COMP530 FALL 2011 HW-6

STUDENT NAME: DUO ZHAO PID: 720090360
COLLABORATORS : HAOYANG HUANG, TIANREN WANG

1 Q1

1. Consider the following four (4) pseudo-code implementations of general semaphores. All use the following semaphore data type (although not all implementation use all the fields of the struct):

```
1 struct semaphore
2         mutex := binary_semaphore
3         delay := binary_semaphore
4         numWaiting := integer
5         queue := system_queue
6         value := integer
```

Are any of the following implementations correct? For each implementation that you think is correct (if any) give the best argument that you can that the implementation is correct (keep your argument to no more than 1 page of text). For each implementation that you think is incorrect (if any), demonstrate a set of interleavings of processes that call up and down that leads to some error condition. Make sure your description of any problem you discover in a semaphore implementation is clear and complete!

In answering this question you should assume an underlying round-robin process scheduler and assume that all semaphores used in an implementation have the appropriate initial value.

```
1  a)
2  down(var sem : semaphore)
3  begin
4      var next : processId
5      sem.mutex.down()
6      while (sem.value = 0) then
7          sem.numWaiting += 1
8          DISABLE_INTS
9          insert_queue(sem.queue, running)
10         next := remove_queue(readyQueue)
11         sem.mutex.up()
12         dispatch(next)
13         ENABLE_INTS
14         sem.mutex.down()
15     end while
16 sem.value := sem.value - 1
17 sem.mutex.up()
```

```
1
2  up(var sem : semaphore)
3  begin
4      var next : processId
5      sem.mutex.down()
6      sem.value := sem.value + 1
7      if (sem.numWaiting > 0) then
8          next := remove_queue(sem.queue)
9          sem.numWaiting -= 1
10         DISABLE_INTS
11         insert_queue(readyQueue, next)
12         ENABLE_INTS
13     end if
14 sem.mutex.up()
15 end up
```

Correct.

1) Mutual Exclusion/No race condtion:

For up(): The mutex lock in Line 5&14 guarantees mutual exclusion from other sem.down() or sem.up() operations, and the code between is atomic.

For down(): The mutex lock in Line 5&11&14 guarantees mutual exclusion between Line 7 and Line 10 in each while loop. The interruption-disabled operation guarantees Line 8 $\sim$ 13 is non-preemptive in each loop, with whose connection, code with while loop is atomic. Right before exiting the while loop, sem.mutex.down() is always called, therefore the reminder code Line 16 is also protect by sem.mutex lock.

2) sem.value $\geqslant 0$

It can be proven by induction.

(1) Initially, sem.value≥0 holds.

(2) suppose aftercertain up()s and down()s have been called, sem.value≥0 still holds and consider the next up()/down() function call. up() always increment sem.value, therefore sem.value≥0 holds. For down(), if the current sem.value≥1, sem.value≥0 holds in the next step. When sem.value = 0 and down() is called, it will enter the while loop(Line 6, Left column). In the Line 12, there is a context switch. It may switch to a process that may or may not change sem.value. Since sem.value is currently 0, other down() call will be blocked at line 5, and the resumed down() will be blocked on Line 14. if up() s are called after Line 12, sem.value≥ 1,then continued by Line 14 down(), sem.value≥ 0. Afterwards, the code will re-enter the loop (Line 6)for rechecking.

Based on (1) and (2). sem.value ≥ 0 is always true.

(3) up() always increments sem.value by 1; down() alway decrements sem.value by 1 unless sem.value=0.

This is guaranteed by Line 16, left column and Line 6, right column. There is no bifurcations that miss the sentences.

```
1  b)
2  down(var sem : semaphore)
3  begin
4      sem.mutex.down()
5      if (sem.value = 0) then
6          sem.numWaiting += 1
7          sem.mutex.up()
8          sem.delay.down()
9      else
10         sem.value := sem.value - 1
11         sem.mutex.up()
12     end if
13 end down
```

```
1
2  up(var sem : semaphore)
3  begin
4      sem.mutex.down()
5      if (sem.numWaiting > 0) then
6          sem.numWaiting -= 1
7          sem.delay.up()
8      else
9          sem.value := sem.value + 1
10     end if
11     sem.mutex.up()
12 end up
```

Incorrect. sem.value may be inconsistent with the atomic operation up() or down().

for up(), sem.value is not guaranteed to be incremented by one after it completes. If sem.value is currently zero, it executes Line 6&7 right column and exit up() without changing sem.value. Although sem.numWaiting > 0 indicates that there are sem.numWaiting down() blocking at Line 8, it does not guarantees that resumes immediate after Line 7 right column.

