CoE 135 (Operating Systems): Problem Set 1

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. Introduction and Prerequisites

- *A*. (10 pts) Top-level Concepts. An operating system has three main roles to perform, namely, virtualization, concurrency, and persistence. Discuss why each role is necessary for operating systems to act as a (justified) "deceiver" and "manager". Provide a high level discussion on one concrete example to strengthen your answer.
  - *Virtualization* - virtualization is the operating system's strategy to trick the processes into thinking that they have all the resources to themselves, and that these are exclusive. This part of the of the OS is the "deceiver". This role is necessary in the play as it doesn't let the processes hold back to the amount of resources they can consume, even if in reality, there is actually just a limited number of resources available.
  - *Concurrency* - concurrency is the virtualization's (the deceiver's) manager. There has to be some sort of way for the deceiver to successfully maintain its act of deception going without the processes "feeling" that they are actually being deceived. This role juggles every other process in the system to maintain this environment. Even if the computer has limited resources, this role swaps out these said processes from the RAM, the memory, etc, to maintain the virtualization's art of deception.
  - *Persistence* - persistence is the role that "memorizes" who the virtualization has already deceived so that the entire structure still maintains its consistency in deceiving. This part tracks down each process and knows their info such that no errors are encountered in swapping the deceived processes.

Self Grade For #: 4

- *B*. (10 pts) Fundamental Data Structures. The tree data structure is a commonly used container for data in operating systems due to its O(lg(N)) time complexity for search, insert, and delete operations assuming that the tree is balanced. However, some of these operations involve recursion, which is usually memory intensive. Is it possible to completely avoid using recursion when dealing with trees (i.e. during general tree traversal)? Why or why not? Explain your answer with code snippets if necessary.
  - The closest kind of tree traversal algorithm that does not use recursion is the **Morris Traversal**. This is an algorithm wherein the tree gets modified throughout the traversal, but is reverted back to its original shape after the entire operation. As per **morrisstackoverflow** the process goes this way: the root node gets initialized as the *current*. As traversal continues, a part of the tree gets 'relocated' and mounted on a lower part of the tree, virtually creating an "unlimited tree". Recursion does not happen because instead of relying on backtracking, the subtree is moved in an iterative way. A sample code from **morristraversal** of geeksforgeeks.org is attached below to show this algorithm:

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree tNode has data, a
    pointer to left child
and a pointer to right child */
struct tNode {
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Function to traverse the binary tree
    without recursion and
without stack */
void MorrisTraversal(struct tNode* root) {
    struct tNode *current, *pre;

    if (root == NULL)
        return;

    current = root;
    while (current != NULL) {
        if (current->left == NULL) {
            printf("%d ", current->data);
            current = current->right;
```

```
    }

        else {

            /* Find the inorder
                predecessor of current */
            pre = current->left;
            while (pre->right != NULL &&
                pre->right != current)
                pre = pre->right;

            /* Make current as the right
                child of its inorder
            predecessor */
            if (pre->right == NULL) {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in
                the 'if' part to restore
            the original tree i.e., fix
                the right child
            of predecessor */
            else {
                pre->right = NULL;
                printf("%d ",
                    current->data);
                current = current->right;
            } /* End of if condition
                pre->right == NULL */
        } /* End of if condition
            current->left == NULL*/
    } /* End of while */
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new
    tNode with the
given data and NULL left and right
    pointers. */
struct tNode* newtNode(int data)  {
    struct tNode* node = new tNode;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return (node);
}

/* Driver program to test above
    functions*/
int main() {

/* Constructed binary tree is
     1
   /   \
  2     3
 / \
4   5
*/
    struct tNode* root = newtNode(1);
    root->left = newtNode(2);
    root->right = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);

    MorrisTraversal(root);

    return 0;
}
```

Self Grade For #: 2.5

III. Process Management System Calls

- (3 pts) Let's start simple.

```
void main() {
    while (fork() >= 0);
}
```

Yes, this snippet will lead to a fork bomb and will generate an **infinite number of processes**. Fork() returns two values if successful: it returns the child's PID to the parent, and it returns 0 to the child that was created by the fork() call, stopping the created child processes from executing the while loop. However, since while loops treat any non-zero integer as the boolean "True", this while loop will never terminate for the parent process (as it returns the child's PID) and will keep on creating processes until the host runs out of resources, unless otherwise, it is protected by the system. Due to this fork bomb, the snippet will eventually eat up all available memory including the ones in the primary storage.

Self Grade For #: 4

- (3 pts) Adding **exec()** into the mix.

```
void main() {
    char *args[] = {"cat /usr/bin/env",
        NULL};
    execvp(args[0], args);
    fork();
}
```

This snippet will not result to a fork bomb. It will only generate **two copies of the processes**. The execvp() call here fails and returns a value of -1, therefore executing the fork() call after it, resulting to a generation of two processes, the parent and the child. If we are to run the **cat /usr/bin/env** terminal command successfully using execvp(), the line should be stated as **char *args[] = {"cat", "/usr/bin/env", NULL};** instead. These created processes will initially reside in main memory, as per the Unix OS's framework.

Self Grade For #: 4

- (4 pts) A bit of mathematical representation. Express your answer for this item in terms of n. Assume that the input is always an integer.

