Design Document for Nachos
Phase 1: Multiprogramming

Abdulaziz Awliyaa, Christopher Connors, Brian Giroux
COSC 3407 - Operating Systems
Group 1

March 18, 2017

Task I – Implement File System Calls

Changes to `UserKernel.java` and Creation of New `File` class

In `UserKernel.java`, we will create a 64 element array which will store all global `File` file objects which are currently active. The structure of the `File` class will look like this:

Listing 1: Outline of `File` class

```
public class File{
  Object OpenFile
  int counter = 1;
  boolean isWriting = FALSE;
  boolean isLinked = TRUE;

  /* ACCESSORS */

  /**
   * @return OpenFile object
   */
  OpenFile getOpenFile();

  /**
   * @return this.counter
   */
  int getCounter();

  /**
   * @return TRUE if this.counter is greater than zero
   *         (ie, there is at least one open file)
   */
  boolean isOpen();

  /**
   * @return this.isWriting
   */
  boolean getIsWriting();
```

```
    /**
     * @return this.isLinked
     */
    boolean getIsLinked();


    /* MUTATORS */

    /**
     * increments the this.counter by 1
     */
    void incCounter();

    /**
     * decrement the this.counter by 1
     */
    void decCounter();

    /**
     * Sets this.isWriting to TRUE.
     */
    void setWriting(boolean writing);

    /**
     * Sets this.isLinked to TRUE.
     */
    void setIsLinked(boolean linked);
}
```

Changes to the `UserProcess.java` class

Implementing the `creat()` Method

This method will create a new file if it does not already exist. If it was already created (ie, it exists in the global array), we will just link to it.

See *Algorithm* 1 on the following page.

Algorithm 1: The `creat()` method

parameter: *name – a pointer to the filename
return **:** int

```
if file is in global array of files then
    if we are not already linked to the file then
        in user process array, add link to the file in global array;
    end
else
    in global array, add the file;
    in user process array, add link to the file in global array;
end
```

Implementing the `open()` Method

We can only open an existing file. If a file does not exist, we return -1, otherwise it returns the file descriptor.

See *Algorithm* 2 on the next page.

Implementing the `read()` Method

This method will allow us to read a number of bytes from a file. The bytes we read from the file will be placed in a buffer.

See *Algorithm* 3 on page 5.

Implementing the `write()` Method

This method will allow us to write a number of bytes from to file. The bytes we write to the file will come from a buffer.

See *Algorithm* 4 on page 6.

Implementing the `close()` Method

This method closes a file if it is not being written to by another process.

See *Algorithm* 5 on page 6.

Algorithm 2: The `open()` method

```
parameter: *name – pointer to the name of the file to open
return   : int – the file descriptor of the file

if OpenFile array exists then
    if the file is in the array then
        call setIsOpen;
        set TRUE;
        return file descriptor (local index);
    else
        if the file is in the global array then
            if the file is linked then
                in user process array, add link to the file in global array;
                return file descriptor (local index);
            else
                return −1;
            end
        else
            return −1;
        end
    end
else
    return −1;
end
```

Implementing the `unlink()` Method

After a process closes a file, it must be unlinked from the table. See *Algorithm* 6 on page 7.

Test Cases

We will:

- open some files
- close the files
- unlink them then try to re-open them

Algorithm 3: The `read()` method

parameter: int fileDescriptor – the file descriptor of the file to read
*buffer[] – a pointer to the buffer array
int count – the number of bytes to read
return : int – the number of bytes actually read

```
counter ← 0;
if the file descriptor is valid then
    if the file is open then
        while not end of file do
            read byte from file to buffer;
            increment counter;
        end
        return counter;
    end
else
    return −1;
end
```

- call halt from a running process that that has open files
- read and write to open files
- read and write to 1 and 0 fileDescriptor without opening them

Algorithm 4: The `write()` method

parameter: int fileDescriptor – the file descriptor of the file to write to
*buffer[] – a pointer to the buffer array
int count – the number of bytes to write
return : int – the number of bytes written

```
if the file descriptor is valid then
    if file is open then
        isWriting ← TRUE;
        for i ← 0 to count − 1 do
            write buffer[i] to file;
        end
        isWriting ← FALSE;
    end
else
    return −1;
end
```

Algorithm 5: The `close()` method

parameter: int fileDescriptor – the file descriptor of the file to close
return : int – returns −1 if there was a problem

```
if the file descriptor is valid then
    if the file is open then
        if the file is being written to then
            wait for the write to end;
        end
        decrement the files counter;
        call unlink();
    else
        return −1;
    end
else
    return −1;
end
```

Algorithm 6: The `unlink()` method

```
parameter: *name – pointer to the name of the file to unlink
if file exists in global array then
    if file is open then
        set isLinked to FALSE;
    else
        remove file from global array;
    end
end
```

Task II – Implement Support for Multiprogramming

Changes to `UserKernel.java`

We will create a linked list to use as a global page table. We will be able to call `getNumPhysPages()` and set the max size of the Linked List to this number. The head of the global page table will be set to the first available page.

Changes to `UserProcess.java`

We will first check if pageTable is already created. If it doesn't already exist, we will create an array of 8 elements that will serve as the pageTable.

We will create a new array of type `TranslationEntry[]` called `translations1`. This pageTable will request free pages from the `UserKernel` global page table and ask for the head of the list. It will then load that page, move the head of the list, load the next page, and so on, until all 8 pages requested are placed in the array. Then it will request that the kernel load in the program from memory. It will use the `translations1` array to translate between the physical memory and the virtual memory.

When the program calls `exit()`, we must make sure that it sets the bit in the global array to indicate that it is free, and to move the head of the list.

To read and write virtual memory, we must make sure we check the PID of the requesting process and check which virtual pages it belongs to, that way a process can not read from or write to another processes table.

Implementing `loadSection()`

This method will allocate the pages that a process needs.

Test Cases

To test this task, we will:

- load programs into memory
- try and fill memory and load another process

- load multiple programs into one process and see if the memory fails
- close programs and make sure memory is freed
- open 2 processes and try to read and write to each other's
- page table which should not work

Task III – Implement System Calls

Implementing the `join()` Method

Only a child can join a parent so we must make sure that when the parent calls `join()`, it knows the PID of the created process and those underneath it. We will keep a linked list of child PIDs in the process.

We will check that `childExit = FALSE` to make sure that the child does indeed exist before joining it.

Inside the `UserKernel.java`, we will create a PID counter that counts up every time a new process is created.

In `UserProcess.java`, we need need a field called "`int pid`" that is set by the kernel on startup.

Implementing the `exit()` Method

We will set global page table to release virtual memory and physical memory. We will set PID and name to null (the process is no longer accessible).

The parent process will have a method called `childExit()` that is called by the child when it exits and sets a Boolean flag.