Similar problem for down(), when sem.value = 0, the down() function will be blocked at Line 8: sem.delay.down(). At the time of unblocking, sem.value≥ 1, however the remainder of the code does not decrease sem.value. In other words, Line 10 Left column sem.value should be put outside the if selection.

```
1  c)
2  down(var sem : semaphore)
3  begin
4      var next : processId
5      sem.mutex.down()
6      if (sem.value = 0) then
7          sem.numWaiting += 1
8          DISABLE_INTS
9          insert_queue(sem.queue, running)
10         next := remove_queue(readyQueue)
11         sem.mutex.up()
12         dispatch(next)
13         ENABLE_INTS
14     else
15         sem.value := sem.value - 1
16         sem.mutex.up()
17     end if
18 end down
```

```
1
2  up(var sem : semaphore)
3  begin
4       var next : processId
5       sem.mutex.down()
6       if (sem.numWaiting > 0) then
7            sem.numWaiting -= 1
8            next := remove_queue(sem.queue)
9            DISABLE_INTS
10           insert_queue(readyQueue, next)
11           ENABLE_INTS
12      else
13           sem.value := sem.value + 1
14      end if
15      sem.mutex.up()
16 end up
```

Incorrect. the value of the semaphore is not guaranteed incremented by one after each up() function call.

Suppose at a time point, sem.value = 0, sem.numWaiting = 1.

when up() is called, the code goes from all the left column except Line 13 without increment the semaphore value.

```
1  d)
2  down(var sem : semaphore)
3  begin
4      var next : processId
5      sem.mutex.down()
6      if (sem.value = 0) then
7          sem.numWaiting += 1
8          DISABLE_INTS
9          insert_queue(sem.queue, running)
10         next := remove_queue(readyQueue)
11         sem.mutex.up()
12         dispatch(next)
13         ENABLE_INTS
14         sem.mutex.down()
15     end if
16     sem.value := sem.value - 1
17     sem.mutex.up()
18 end down
```

```
1
2  up(var sem : semaphore)
3  begin
4       var next : processId
5       sem.mutex.down()
6       sem.value := sem.value + 1
7       if (sem.numWaiting > 0) then
8            sem.numWaiting -= 1
9            next := remove_queue(sem.queue)
10           DISABLE_INTS
11           insert_queue(readyQueue, running)
12           sem.mutex.up()
13           dispatch(next)
14           ENABLE_INTS
15      else
16           sem.mutex.up()
17      end if
18 end up
```

Incorrect, since sem.value may be negative in some cases.

Suppose sem.value = 0 and sem.numWaiting = 0. (This is possible. Like when initially, sem.value = 3, sem.numWaiting = 0. After 3 consecutive down() is called, sem.value = 0 and sem.numWaiting = 0 is current status.) In addition, the readyQueue is NULL. At this time another down() is called, there is nothing to dispatch and the the code will arrive line 16 and making the sem.value -1.

If the readyQueue is never empty. in some interleaving sense. sem.value == -1 is still possible. For instance. there are initially 2 down()s and 1 up() in the readyQueue. Initially, sem.value == 0. First down() blocks at Line 12 waiting for resumption, and switch to an up(), since sem.numWaiting > 0, it will arrive at Line 13(sem.value = 1 for now). And suppose it switch to another down(). it will jump to Line 16, after which sem.value = 0. Then the first down reassumes and completes its if selection and arrives at Line 16, after which sem.value = -1.

In addition there is dispatch(next) in up() function, which will block up() temporarily and grant CPU to another process, which is not the desirable effect (that will delay the code after dispatch) for up() function.

