CoE 135 (Operating Systems): Problem Set 2

Salmon, Paulino III I.
2015-11557
paulino.salmon@eee.upd.edu.ph

I. Honor Code

I pledge that I answered all the items in this Problem Set to the best of my understanding, and with the aid of properly cited resources and collaborators in problems I could not solve on my own. I made sure that I cited all resources and stated the names of all those who guided me in answering all the items along with their invaluable contributions.

- Paulino I. Salmon III, 2015-11557

II. Address Translation

- (10 pts) Consider the following Segment Table:

Segment Number	Base	Length
0	220	500
1	2000	15
2	100	90
3	1333	555
4	1969	71

What are the physical addresses of the following logical addresses given the segment table above?

  - Segment 0, Address 430
    430 < 500 (Valid), 430 + 220 = **650**
  - Segment 1, Address 10
    10 < 15 (Valid), 10 + 2000 = **2010**
  - Segment 2, Address 500
    500 !< 90 **(Invalid)**
  - Segment 3, Address 400
    400 < 555 (Valid), 400 + 1333 = **1733**
  - Segment 4, Address 96
    96 !< 71 **(Invalid)**

Self Grade For #: 4

- (5 pts) A computing machine provides its users with a virtual-memory space of 2^{32} bytes. The computer has 2^{18} bytes of physical memory. The virtual memory is implemented by a paging scheme, and the page size is 4096 bytes. A user process requests access to the virtual address 11123456. Determine and explain how the system establishes the corresponding physical address.
  - Page sizes are typically represented in powers of two for the reason that it's easier to divide the binary format of a number to represent both the page number and the page offset. If it is given that the page size is $log_2(4096) = 2^{12}$, the remaining bits will therefore be given to the page table, $32 - 12 = 20; 2^{20}$.

Self Grade For #: 3

- (5 pts) You are given a demand-paged memory management scheme. It takes 10ms to service a page fault if an empty page is available or the replaced page is not modified, and it takes 18ms if the replaced page is modified. Memory access time takes 80ns. If the page to be replaced is modified 70% of the time, what is the maximum acceptable page-fault rate for an effective access time of 250ns?
  - Let n be equal to the page-fault rate (miss ratio)

    250ns = (n)[(10ms)(0.3) + (18ms)(0.7) + 80ns] + (1-n)(80ns)

    Solving for n we get:

$$n = 1.0897435 \cdot 10^{-5} \tag{1}$$

Self Grade For #: 4

III. Demand Paging

- (20 pts) Demand paging is a mechanism wherein we use a page replacement algorithm to manage the pages loaded onto the physical memory frames. Suppose that a newly created process is given 4 frames by the OS, and generates the following page references below:

$$123421351234151324 \tag{2}$$

(5 pts) How many page faults would occur with a FIFO (First-In-First-Out) page replacement policy? Place an 'X' in each box corresponding to a page fault. Place the total number of page faults under the # column.

1	2	3	4	2	1	3	5	1	2	3	4
X	X	X	X				X	X	X	X	X

1	5	1	3	2	4	#
	X	X		X	X	13

Self Grade For #: 4

(5 pts) How many page faults would occur with LRU (Least Recently Used) page replacement policy? Place an 'X' in each box corresponding to a page fault. Place the total number of page faults under the # column.

1	2	3	4	2	1	3	5	1	2	3	4
X	X	X	X				X				X

1	5	1	3	2	4	#
	X			X	X	9

Self Grade For #: 4

(5 pts) How many page faults would occur with LFU (Least Frequently Used) page replacement policy? Place an 'X' in each box corresponding to a page fault. Place the total number of page faults under the # column.

1	2	3	4	2	1	3	5	1	2	3	4
X	X	X	X				X				X

1	5	1	3	2	4	#
	X				X	8

Self Grade For #: 4

(5 pts) How many page faults would occur with clock replacement/second chance replacement policy? Place an 'X' in each box corresponding to a page fault. Place the total number of page faults under the # column.

1	2	3	4	2	1	3	5	1	2	3	4
X	X	X	X				X	X	X	X	X

1	5	1	3	2	4	#
	X			X		11

Self Grade For #: 3

IV. Reader-Writer Problem

- (20 pts) Reader-Writer Problem. You are given the Reader-Writer pseudocode below for accessing a shared resource: executed.

```
readerThreadRoutine() {
        lock.acquire();
        while (activeWrite > 0) {
                waitRead++;
                okToRead.wait(&lock);
                waitRead--;
        }
        activeRead++;
        lock.release();

        AccessResource(ReadOnly);

        lock.acquire();
        activeRead--;
        if (activeRead == 0 && waitWrite
            > 0)
                okToWrite.signal();
        lock.release();
}

writerThreadRoutine() {
        lock.acquire();
        while ( (activeWrite + activeRead
            + waitRead) > 0 ) {
                waitWrite++;
                okToWrite.wait(&lock);
                waitWrite--;
        }
        activeWrite++;

        lock.release();

        AccessResource(ReadWrite);

        lock.acquire();
        activeWrite--;
        if (waitRead > 0) {
                okToRead.broadcast();
        } else if (waitWrite > 0) {
                okToWrite.signal();
        }
        lock.release();
}
```