```
void main() {
    int n;
    scanf("%d", &n);
    if (n < 0)
        return 1;
    while (n >>= 1)
        fork();
}
```

As long as a non-negative value is supplied in 'n', the code will not return a value of 1, and will execute the fork bomb while loop. The number of processes generated depends fully on the input 'n' with respect to its highest power of 2 (its leftmost 1 for its bitstream representation). Plugging in 1 generates 0 processes, plugging in 2 generates 2 processes, 3 generates 2 processes, 4 generates 4, 5 generates 4, and so on. The output depends on the input's power of 2. Putting all these in an equation, the total number of processes generated in terms of n is: **2 raised to the integer value/floor division of:**

$$\left\lfloor \frac{log_2(n)}{1} \right\rfloor$$

Unless a very large n is supplied by the user, the program would not need that much memory to contain all the excess processes.

Self Grade For #: 4

- (4 pts) Data types. Note the additional question in the code comment.

```
void main() {
    int a = 0;
    //What if this was char instead of
        int?
    while (++a)
        fork();
}
```

For the case of the *int* data type, this snippet will fork bomb and will produce an infinite amount of processes until all the memory slots have been consumed by the fork bomb. Since the while loop has a "++a" condition, that condition will always be a non-zero value, in which, C will regard it to be true. Upon reaching int's maximum value, the variable 'a' will revert back to 0, in which, theoretically, fork() will be invoked by a total of 4294967295 times (assuming an int is 4 bytes) before the while loop breaks from the 0 reset, generating a total of $\mathbf{2^{4294967295}}$ processes.

For the case of the *char* data type, the program will stop running once it invoked fork() for a total 255 times, which is the maximum size (in bits) for a 1 byte char, as the counter will revert back to 0, breaking the while loop, after adding 1 to its maximum value. Adding recursion wherein all the created processes will call fork too, the total number of processes that will be created when converting it to char will be $\mathbf{2^{255}}$.

The int data type will probably bombard the memory with these processes, whereas, the char data type might still be able to contain it just in the main memory.

Self Grade For #: 4

- (6 pts) Hint 1: Draw some of it out, or Hint 2: Use Backtracking or Dynamic Programming.

```
void main() {
    fork();
    fork() && fork() || fork();
    if (fork() || (!fork()))
        if (fork() && (!fork()))
            if (fork() || fork())
                fork();
}
```

This snippet will not result to a fork bomb, but will generate a total of **149 processes** instead. Unless a lot of processes are already running prior to this program, the default memory allotted for these processes can still reside in the main memory.

Self Grade For #: 3

IV. I/O System Calls

- *A*. (10 pts) In your Machine Exercise 2, you were required to capture terminal input from the user without pressing the Enter key. There are two common ways to do this, via **ncurses** or via **termios**. Compare and contrast the two approaches in terms of capturing terminal input.
  - *ncurses*
    The ncurses package supports: overall screen, window and pad manipulation, and control over the terminal. This library supports raw/cbreak mode, meaning, the characters typed by the user are immediately passed through to the program. Upon pressing this specific key, this feature then generates a signal to be passed. In the raw mode of the terminal, as opposed to the typical canonical mode, the data just flows through freely without the interruption of the kernel. Ncurses allows for switching between these two modes via its built-in functions.
  - *termios*
    The termios library makes use of the cfmakeraw() function when switching to raw terminal mode and the ICANON flag in c_lflag whether or not it switches to canonical/non-canonical mode. Termios makes use of this flag to disable the buffered I/O and accept terminal inputs without the need for newlines.

Self Grade For #: 3

- *B*. (15 pts) Discuss all the steps that need to be performed when a C program expresses intent to write text to a file via fwrite().

- *(All these will be based on the strace output of the file API program from ME4.)*

  1) The first system call that would be executed is the **execve()** call, which signifies the start of the executable's life.

  2) The next call to be encountered is the **brk(NULL)**. This is where the program is allowed to manipulate the location of where it's going to "break".

  3) After there first two is where the **mmap()** calls come in. The process is now alloted some part of memory inside the system for it to run and do its job.

  4) The **access()** system call then comes into play. The purpose of the first instance of access() is to preload all the needed libraries that need to be set up before the process starts.

  5) After calling **fstat()** on the now opened file descriptor, and after all the initial system calls are finished executing, the calls now jump to the program's actual functionality, which is the objective of the **fwrite()** function.

  6) Initially, the file needs to be opened first and called for writing before any operation can be performed on it. The purpose of this is so that we can interact with the file itself by just treating it as a typical C variable. This is where **fopen()** comes in, which is the parallelism of the system call **open()**.

  7) After a series of **write()** and **fstat()** manipulations on the objective file itself, we when then need to call the **fclose()** function to commit all the changes we made to the file, which is the parallelism of the system call **close()**.

  8) After the entire process, this is now removed from the process list to free up memory for more processes that need it in the future.

Self Grade For #: 4

V. Asynchronous Programming and Signals

- (15 pts) You are given the code template below. Using the template, add additional code that will ensure that the string "data = 0" will be printed before "data = 1" in all instances the program is executed.