2 Q2

Consider the following producer/consumer system solution using general semaphores (see slide 9 from the lecture on semaphores):

```
1  process Producer
2  begin
3      loop
4          <produce a character "c">
5          emptyBuffers.down()
6          buf[nextIn] := c
7          nextIn := nextIn+1 mod n
8          fullBuffers.up()
9      end loop
10 end Producer
```

```
1  process Consumer
2      begin
3          loop
4          fullBuffers.down()
5          data := buf[nextOut]
6          nextOut := nextOut+1 mod n
7          emptyBuffers.up()
8          <consume "data">
9      end loop
10 end Consumer
```

For each of the (correct) semaphore implementation(s) from the previous problem, simulate the execution of this producer/consumer system and compare the execution to that of the same producer/consumer system that uses the implementation of general semaphores presented in class (on slide 31 of lecture 6), namely:

```
1  down(var sem : semaphore)
2  begin
3      var next : processId
4      sem.mutex.down()
5      if (sem.value = 0) then
6          sem.numWaiting += 1
7          DISABLE_INTS
8          insert_queue(sem.queue, running)
9          next := remove_queue(readyQueue)
10         sem.mutex.up()
11         dispatch(next)
12         ENABLE_INTS
13         sem.numWaiting -= 1
14     end if
15     sem.value := sem.value - 1
16     sem.mutex.up()
17 end down
```

```
1  up(var sem : semaphore)
2  begin
3      var next : processId
4      sem.mutex.down()
5      sem.value := sem.value + 1
6      if (sem.numWaiting > 0) then
7          next := remove_queue(sem.queue)
8          DISABLE_INTS
9          insert_queue(readyQueue, next)
10         ENABLE_INTS
11     else
12         sem.mutex.up()
13     end if
14 end up
```

For each of the correct semaphore implementation(s) from problem 1, how will the execution of the producer/consumer system differ from the execution of the same producer/consumer system using the implementation of semaphores from class? (Hint: They will differ in a big way!)

Compare the above code with the code in a)

The above code Line 12 right column: sem.mutex.up() is in only one branch of if selection. Therefore, if the code enters Line 7 with sem.numWaiting > 0, and the up() function ends. the sem.mutex is not released. In this case, only the code that previous blocked at Line 11 left column may be resumed. That is to say, if currently, sem.value == 0 and sem.numWaiting > 0, what's following an up() must be a down(), and up() cannot be repeatedly execute in this case.

Specially, if the size of the bounded buffer is 5. and currently the buffer is full. That is fullBuffers.value = 5 and emptyBuffers.value = 0 and there is 1 Producer blocked at emptyBuffers.down(). At this time, it's the Consumer's turn to execute. After the Consumer executes Line 7: emptyBuffers .up(), the producer must be resumed at Line 5. There is no possibility that the Consumer executes twice consecutively in this scenario. That's to say in the above implementation, if at any time the buffer is full and a producer arrives first(blocked). the following execution

sequence must be Consumer(), Producer(), ..., Consumer(), Producer() in an alternating(baton-passing) manner. This is not the case in the 1) implementation.

3 Q3

Recall that when an entry procedure bar of a Hoare-style monitor foo is called, the call is translated by the compiler in the series of calls:

```
1        foo.enterMonitor()
2        foo.bar()
3        foo.exitMonitor()
```

