row has 400 bytes (*i.e.*, 2 pages) and there are $200 = 2 \times 100$ pages for matrix `A[ ][ ]`. For Part (b), `A[ ][ ]` is initialized row-by-row and, as a result, initializing each row generates 2 page faults and the total number of page faults is $200 = 2 \times 100$. For Part (a), `A[ ][ ]` is initialized column-by-column, and, each access to a row will cause a page fault because two page frames can only hold one row. Consequently, for each column there will be 100 page faults. Because there are 100 columns, the total number of page faults is 10,000$= 100 \times 100$.

Problem	*(a)*	*(b)*
Answer	10,000	200

7. **[15 points]** Consider a file currently consisting of 100 blocks. Assume that the file-control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow at the beginning but there is room to grows at the end. Also assume that the block information to be added is stored in memory.

   (a) The block is added at the beginning.

   (b) The block is added in the middle.

   (c) The block is added at the end.

   (d) The block is removed from the beginning.

   (e) The block is removed from the middle.

   (f) The block is removed from the end.

   Fill out the following table and provide a detailed elaboration. **Without a detailed elaboration, you risk to receive a very low or even 0 points.**

	Contiguous	*Linked*	*Indexed*
(a)			
(b)			
(c)			
(d)			
(e)			
(f)			

   Answer: The correct answers are shown in the table below:

	Contiguous	*Linked*	*Indexed*
(a)	201	1	1
(b)	101	52	1
(c)	1	3	1
(d)	198	1	0
(e)	98	52	0
(f)	0	100	0

   (a) The block is added at the beginning.

      i. **Contiguous**: All 100 blocks have to be shifted one block. Because each shift requires one read and one write, $200 = 100 \times 2$ disk I/O operations are needed. Then, the new block is written to the first location, which requires one write. As a result, the number of disk I/O operations is $201 = 2 \times 100 + 1$.

      ii. **Linked**: The address to the first block is found in the directory. Thus, this address is add to the address field of the new block, which is written to an available location. Hence, one 1 disk operation is needed.

iii. **Indexed**: The pointers in the index block are shifted to make the first position available so that the new block could be recorded there. Therefore, only 1 disk I/O is needed to write the new block to disk.

(b) The block is added in the middle. This is very similar to adding at the beginning.

i. **Contiguous**: The middle block can be computed from the directory because blocks are allocated in a contiguous way. Then, the 50 blocks in the second half are moved one block, which requires 50 reads and 50 writes. Finally, the new block is written to the original 50th position, and the number of disk I/O operations is $101 = 2 \times 50 + 1$.

ii. **Linked**: We need 50 reads to retrieve the address of the 51th block. The address of the 51th block is added to the address of the new block, which is written to disk with 1 disk I/O operation. The address of the original 50th block has to be updated with the address of the new block. Therefore, the number of disk I/O operations is $52 = (50 + 1) + 1$.

iii. **Indexed**: This is the same as the previous case, and only 1 disk I/O is needed to write the new block to disk.

(c) The block is added at the end.

i. **Contiguous**: One disk I/O operation is needed to write the new block to the end of the original contiguous allocation provided that there is a free block available. Otherwise, compaction is needed.

ii. **Linked**: First we need to retrieve the last block from directory (1 read). Its address field is updated with the address of the new block, and written back to disk (1 write). One more write is needed to write the new block to disk. Therefore, the number of disk I/O operation is $3 = 1 + 1 + 1$.

iii. **Indexed**: This is the same as the previous case, and only 1 disk I/O is needed to write the new block to disk.

(d) The block is removed from the beginning.

i. **Contiguous**: The remaining 99 blocks are moved forward and each move requires 1 read and 1 write, The total number of disk I/O operations is $198 = 2 \times 99$.

ii. **Linked**: Read the first block to find the second one. The second block's address is used to update the directory of this file. Hence, only 1 disk I/O operation is needed.

iii. **Indexed**: Shifting the addresses stored in the index block one position upward does not need any disk I/O operation.

(e) The block is removed from the middle.

i. **Contiguous**: The remaining 49 blocks are moved forward and each move requires 1 read and 1 write, The total number of disk I/O operations is $98 = 2 \times 49$.

ii. **Linked**: This is the same as inserting in the middle. The number of disk I/O operations is 52.

iii. **Indexed**: Shifting the addresses stored in the index one position upward does not need any disk I/O operation.

(f) The block is removed from the end.

i. **Contiguous**: Just modify the directory to reflect the number of blocks is one less than the original. No disk I/O operation is needed.

ii. **Linked**: We need to read the first 99 blocks, change the address field of the 99th to `NULL`, and write it back. The number of disk I/O operations is $100 = 99 + 1$.

iii. **Indexed**: Shifting the addresses stored in the index one position upward does not need any disk I/O operation.

■