```
int data = 0;
// global variable

void process1() { data = 0; }
// for the child

void process2() { data = 1; }
// for the parent

int main() {

pid_t pid = fork();
int returnValue;

    switch(pid) {
    case -1:
        perror("Fork error.\n");
        exit(1);
    case 0:
        process1();
        returnValue = 44;
        break;
    default:
        wait(&returnValue);
        process2();
        break;
    }
    printf("data = %d\n", data);
    return 0;
}
```

In this code, I just used process1() and process2() as functions that assign an integer value to the variable 'data'. In this case I dedicated process1() for the child, and process2() for the parent. In this implementation, "data = 0" will always print first before "data = 1" since in all instances since, in the case switch conditions, the parent waits for the child to finish executing first before it even starts to jump in process2().

Self Grade For #: 4

VI. File Systems

Consider the following inode structure:

```
typedef struct {
    block_sector_t direct_ptr[16];
    block_sector_t singly_indirect[8];
    //no doubly_indirect
    block_sector_t triply_indirect[2];
    size_t sz;
} inode_t;

typedef struct {
        block_sector_t block_nums[256];
} indirect_block_t;
```

- *A.* (6 pts) If each block/sector had a block size of 1kB, what is the maximum file size supported by this design?
  - 16 direct pointers, 8 singly direct, 2 triply direct, 256 pointers in 1 block, 1kB per block.
    $(16 + (8)(256) + (2)(256^3)) * (1024)$ = **32 GB**

Self Grade For #: 4

- *B*. (3 pts) How many data blocks/sectors are needed to represent a file with maximum file size in this design?
  - 32 GB = 2^{35}; Block size: 1kB = 2^{10}
    $2^{35}/2^{10}$ = $\mathbf{2^{25}}$ **blocks**

Self Grade For #: 4

- *C*. (3 pts) How many singly indirect inode blocks/sectors are needed to represent a file with maximum file size in this design?
  - Maximum file size that **one** singly indirect node can represent: $256^{1+1} * (1024) = 2^{26}$
    Number of singly indirect pointers needed to cover a 32 GB (or 2^{35}) size: $2^{35}/2^{26}$ = $\mathbf{2^{9}}$ **blocks**

Self Grade For #: 4

- *D*. (3 pts) How many doubly indirect inode blocks/sectors are needed to represent a file with maximum file size in this design?
  - Maximum file size that **one** doubly indirect node can represent: $256^{2+1} * (1024) = 2^{34}$
    Number of doubly indirect pointers needed to cover a 32 GB (or 2^{35}) size: $2^{35}/2^{34}$ = **2 blocks**

Self Grade For #: 4

- *E*. (3 pts) How many triply indirect inode blocks/sectors are needed to represent a file with maximum file size in this design?
  - Maximum file size that **one** triply indirect node can represent: $256^{3+1} * (1024) = 2^{42}$
    Number of triply indirect pointers needed to cover a 32 GB (or 2^{35}) size: $2^{35}/2^{34}$ = $\mathbf{2^{-7}}$ **blocks**

Self Grade For #: 4

- *F*. (2 pts) How many data blocks/sectors will be used by a file of size 6GB?
  - 6 GB/ 1kB = $\mathbf{2^{22.5849625}}$ **blocks**

Self Grade For #: 4

VII. Bonus

- (10 pts) Machines running Linux OS commonly make use of the ext* family of file systems. Briefly discuss the common features among ext2, ext3, and ext4, as well as the improvements from ext2 to ext3, and ext3 to ext4.
  - *ext2*
    - Second Extended File System
    - Developed to overcome the limitations of the first ever file system.
    - Does not have a journaling feature.
    - Maximum filename length of 255 characters.
    - Maximum individual file size range: 16 GB - 2 TB.
    - Overall filesystem size range: 2 TB - 32 TB.
  - *ext3*
    - Third Extended File System
    - Was implemented in Linux 2.4.15.
    - Now has a journaling feature..
    - Maximum filename length of 255 characters.
    - Maximum individual file size range: 16 GB - 2 TB.
    - Overall filesystem size range: 2 TB - 32 TB.
  - *ext4*
    - Fourth Extended File System
    - Was implemented in Linux 2.6.19.
    - Can also support journaling filesystems..
    - Maximum filename length of 255 characters.
    - Maximum individual file size range: 16 GB - 16 TB.
    - Overall filesystem maximum size: 1 EB (Exabyte).

  As per **extbonus** and **extbonus3** the notable improvements of the filesystems would be its drastic changes to the maximum file sizes/filesystem sizes that it can handle. Ext2 and ext3 could only hold until 2 TB per file, whereas, ext4 jumped to an even higher value of 16 TB. The same goes for their improved maximum filesystem size: ext2 and ext3 only has a peak of 32 TB, whereas, ext4 now has a skyrocketing amount of 1 exabyte.
  Another improvement worth noting would be is: the progression of adding support for journaling filesystems.

Self Grade For #: 4