Modify the pseudo code presented in class that implements a run-time system for a Hoare-style monitor (the pseudo-code for the procedures enterMonitor, exitMonitor, signal, and wait) to develop a new runtime system for a Java synchronized class. That is, rewrite the pseudo code presented in class to implement four new functions: SyncClassEntry, SyncClassExit, wait, and notify. Note that you are not writing a Java program. You are writing generic operating system code (in the style of the pseudo code I present in class) that would be used by a Java implementation (e.g., a Java JVM) to support synchronized classes.

```
1  Var
2          monitorCodeMutex : binarySem: = 1;
3          monitorBusy: boolean := FALSE;
4          entryQueue: systemQueue;
5          entryQueue.numWaiting: integer:= 0; // the size of of entryQueue
6          waitQueue: systemQueue;
7          waitQueue.numWaiting: integer:= 0; // the size of waitQueue
```

```
1  ================================================
2  Procedure syncClassEntry()
3  begin
4          var next : process ID
5          monitorCodeMutex.down();
6          if(monitorBusy) then
7                  entryQueue.numWaiting += 1;
8                  entryQueue.insert_queue(running);
9                  DISABLE_INTS
10                 next := remove_queue(readyQueue);
11                 monitorCodeMutex.up();
12                 dispatch(next);
13                 ENABLE_INTS
14         else
15                 monitorBusy := TRUE;
16                 monitorCodeMutex.up();
17                 end if
18 end syncClassEntry
19 ================================================
```

```
1  ================================================
2  Procedure syncClassExit()
3  begin
4          boolean waiterAwoken = FALSE;
5          monitorCodeMutex.down();
6          waiterAwoken := wakeAWaiter();
7          if(!waiterAwoken) then
8                  monitorBusy := FALSE
9          end if
10         monitorCodeMutex.up()
11 end ExitMonitor
12 ================================================
```

```
1  ================================================
2  function wakeAWaiter(): boolean
3  begin
4          var waiter : Process ID;
5          if(entryQueue.numWaiting>0) then
6                  entryQueue.numWaiting -=1;
7                  waiter := remove_queue(entryQueue);
8          else
9                  return(FALSE);
10         end if
11         DISABLE_INTS
12                 readyQueue.insert_queue(waiter);
13         ENABLE_INTS
14         return(TRUE);
15 end wakeAWaiter
16 ================================================
```

```
1  ================================================
2  Procedure wait()
3  begin
4          var next : process ID
5          waitQueue.numWaiting += 1;
6          waitQueue.insert_queue(running);
7          monitorCodeMutex.down(); // block current process
8          if(!wakeAWaiter()) then
9                  monitorBusy := FALSE;
10         end if
11         DISABLE_INTS;
12         next := remove_queue(readyQueue);
13         monitorCodeMutex.up();
14         dispatch(next);
15         ENABLE_INTS
16 end Wait
17 ================================================
```

```
1  ================================================
2  Procedure notify()
3  begin
4          var next : process ID
5          if(waitQueue.numWaiting > 0) then
6                  waitQueue.numWaiting -= 1;
7                  entryQueue.numWaiting += 1;
8                  notified := remove_queue(waitQueue);
9                  entryQueue.insert_queue(notified);
10         end if
11 End notify
12 ================================================
```

4 Q4

An interactive computer system has enough memory to hold four programs in main memory. Assume programs spend 50% of their turnaround time waiting for I/O operations to complete when executing alone on a dedicated computer system. What fraction of time is the CPU idle when the programs execute simultaneously on the same computer system?

Note that your answer here will depend on a number of factors such as the scheduling policy employed, the structure of the jobs, the number of I/O devices present, etc. Thus, to answer this question, give upper and lower bounds on achievable processor utilization. To do this, you should

determine under what conditions the CPU would as busy as possible and as idle as possible and s. In all cases you should assume that the operating system always tries to overlap I/O and computation whenever possible.

fraction of CPU idle time - 0% ~ 50%

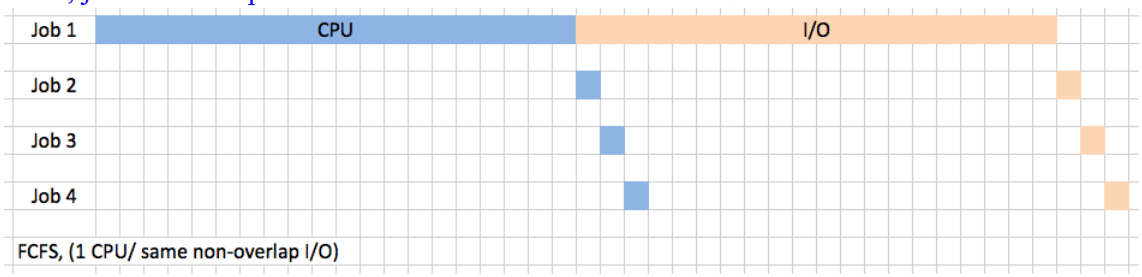
Example 1: CPU Idle fraction = 0%:

Time Slice	1	2	3	4	5	6	7	8
Job 1	CPU		I/O				CPU	
Job 2			CPU		I/O			
Job 3					CPU	I/O		
Job 4						CPU	I/O	
FCFS, each job access diffent I/Os, which can work concurrently.								
Runing	1	1	2	2	3	4	1	1
Reday	2	2	3	3	4		4	
	3	3	4	4				
	4	4						
Waiting			1	1	1	1		
					2	2		
						3		
Terminated							2	2
							3	3
								4
"-- CPU idle time = 0 --"								

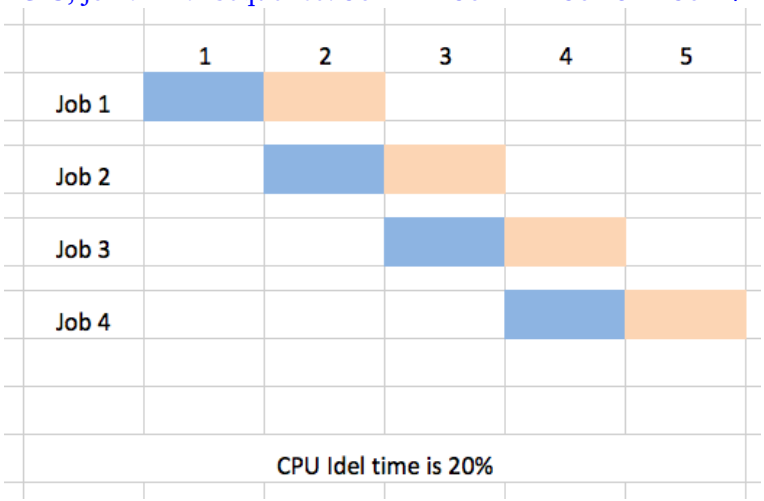
Example 2: CPU Idle fraction = 0%:

	1	2	3	4	5	6	7	8
Job 1								
Job 2								
Job 3								
Job 4								
FCFS(1 CPU/ Different I/O)								
Runing	1	2	3	4	1	2	3	4
Reday	2	3	4	1	2	3	4	
	3	4						
	4							
Waiting		1	1	2	3	4		
			2	3	4			
Terminated						1	1	1
							2	2
								3
"-- CPU idle time = 0% --"								

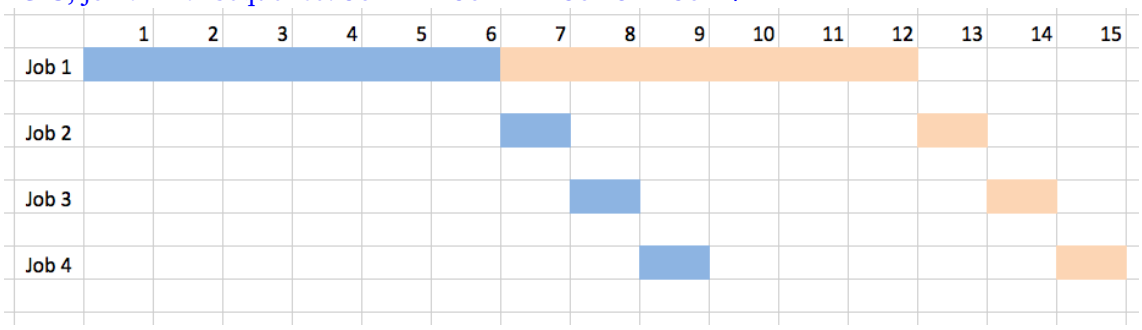
Example 2: CPU Idle fraction → 50% :
FCFS, job arrival sequence: Job 1 → Job 2 → Job 3 → Job 4



Example 3: CPU Idle fraction → 20% :
FCFS, job arrival sequence: Job 1 → Job 2 → Job 3 → Job 4



Example 4: CPU Idle fraction → 40% :
FCFS, job arrival sequence: Job 1 → Job 2 → Job 3 → Job 4



