SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 300: Operating Systems I

Assignment 3

Sample Solution

POLICIES:

1. **Coverage**
   Chapters 7-9
2. **Grade**
   10 points, 100% counted into the final grade
3. **Individual or Group**
   Individual based, but group discussion is allowed and encouraged
4. **Academic Honesty**
   Violation of academic honesty may result in a penalty more severe than zero credit for an assignment, a test, and/or an exam.
5. **Submission**
   Electronic copy via CourSys
6. **Late Submission**
   2-point deduction for late submission within one week;
   5-point deduction for late submission over one week;
   Deduction ceases upon zero;
   Late submissions after the sample solution is available will NOT be graded.

QUESTIONS:

1. **2 points**
   Consider the following snapshot of a system:

	Allocation	Max
	ABCD	ABCD
P_0	3014	5117
P_1	2210	3211
P_2	3121	3321
P_3	0510	4512
P_4	4212	6325

Using the banker's algorithm, determine whether or not each of the following

states is unsafe. If the state is safe, illustrate the order in which the processes may complete.
a. Available = (0, 3, 0, 1)
b. Available = (1, 0, 0, 2)
[Grading Rubric: 1 point per state. If a safe sequence exists, BOTH the correct safe sequence AND the derivation steps are required.]

a. Unsafe.

Analysis guidelines:
Given the Allocation and Max columns, we can first derive the Request column by Allocation - Max:

	Allocation	Max	Request
	ABCD	ABCD	ABCD
P_0	3014	5117	2103
P_1	2210	3211	1001
P_2	3121	3321	0200
P_3	0510	4512	4002
P_4	4212	6325	2113

The Request column specifies how many more resources each process needs to complete.
Following the banker's algorithm, we compare Request with Available to decide whether the available resources are sufficient for a process to execute.

That is, if Request is no greater than Available, the corresponding process can be executed. Upon completion, the resources allocated to the process should be reclaimed and added to Available.
With the incremented Available, the banker's algorithm iterates the comparison of Request and Available for subsequent processes.

If Request is less than Available, the corresponding process cannot be executed.

b. Safe.
More than one safe sequence may be feasible, being omitted here.
Deriving a safe sequence still follows the preceding analysis guidelines.

2. 2 points

Consider a system with a number r of resources of the same type. These resources are shared by a number p of processes. A process can request or release only one resource at a time. Prove that the system is deadlock free given the following two conditions:
a. The number of the maximum need of each process is in [1, r];
b. The sum of all maximum needs is less than $r + p$.
[Grading Rubric: 2 points if a correct proof is provided. 0 point otherwise.]

Ref: http://www.cs.du.edu/~dconnors/courses/comp3361/assignments/sol3.txt

Suppose N = Sum($Need_i$), A = Sum($Allocation_i$), and M = Sum(Max_i).
Use contradiction to prove.

Assume this system is not deadlock free. If there exists a deadlock state, then at that state we have A = r (because by condition a, only one resource can be requested at a time). That is, all the r resources have been allocated. Since it's a deadlock state, no process is supposed to be executed next.
From condition b, N + A = M < r + p. So we get N + r < r + p, which derives to N < p. It shows that at least one process P_i that associates with $Need_i = 0$.
Associating with $Need_i = 0$ means that P_i does not need more resource to proceed. That is, it can be executed right away. This contradicts with the assumption that the system is under a deadlock state and no process can be executed.

3. **1 point**
Why are page sizes always powers of 2?

Consider a logical address space of 64 pages of 1,024 words each, mapped onto a physical memory of 32 frames.
a. How many bits are there in the logical address?
b. How many bits are there in the physical address?
[Grading Rubric: 1 point if ALL three subquestions are correctly answered with necessary derivations. 0 point otherwise.]

Ref: http://www.di-srv.unisa.it/~paodar/OS-exercise/8-sol.pdf

Power of 2:
Recall that paging is implemented by breaking up an address into a page and offset number. It is most efficient to break the address into X page bits and Y offset bits, rather than perform arithmetic on the address to calculate the page number and offset. Because each bit position represents a power of 2, splitting an address between bits results in a page size that is a power of 2.

a. 16 bits
6 (=log64) bits for page index;
10 (=log1024) bits for page offset;

b. 15 bits
5 (=log32) bits for page index;
10 (=log1024) bits for page offset;

4. **1 point**

Given six memory partitions of 300 KB, 600 KB, 350 KB, 200 KB, 750 KB, and 125 KB (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of size 115 KB, 500 KB, 358 KB, 200 KB, and 375 KB (in order)?
[Grading Rubric: 1 point if ALL three placement decisions are correctly determined. 0 point otherwise.]