The code snippet above uses two condition variables, one for waiting readers and another for waiting writers. Suppose that all of the requests arrive in this order with a very small time interval between them. Note that none of these threads have started yet:

W1 R1 R2 R3 W2 W3 R4 R5 R6 W4 R7 W5 W6 R8 R9 W7 R10

(5 pts) In what order would the above code process the above requests? If you have a group of requests that are equivalent (unordered), indicate this by surrounding the equivalent/ unordered requests with square brackets '[]'. You can assume that the wait queues for condition variables are FIFO in nature (i.e. signal() wakes up the oldest thread on the queue). Explain your answer.

- The above code prioritizes writes over reads as implemented in the *writerThreadRoutine()* function. Therefore, the following sequence will be executed:

  W1 W2 W3 W4 W5 W6 [R1 R2 R3 R4 R5 R6 R7 R8 R9 R10]

Self Grade For #: 3

(3 pts) How would the order of processing the above requests change if W1 was currently running when the other requests arrive? Explain your answer.

- As writes are still prioritized over asynchronous reads, the sequence will remain the same:

  W1 W2 W3 W4 W5 W6 [R1 R2 R3 R4 R5 R6 R7 R8 R9 R10]

Self Grade For #: 2

(3 pts) What will be prioritised by the code snippet above? Is it the readers or the writers? Explain your answer.

- The code prioritizes the writers first, as the condition gets trapped in a while loop and is locked until both the *activeWrite* and the *activeRead* conditions are triggered (decremented or incremented) properly to break the lock.

Self Grade For #: 3

(3 pts) How does the code snippet above ensure that if activeRead > 0, then activeWrite == 0?

- TIn the first while loop of the *readerThreadRoutine()*, the functions gets stalled in the while loop until activeWrite reaches a value of zero. The following condition after the while loop gets broken (when activeWrite == 0) is where activeRead gets incremented, therefore, satisfying the conditions of the question. The lock is released afterwards.

Self Grade For #: 3

(3 pts) How does the code snippet above ensure that there can only be one writer at a time, and that if activeWrite == 1, then activeRead == 0?

- In the *readerThreadRoutine()* function, there is an if statement in the lower half that checks whether activeRead == 0 AND waitWrite > 0. If it so happens that the condition proves true, this signals that there are currently no active readers in the critical section and that a pending write is ready to be deployed, thus, the okToWrite.signal() is broadcasted, and the lock is released.
  With regards to how it ensures that activeWrite == 1 when activeRead == 0, this condition is checked in the first part of the *writerThreadRoutine()* function. The while loop ensures that neither activeWrite, activeRead nor waitRead has a value greater than 0. When this loop gets broken, it is ensured that activeRead has indeed a value of 0. The variable activeWrite is incremented to 1 afterwards.

Self Grade For #: 3

(3 pts) Why does the reader not need to check if waitRead > 0?

- The reader can only be deployed if no writes are currently running in the critical section. The writer checks the waitRead > 0 because it indicates that readers are queued up to enter the critical section, and will therefore broadcast a go signal to the queued up readers that it is safe to proceed. In comparison to the reader part, on the other hand, multiple queued up readers can safely enter the critical zone simultaneously as long as no writer is inside, eliminating the need to check if there is a ready reader on standby.

Self Grade For #: 3

V. Interprocess Communication

(20 pts) You are given the code template below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

#include <assert.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>

#define PAGESIZE 4096
#define CTRMAX 1000000

#define increment_val(ptr) (*ptr) += 1

int main() {
        uint32_t *smem = mmap(NULL, PAGESIZE,
        PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS, -1, 0);
        assert(smem != MAP_FAILED);

        *smem = 10;

        pid_t pid;

        pid = fork();
        assert(pid >= 0);

        if (pid) {
                for (int ctr = 0; ctr <
                    CTRMAX; ++ctr)
                        increment_val(smem);
                wait(NULL);
        }
        else {
                for (int ctr = 0; ctr <
                    CTRMAX; ++ctr)
                        increment_val(smem);
        }

        if (pid) {
                printf("Data: %d\n", *smem);
                int unmap_res = munmap(smem,
                    PAGESIZE);
                assert(unmap_res == 0);
        }

        return 0;
}
```

- (3 pts) The code snippet above will rarely print the value 2000010 at line 45, if at all, every time the code is executed. What is the reason behind this anomaly?
  - The #define for CTRMAX is only up until 1000000. This sets the limit for the increments, and even with the help of the child process increments, this limit will barely break the 2000010 limit required in this question.

Self Grade For #: 4

- (6 pts) Propose a modification to the given code template to ensure that the value 2000010 is always printed at line 45 (without printing it directly!).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

#include <assert.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>

#define PAGESIZE 4096
#define CTRMAX 2000000

#define increment_val(ptr) (*ptr) += 1

int main() {
        uint32_t *smem = mmap(NULL,
            PAGESIZE,
        PROT_READ | PROT_WRITE,
        MAP_SHARED | MAP_ANONYMOUS, -1,
            0);
        assert(smem != MAP_FAILED);

        *smem = 10;

        pid_t pid;

        pid = fork();
        assert(pid >= 0);

        if (pid) {
                for (int ctr = 0; ctr <
                    CTRMAX; ++ctr)
                        increment_val(smem);
                wait(NULL);
        }

        if (pid) {
                printf("Data: %d\n",
                    *smem);
                int unmap_res =
                    munmap(smem, PAGESIZE);
                assert(unmap_res == 0);
        }

        return 0;
}
```