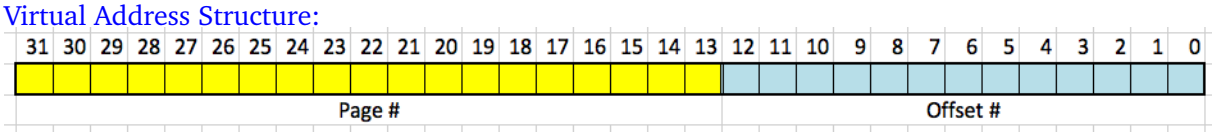
5 Q5

A machine has a 32-bit address space and an 8K byte page size. The page table for the running process is stored in a hardware memory management unit (MMU). In the MMU, the page table is stored in a large bank of 32-bit hardware registers; one register for each row of the page table. During a context switch, the page table for the previous running process is copied into an operating data structure (i.e., copied to main memory) and the MMU is loaded with the page table for the new running process (i.e., page table entries are copied from an OS data structure [from main memory] into MMU registers). Assume a read or write operation to an individual MMU register requires 50 ns (50×10^{-9} secs). Assume (because of pipelining) that main memory access times are negligible.

a) If the operating system uses a scheduling quantum of 100 ms ($100?10^{-3} secs$), assuming the

system always has processes to execute on the CPU, what is the overhead incurred by the system for managing the MMU on context switches? Express the overhead as the fraction of a quantum spent managing the MMU.

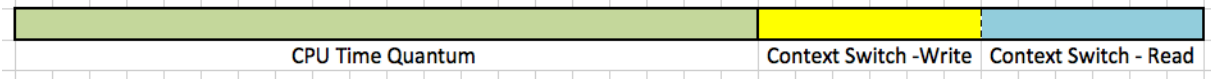
Virtual Address Structure:



The number of bits for offset number: 13 ($8K = 8 \times 2^{10} = 2^{13}$)

The number of bits for page number: 19 ($32 - 13$)

One period for CPU time slice and MMU management:



$$\text{Overhead} = \frac{\text{context switch(W\&R)}}{1 \text{ quantum} + \text{context switch(W\&R)}} = \frac{2 \times 2^{19} \times 50 \times 10^{-9}}{0.1 + 2 \times 2^{19} \times 50 \times 10^{-9}} = 34.4\% \quad (1)$$

b) Does it matter in your analysis if processes are primarily CPU bound or primarily I/O bound? Explain.

No, it does not matter if the processes are CPU bound or I/O bound.

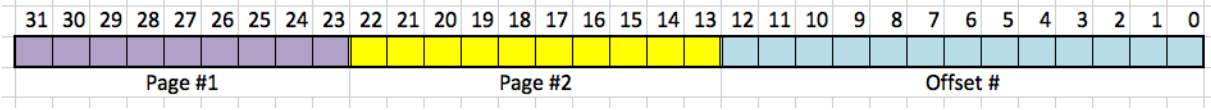
A CPU bound or I/O bound process tells whether the process spends most of their time in executing CPU or doing I/O. It's a global concept, compared with local time slice it's significantly large. The CPU bound or I/O bound property does not affect the general time slices (100 ms) and the MMU management time.

c) If one were to employ a multi-level paging scheme, would the overhead for managing the MMU during a context switch increase, decrease, or stay the same (or be roughly the same)? Explain.

The overhead of MMU management time will decrease. Since within the multi-level paging scheme, the size of page table stored in register will be significantly smaller. Mathematically, there is less permutations in the multi-level paging.

d) Design a multi-level paging scheme for this computer system such that the overhead of managing the MMU context switch consumes less than 1% of a scheduling quantum.

Virtual Address Structure:



Divide the page number space into two parts: 9 bits for page #1 and 9 bits for page #2, then the size of each page table is reduce to 2^9 in 1st Level and 2^{10} in the 2nd level, respectively.

(In the following scheme, among all 2nd level page tables, only ONE was loaded into register, others will be loaded on demanding, demanded 2nd level paging.)

$$\text{Overhead} = \frac{\text{context switch(W\&R)}}{1 \text{ quantum} + \text{context switch(W\&R)}} \quad (2)$$

$$= \frac{2 \times (2^9 + 2^{10}) \times 50 \times 10^{-9}}{0.1 + 2 \times (2^9 + 2^{10}) \times 50 \times 10^{-9}} = 0.153\% < 1\% \quad (3)$$

e) What is the impact of your scheme from (d), if any, on the time required to handle a page fault? Explain.