Ref: http://www.salimarfaoui.com/Com310Lectures/main_Memory_Review.pdf

First-fit:
115 KB is put in 300 KB partition, leaving (185 KB, 600 KB, 350 KB, 200 KB, 750 KB, 125 KB);
500 KB is put in 600 KB partition, leaving (185 KB, 100 KB, 350 KB, 200 KB, 750 KB, 125 KB);
358 KB is put in 750 KB partition, leaving (185 KB, 100 KB, 350 KB, 200 KB, 392 KB, 125 KB);
200 KB is put in 350 KB partition, leaving (185 KB, 100 KB, 150 KB, 200 KB, 392 KB, 125 KB);
375 KB is put in 392 KB partition, leaving (185 KB, 100 KB, 150 KB, 200 KB, 17 KB, 125 KB)

Best-fit:
115 KB is put in 125 KB partition, leaving (300 KB, 600 KB, 350 KB, 200 KB, 750 KB, 10 KB);
500 KB is put in 600 KB partition, leaving (300 KB, 100 KB, 350 KB, 200 KB, 750 KB, 10 KB);
358 KB is put in 750 KB partition, leaving (300 KB, 100 KB, 350 KB, 200 KB, 392 KB, 10 KB);
200 KB is put in 200 KB partition, leaving (300 KB, 100 KB, 350 KB, 0 KB, 392 KB, 10 KB);
375 KB is put in 392 KB partition, leaving (300 KB, 100 KB, 350 KB, 0 KB, 17 KB, 10 KB)

Worst-fit:
115 KB is put in 750 KB partition, leaving (300 KB, 600 KB, 350 KB, 200 KB, 635 KB, 125 KB) ;
500 KB is put in 635 KB partition, leaving (300 KB, 600 KB, 350 KB, 200 KB, 135 KB, 125 KB);
58 KB is put in 600 KB partition, leaving (300 KB, 242 KB, 350 KB, 200 KB, 135 KB, 125 KB);
200 KB is put in 350 KB partition, leaving (300 KB, 242 KB, 150 KB, 200 KB, 135 KB, 125 KB);
375 KB must wait.

