Task 2:

Introduction

The purpose of task 2 is to implement support for multiprogramming. Specifically, we must find a way to allocate the machine's physical memory without overlap in memory usage. The solution must make use of "gaps" in the free memory pool. The process's memory must be freed on exit. Each process must maintain a page table. A global frame table is also used to list the available pages of memory.

List of methods to add/modify

UserKernel Methods

NEW

-allocatePages – used with UserProcess.loadSections to allocate pages.

-releasePage – used with UserProcess.unloadSections to release pages.

MODIFIED

-initialize – now also initializes free pages through a linked list, as well as a lock for synchronization purposes. (linked list and lock are global variables)

UserProcess Methods

NEW

-getTranslationEntry -

@params : int VirtualPageNumber(vpn), boolean writeNotRead[1]

@return: result (TranslationEntry)

MODIFIED

-readVirtualMemory – added virtual to physical translations

-writeVirtualMemory – added virtual to physical translations

-loadSections – Allocates physical page numbers for size needed + stack

-unloadSections – releases pages previously allocated and closes coff object

[1] sets the TranslationEntry's dirty bit to true is the translation is for a write.

Pseudocode – UserKernel.initialize

```
//listLock and freePages are global variables to UserKernel. The following was added to the method:
listLock = new Lock();
freePages = new LinkedList
for(inti  = 0; i < Processor's physical pages; i++)
        freePages.add(i);
```

Pseudocode –UserKernel.allocatePages

```
//New method. Param = number of pages to allocate (num). returns int array containing page numbers
// from freePages list. Note that pages allocated are not necessarily contiguous.  UserKernel.releasePage
//(see below) might cause gaps. result for example could be = {2, 3, 6, 7, 13} This method is used by
//UserProcess.loadSections()
acquire listLock;
if(freePages.size() < num) {
        release listLock;
        return null;
}
for (int i = 0; i < num; i++)
        result[i] = freePages.remove();
release listLock;
return result;
```

Pseudocode – UserKernel.releasePage

```
//Releases a page specified by a physical page number(physPgNum). Used by
//UserProcess.unloadSections()
acquire listLock;
freePages.add(physPgNum);
release listLock;
```

Pseudocode – UserProcess.readVirtualMemory

```
//previously, it was assumed that virtual addresses equal physical addresses. Now we do this:
int VPN = //use Processor class to get first Virtual Page Number (arg = vaddr)
firstOffset = //use Processor class to get offset (arg = vaddr)
vpnEnd = //use Processor class to get last Virtual Page Number (arg = vaddr + length)

Translation entry = getTranslationEntry(VPN, false) //see getTranslationEntry method below
if(entry == null)
        return 0;
amount = Min(length, pageSize – firstOffset);
ArrayCopy(from memory starting at entry physical address,
            to array specified in data array, starting at offset specified
            for specified amount)
offset += amount;
for(inti  = VPN + 1; i <= vpnEnd; i++) {      //for the remaining bytes, do
        entry = getTranslationEntry(i, false);
        if(entry == null)
                return amount;
        amount2 = min(length – amount, pageSize);
ArrayCopy(same as ArrayCopy above, except copied for specified amount2)
        offset += amount2;
        amount += amount2;
}
return amount;
```

Pseudocode – UserProcess.writeVirtualMemory

```
//Almost exactly like above, except we are writing instead of reading. Only the ArrayCopy portion will be
//different, and the second argument used in getTranslationEntry will be true instead of false.
//for example:
```

```
TranslationEntry entry = getTranslationEntry(VPN, true);
…
ArrayCopy(from array specified in data array, starting at offset specified,
                to memory, starting from entry physical address
               for specified amount)
….
entry = getTranslationEntry(i, true);
…
ArrayCopy(same as ArrayCopy above, except copied for specified amount2)
…
```

Pseudocode – UserProcess.getTranslationEntry

```
//takes int VPN and Boolean writeNotRead as arguments, returns a TranslationEntry object. (see
//Machine. TranslationEntry.java). It was hard to write in pseudocode, it seemed more intuitive and
//logical as actual code that I already did.
if (vpn <0 || vpn >= numPages)
        return null;
TranslationEntry result = pageTable[vpn];
if(result == null)
        return null;
if(result.readOnly && writeNotRead) //can't write to a read-only page
        return null;
result.used = true;
if(writeNotRead)
        result.dirty = true;
return result;
```

Pseudocode –UserProcess.loadSections

```
//Allocate physical page numbers, see UserKernel.
physicalPageNumsArray = userKernel.allocatePages(numPages);
if(phsyicalPageNumsArray == null){
        coff.close();
        Execute Debugger for insufficient physical memory
        return false;

        //only modify what's inside the second for-loop
        for (int i = 0; i<section.getLenth(); i++){
                vpn = section.getFirstVPN()+1;
                ppn = physicalPageNumsArray[vpn];
                pageTable[vpn] = new TranslationEntry(i, physicalPageNumsArray[i], true, false, false,
                                                        false);
                section.loadPage(i, ppn);
        }
        //outside of the first for-loop:
        //allocate free pages for stack
        for(int i  = numPages – stackPages; i < numPages; i++) {
                pageTable[i] = new TranslationEntry(vpn, ppn, true, section.isReadOnly(), false, false)
                section.loadPage(i, ppn);
        }
}
```

Pseudocode – UserProcess.unloadSections

```
// a lot simpler. close coff, release pages and make pageTable == null.
coff.close();
for(int i  = 0; i < numPages; i++)
        UserKernel.releasePage(pageTable[i].ppn);
```

```
pageTable = null;
```

Test Cases

Test 1 – Make sure that non-contiguous memory allocation works. Try allocating and deallocating memory in a way to create gaps (fragmentation)

Test 2 – Test memory bounds by asking for more memory than is available.

Test 3 – Make sure virtual to physical and physical to virtual translations are valid.

Test 4 – Make sure the right amount of memory is allocated (including memory requested + stack)

Test 5 – Make sure pages are released properly (ask for these pages specifically using a test program once they are released)

Test 6 – Make sure memory allocated to a process cannot be accessed by a test program.