In the 2 level paging scheme above, The MMU first looks up the index in the 1st level page table to identify the 2nd level page table to access. On the 2nd level page table that is found by 1st level page table lookup, MMU search the page frame number according to the page #2.

The page fault can either occur in the final indexed frame is not loaded into memory, or the demanded 2nd level page table is not loaded into register. Therefore, the page replacement strategy should be capable handling both levels of page faults - faults due to unloaded page table, as well as faults due to unloaded page frames.

When page faults occur, the page replacement algorithm may be more complicated and result in therefore more time to handle. Page faults can occur at each page level so the page handling mechanism must be able to handle and locate all levels of page faults. For instance, in a 2nd level paging scheme, when the page fault occurs at 1st level, one of the 2nd level page tables was replaced by a new page table. As a result, all the frames that the old page table referred to was replaced. If there are a lot of dirty pages associate with the old page table, a lot more write-back-to disk storage is needed. That is to say if an upper level of pages was updated, all the lower levels of pages or page frames also needs to update as well.

6 Q6

A famous processor/computer system was designed that did not have a reference bit in the page tables it supported. Nonetheless, developers wanted to run operating systems on this system whose virtual memory management system required a reference bit. Develop a scheme the operating system can use to simulate the effect of having a reference bit. Describe how a clock page replacement algorithm would work with your simulated reference bit. Discuss what the cost of your simulation would be in terms of system overhead (compared to the operation of the virtual memory system if the hardware had a reference bit).

When a process is created, associate with its page table, an equal-size boolean list was created as well in the PCB. (boolean referenceBit[page table size] = 0). The array referenceBit works like the reference bit in page table entry. referenceBit[i] corresponds with the i^{th} row in the page table. Each time a page - page i - was referenced, referenceBit[i] is set to 1. The replacement policy checks the referenceBit[i] before attempting to replace page i. If referenceBit[i] equals 1, the clock pointer resets it to 0 and moves to the next page table entry, and therefore a recently referenced page is granted a second chance to remain in the memory.

7 Q7

Consider a page table in which the entries for the i^{th} and j^{th} pages (rows i and j of the page table) are both valid (the valid/resident bit is set in both entries) and both contain the same frame number.

a) What is the effect of allowing two entries in a page table to contain the same page frame number?

It allows two different virtual page numbers (Page i and j) mapping to the same physically frame. Therefore when a process generates two virtual addresses $VAS_1 = (p_i, o_k)$, $VAS_2 = (p_j, o_s)$ respectively, if their offsets are also identical (i.e. $o_k == o_s$), then VAS_1 and VAS_2 are mapped to the same physical memory address, although $VAS_1 \neq VAS_2 (p_i \neq p_j)$.

Since many-to-one mapping is allowed from page number space to frame number space, it may require a larger size of page table to ensure the desired entries in the corresponding physical memory space are mapped to, due to the redundancy in the page table compared to the non-redundant page table.

b) What would the effect of updating some byte in page i be on page j?

Since page i and page j are referenced to the same physical frame, page i and page j are automatically synchronized. If some instructions updated the data in page i, the change will be reflected in the view of page j as well, the content in i will change in the same manner as page i.

c) How could an operating system exploit this effect to decrease the amount of time needed to copy a large amount of memory from one place to another (e.g., to do a form of message passing)?

When a large amount of memory is needed to be duplicated, it may copy the page table referencing it instead of copying the actual physical contents. Since the size of page table is significant smaller than the frames it maps, this mechanism reduces the working time a lot. This is especially practical when large amounts of re-entrant codes exist and just copying the corresponding page table. However, for entrant codes, it may require additional synchronization policies to ensure data consistency.

Specially, It may need a read only bit to indicate whether a page is shared or not. If any process is seeking to modify a shared page, it may need to first copy its own copy on which it modified, in order to avoid unwanted data synchronization, like in message passing. If data synchronization is desired, additional synchronization mechanism must be introduced between the processes with respect to the shared frame, like mutual exclusion and condition synchronization.

d) Now consider allowing entries in the page tables of two different processes to contain valid frame numbers (i.e., two processes have valid page mappings that map to the same frame number). Explain how an operating system can exploit this effect to (1) efficiently create new processes while (2) ensuring complete memory protection between the processes (i.e., ensure that any data written by one process cannot be read by the second process). Assume the hardware MMU supports a Òread-onlyÓ bit in the page table that results in an exception being generated if a process writes to a read-only page.