5. 1 point

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? Use LRU replacement, and assume 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;
```

[Grading Rubric: 1 point if ALL two subquestions are correctly answered with necessary derivations. 0 point otherwise.]

**Three types of solutions may be acceptable given different interpretation of "page size 200"!!

**CORRECT Solution 1:

a. 5000

b. 50

Each page contains up to 200 items. Then each page contains two rows.

a. In this case, the array is accessed column by column.
Each time an item is referenced, the row it belongs to should be loaded into memory, taking up a half page. Alongside, the next row is also loaded into the same page. Since two rows are loaded into the same page, they can serve two access requests. That is, one page fault occurs every two accesses.
Since the total number of accesses is 10000, the number of page faults is 10000/2 = 5000.

b. In this case, the array is accessed row by row. Each row needs a half page to store. That is, each page stores up to two rows. After the first item in a certain page encounters a page fault, the entire page it belongs to and the next page will be both loaded into memory. Thus, one page fault occurs every two rows. Since there are 100 rows in total, the number of page faults is 100/2 = 50.

**CORRECT Solution 2:

a. 10000
b. 100

Each array item is an integer, taking up 2 bytes (16 bits). Then each page holds 200/2 = 100 items.
Since each row contains 100 items, it needs 100/100 = 1 page.
The entire array needs 100 = 100 pages.

a. In this case, the array is accessed column by column. Each time an item is referenced, the row it belongs to should be loaded into memory, taking up one page. The next item to be referenced is at another row, which is current out of the memory. This reference will introduce a page fault and requires loading another row into memory. That row takes up the other page.
Following this observation, each array reference will cause a page fault.
The number of page faults is thus 100 x 100 = 10000.

b. In this case, the array is accessed row by row. Each row needs 1 page to store.
After the first item in a page encounters a page fault, the entire page it belongs to will be loaded into memory. That is, the other 100-1=99 items in the same page suffer from no page faults.
So, in this case, each row induces 1 page faults.
100 rows will generate 1 x 100 = 100 page faults.

**CORRECT Solution 3:
a. 10000
b. 200

Each array item is an integer, taking up 4 bytes (32 bits) Then each page holds 200/4 = 50 items.
Since each row contains 100 items, it needs 100/50 = 2 pages.
The entire array needs 2 x 100 = 200 pages.

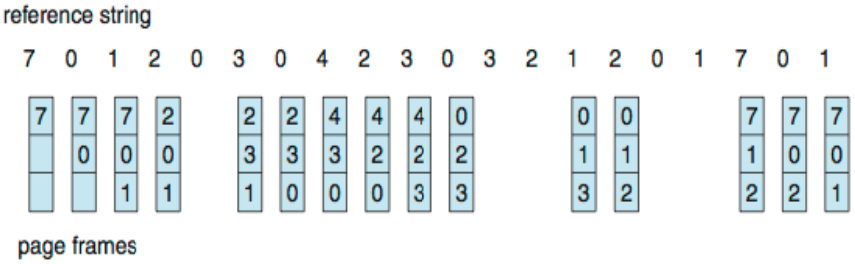
a. In this case, the array is accessed column by column.
Each time an item is referenced, the row it belongs to should be loaded into memory, taking up two pages. The next item to be referenced is at another row, which is current out of the memory. This reference will introduce a page fault and requires loading another row into memory.
Following this observation, each array reference will cause a page fault.
The number of page faults is thus 100 x 100 = 10000.

b. In this case, the array is accessed row by row. Each row needs 2 pages to store.
For either page, after the first item therein encounters a page fault, the entire page it belongs to will be loaded into memory. That is, the other 50-1=49 items in the same page suffer from no page faults.

So, in this case, each row induces 2 pages faults.
100 rows will generate 2 x 100 = 200 page faults.

6. **1 point**

Consider the following page reference string:
7, 2, 3, 1, 2, 5, 3, 4, 6, 7, 7, 1, 0, 5, 4, 6, 2, 3, 0, 1.
Assuming demand paging with three frames, how many page faults would occur for the following replacement algorithms?
a. LRU replacement
b. FIFO replacement
c. Optimal replacement
[Grading Rubric: 1 point if ALL three subquestions are correctly answered with illustrations similar as the following figure. 0 point otherwise.]



a. LRU replacement: 18 page faults

7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
7	7	7	1	1	1	3	3	3	7	7	7	7	5	5	5	2	2	2	1
	2	2	2	2	2	2	4	4	4	4	1	1	1	4	4	4	3	3	3
		3	3	3	5	5	5	6	6	6	6	0	0	0	6	6	6	0	0

b. FIFO replacement: 17 page faults

7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
7	7	7	1	1	1	1	1	6	6	6	6	0	0	0	6	6	6	0	0
	2	2	2	2	5	5	5	5	7	7	7	7	5	5	5	2	2	2	1
		3	3	3	3	3	4	4	4	4	1	1	1	4	4	4	3	3	3

c. Optimal replacement: 13 page faults

7	2	3	1	2	5	3	4	6	7	7	1	0	5	4	6	2	3	0	1
7	7	7	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	5	5	5	5	5	5	5	5	5	4	6	2	3	3	3
		3	3	3	3	3	4	6	7	7	7	0	0	0	0	0	0	0	0

7. **2 points**

In a multilevel cache system, the CPU first sends the memory request to level 1 cache. If it is a cache hit, the data is transferred to the CPU. If it is a cache miss, the CPU will send the same memory request to level 2 cache. If it is still a cache miss there, the CPU will further send the memory request to lower level caches

until a cache hit happens or memory access takes place.

A limitation of the preceding multilevel caching is that, even though a data block is cached in some lower level cache, the system still needs to endure all the time cost by the memory request going through all higher level caches with cache misses. Similarly, even if a data block is not cached, the CPU still sends memory request to one level of cache after another, taking likely a long time before accessing the memory.

Design a possible solution against the preceding limitation and show how it speeds up the average memory access time.
[Grading Rubric: Open question! Time to convince the TA.]

Open question, to be discussed in class.
The performance improvement by the proposed solution need be justified.