Modifications: Changed CTRMAX to 2000000 and erased the increments in the child process if-else condition.

Self Grade For #: 4

- (5 pts) If the program above were to be implemented using message passing API, would the anomaly still occur without the modification needed for shared memory API? Explain your answer.
  - Since the upper limit is still bounded by the #define CTRMAX 1000000, the value 2000010 will still be, if at all, rarely printed, since changing the APIs should not affect the defines at the starting part of the snippet.

Self Grade For #: 4

- (6 pts) Present a reimplementation of the code template provided using message passing API.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

#include <assert.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define PAGESIZE 4096
#define CTRMAX 2000000

#define increment_val(ptr) ptr += 1

int main() {

        key_t key;
        key = ftok("probset", 65);

        uint32_t smem = msgget(key, 0666
            | IPC_CREAT);

        smem = 10;

        pid_t pid;

        pid = fork();
        assert(pid >= 0);

        if (pid) {
                for (int ctr = 0; ctr <
                    CTRMAX; ++ctr)
```

```
                increment_val(smem);
        wait(NULL);
    }
    else {
        for (int ctr = 0; ctr <
            CTRMAX; ++ctr)
                increment_val(smem);
    }

    if (pid) {
        printf("Data: %d\n",
            smem);
        msgctl(smem, IPC_RMID,
            NULL);
    }

    return 0;
}
```

Self Grade For #: 4

VI. Overuse of Threads and Processes

- (20 pts) Discuss two different reasons as to why the overuse of threads/use of too many threads is bad. Discuss two different reasons as to why the overuse or processes/use of too many processes is bad. Give an example for each reason.
  - *Threads* - one disadvantage of overusing threads is being able to **effectively synchronize all these created threads**. While the use of mutex can easily synchronize threads regardless of number, the idea of being able to do so with good performance is the downside. Another disadvantage of this would be is the **thundering herd problem**. This problem happens when a huge number of resources are awoken at the same time by an event they are waiting for. All these threads will eventually compete with each other as only one is able to win, therefore starving other threads of other resources. This will slow down the machine until this "herd" of threads is calmed down again.
  - *Processes* - while the **thundering herd problem** is applicable to threads, this problem is also applicable to the case of too many processes competing against each other. Too many processes would also lead to constant data swapping. This results to a problem known as **thrashing**. This happens when the present resources are overused due to the vast number of processes undergoing constant page faults and swapping.

Self Grade For #: 4

VII. Process States

- (10 pts) Determine which state the processes below belong to given their current situation.
  - (2 pts) Newly spawned child process
    Answer: **New**

    Self Grade For #: 4
  - (2 pts) Process is currently performing busy waiting
    Answer: **Waiting**

    Self Grade For #: 4
  - (2 pts) Process issued a write request to your Network Interface Card
    Answer: **Running**

    Self Grade For #: 4
  - (2 pts) Process just invoked exit()
    Answer: **Terminated**

    Self Grade For #: 4
  - (2 pts) Process was just kicked out of the CPU after its time slice expired
    Answer: **Ready**

    Self Grade For #: 3

References

[1] *Thrashing (computer science)*, Wikipedia, Accessed 2019-11-22, 2019. [Online]. Available: https : / / en . wikipedia . org / wiki / Thrashing _ (computer _ science) #Causes..

[2] *Thundering herd problem*, Wikipedia, Accessed 2019-11-22, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Thundering_herd_problem#:~:targetText=In%20computer%20science%2C%20the%20thundering,but%20only%20one%20will%20win..

[3] Borealid, *What's harder, synchronizing 2 threads or 1000 threads?* Stack Overflow, Accessed 2019-11-22, 2010. [Online]. Available: https://stackoverflow.com/questions/3362553/whats-harder-synchronizing-2-threads-or-1000-threads#:~:targetText=You%20could%20make%20the%20case,not%20so%20with%20only%202.&targetText=The%20real%20answer%20is%20%22synchronizing,in%20various%20ways%2C%20period.%22.

[4] Abraham Silberschatz, *Operating Systems Concepts*. Wiley, 2012.

[5] Limjoco, Jethro, *MEMORY MANAGEMENT PART I*, CoE135 Lecture, 2019.

[6] Jethro Limjoco, *PROCESS MANAGEMENT AND INTERPROCESS COMMUNICATION*, CoE135 Lecture, 2019.