When a new process is created, like $fork()$ in Unix, the system copies the page table pointing to the relevant memory rather than copying the actual data. Since the size of page table is significant smaller than the frames it maps, it will perform more efficiently. when a page is shared, the read-only bit is marked to protect process to modify the page. if any process attempt to modify the "shared page", it can capture the exception signal and hard-copy the memory. It follows that the process manipulate on its own copy. If there is not a lot write action, still a high portion of memory will be shared between the two processes and saves a lot hard-copying time.

8 Q8

Consider the two-dimensional array definition:

int A[100][100] /* C language definition */

The array is stored in row-major order (i.e., the array is stored in memory as A[1][1], A[1][2], A[1][3], ..., A[1][100], A[2][1], A[2][2], ...) starting at location 200 in the program's virtual address space. (Assume the program that accesses the array starts at location 0 of the program's virtual address space.) Assume a word-addressable paged virtual memory system with a page size of 200 words. Assume further that the compiler allocates a word of storage for each integer data type.

a) If a process executing this program is allocated three page frames, how many page faults are generated by each of the following array-initialization loops, using LRU page replacement? Assume page frame 1 contains the code for the process, and the other two frames are initially empty. Show how you etermined the number of page faults.

```
1  i)
2  for j := 1 to 100
3      for i := 1 to 100
4          A[i][j] := 0
5      end for
6  end for
```

```
1  j)                                  4          A[i][j] := 0
2  for i := 1 to 100                   5       end for
3     for j := 1 to 100                6  end for
```

i) 5000

Since the page size is 200 words and each word can store one integer data type, therefore each page can contain two rows ($\frac{200\times1}{100} = 2$)of int A. There are totally 50 virtual pages and only 2 page frames for page replacement, since the 1st frame is reserved for code, the 2nd and 3rd frames are for loading data. For each fixed column j, every other row i called will generate a page fault, since each page replacement will load two rows of the array. Therefore in total, given j goes through 1 to 100, $50 \times 100 = 5000$ page faults will occur.

ii)50

for each odd $i = 1,3,5,\cdots,99$, $j = 1$ page fault occurs(50 in total). After this page fault, all entries A[2n - 1][:] A[2n][:] will be loaded. therefore when j loops or i inrements from $2n-1$ to $2n$, it will hit in the loaded working table.

b) Repeat part (a) using the working set page replacement strategy with a window-size of 10 references. Again, show how you determined the number of page faults. (For this part you may ignore the memory references required to fetch instructions and instead focus only on the memory references that access the array A.)

Suppose the 50 pages are $p_1, p_2, \ldots, p_{50}$, in which $p_i (0 \leqslant i \leqslant 50)$ corresponds to $A[2i-1 : 2i][:]$.

i) 5000

The reference sequence is

$$j = 1 : p_1, p_1, p_2, p_2, \ldots, p_{49}, p_{49}, p_{50}, p_{50} \tag{4}$$
$$j = 2 : p_1, p_1, p_2, p_2, \ldots, p_{49}, p_{49}, p_{50}, p_{50} \tag{5}$$
$$\ldots \tag{6}$$
$$j = 100 : p_1, p_1, p_2, p_2, \ldots, p_{49}, p_{49}, p_{50}, p_{50} \tag{7}$$

The window size is 10, therefore there are only 5 or 6 referenced pages in the working set at any given time. However, there 50 page faults for each j. When j is incremented, the working set is large enough to hold the new page reference, so there are still page faults. For each j, there are 50 page faults and there are 100 $j's$. so the page faults are 5000 in total.

ii) 50

The reference sequence is

$$\underbrace{p_1, p_1, \ldots, p_1}_{\text{200 times}}, \underbrace{p_2, p_2, \ldots, p_2}_{\text{200 times}}, \ldots, \underbrace{p_{50}, p_{50}, \ldots, p_{50}}_{\text{200 times}} \tag{8}$$

The page faults occurs at the beginning of each "200 times" page request. since each intial p_i is not contained in the previous 10 references. Therefore there are total 50 page faults.