Allocating the machine's physical memory in UserKernel

To allocate the machine's physical memory so that different processes can access it without memory overlap, utilize a lock called freePageLock and a LinkedList called freePage which will be used as a global link list that stores the physical memory page numbers of the machine. To initiate it, a for loop is used to iterate through the physical pages of the machine. And the freePageLock is used to ensure the page tables are not being used when adding the physical pages. Then the page number is added to the freePage linked list and the freePageLock is released. This repeats for every iteration until the process of the initiation is complete. This allows different threads to access the freePage linked list and use the physical memory page numbers stored in it.

When the process ends and exit, unloadSections, or even illegal operation is called, or the user process will call the delPage. acquires the freePageLock before adding the page to the freePage and frees the memory.

```
User Kernel:
  public class UserKernel extends ThreadedKernel {
   public void initialize(String[] args) {
    freePageLock = new Lock();
    freePage = new LinkedList<Integer>();

    for (int i = 0; i < Machine.processor().getNumPhysPages(); i++){
        freePageLock.acquire();
        freePage.add(i);
        freePageLock.release();
      }
    }
    public static void delPage(int ppn){
        freePageLock.acquire();
        freePage.add(ppn);
        freePageLock.release();
    }
```

Load and Unload Sectionas

In the UserProcess, the loadSections() method is responsible for allocating the requisite number of pages to the user process based on the size of the user program and the allocation policy. By using TranslationEntry - a single translation between a virtual page and a physical page, the virtual page number is translated into the physical page number. If the number of virtual pages exceeds or just does not match the number of physical pages, an error is returned. Otherwise, The sections of the page numbers are loaded into their respective physical memory pages in a for-loop one section at the time.

To clear up the resources used by the loadSections(), the unloadSections method is used to reset all the process page tables by getting the virtual page table entry and translating it, and call UserKernel.delPage() to add that physical page as freepage. This ensures that the page is free to use by another process, thus freeing up physical memory for other processes. After deleting the pages, the coff file is then closed.

```
    protected boolean loadSections() {

        if (numPages > Machine.processor().getNumPhysPages()) {
            coff.close();
            Lib.debug(dbgProcess, "\tinsufficient physical memory");
            return false;
        }

        for (int s=0; s<coff.getNumSections(); s++) {
            CoffSection section = coff.getSection(s);

            Lib.debug(dbgProcess, "\tinitializing " + section.getName()
                    + " section (" + section.getLength() + " pages)");

            for (int i=0; i<section.getLength(); i++) {
                int vpn = section.getFirstVPN()+i;
                TranslationEntry entry = pageTable[vpn];
                if (entry ==null){
                    return false;
                }
                section.loadPage(i, entry.ppn);
            }
        }
        return true;
    }
    protected void unloadSections() {
        for (int i = 0; i < pageTable.length; i++){
            if(pageTable[i].valid){
                UserKernel.delPage(pageTable[i].ppn);
                pageTable[i] = new TranslationEntry(pageTable[i].vpn, 0,
false, false, false, false);
            }
        }
```

```
        coff.close();
    }
```

writeVirtualMemory

From the original function which let physical and virtual memory the same, this function is used to write data from a specified array to a process's virtual memory. It handles the address translation details and ensures that the process is not destroyed if an error occurs. It begins by ensuring that the offset and length are greater than or equal to zero, and that the offset plus length is less than or equal to the length of the data. The function then gets the memory from the processor and sets up variables to keep track of the offset in the page, the virtual page number of the process, and the amount of data transferred. It then loops through each page, setting the page table as used and dirty in the page to indicate the page is written by a user program, a table and concatenate a page number the data to the virtual memory, until the number of bytes turns to 0, calculate by minusing the length to amount that is transferred each time. Finally, it returns the number of bytes successfully copied.

readVirtualMemory

readVirtualMemory is Pretty similar to writeVirtualMemory, but instead of

writing data from an array to the processor's memory, it reads data from the processor's memory to the array. And Only mark the page as used, which means read.

```
    public int writeVirtualMemory(int vaddr, byte[] data, int offset,int
length) {
    Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <=
data.length);
    if (vaddr < 0||numPages==0){
        return 0;
    }

    byte[] memory = Machine.processor().getMemory();
    int addOffset = Processor.offsetFromAddress(vaddr);
    int vpn = Processor.pageFromAddress(vaddr);
    int transferred = 0;

    for (int i = vpn; length>0; i++) {
         if (vpn>numPages){
            return transferred;
        }
        TranslationEntry entry = pageTable[i];
        entry.used = true;
        entry.dirty = true;
```

```
        int phyAddr = Processor.makeAddress(entry.ppn,addOffset);
        int remain = Math.min(length, pageSize - addOffset);
        System.arraycopy(data, offset, memory, phyAddr, remain);
        addOffset = 0;
        transferred += remain;
        offset += remain;
        length -= remain;
    }
    return transferred;
    }

    public int readVirtualMemory(int vaddr, byte[] data, int offset, int
length) {
    Lib.assertTrue(offset >= 0 && length >= 0 && offset+length <=
data.length);

    byte[] memory = Machine.processor().getMemory();

    int addOffset = Processor.offsetFromAddress(vaddr);
    int vpn = Processor.pageFromAddress(vaddr);
    int transferred = 0;
    for (int i = vpn; length>0; i++) {
        if(vpn>numPages){
            return transferred;
        }
        TranslationEntry entry = pageTable[i];
        entry.used = true;
        int phyAddr = Processor.makeAddress(entry.ppn,addOffset);
        int amount = Math.min(length, pageSize - addOffset);
        System.arraycopy(memory, phyAddr, data, offset, amount);
        addOffset = 0;
        transferred += amount;
        offset += amount;
        length -= amount;

    }

    return transferred;
    }
```

handleExec

handleExec function handles an exec system call, which executes the program stored in the specified file, with the specified arguments, in a new child process. It first checks the validity of the arguments, including the virtual address containing the file name and the two arguments containing the number of arguments and the address of the argument list, and checks if it ends with ".coff". If not, it returns -1. After that, it reads the number of arguments and the address of the argument list from the two arguments and reads all the arguments from the input. Getting a String arraylist built up with the size of the Buffer, and starting a for loop repeat the times of the buffer size given, reading each arguments. Finally, it creates a new UserProcess object with a new unique processID, adds it to the childStat which is a half implement part 3 hash table solution but failed due to time constraints and childProcesses, and executes the program with the specified file name and arguments. If the program fails to load, it returns -1, otherwise it returns the processID of the new child process.

```
    private int handleExec(int VA, int arg1, int arg2) {


        String fileName = readVirtualMemoryString(VA, 256);
//fileName is execute file???
        if (VA<0 || arg1 <0 || arg2 <0|| fileName == null) {
            return -1;
        }

        byte[] argBuffer = new byte[4];
        String[] args = new String[arg1];
        for (int i = 0; i < arg1; i++) {
            int va = Lib.bytesToInt(argBuffer, i * 4, 4);
            args[i]= readVirtualMemoryString(va, 256);
            int memReadLen = readVirtualMemory(arg2 + i * 4, argBuffer);
            if (memReadLen != 4) {
                return -1;
            }
        }

        UserProcess child = UserProcess.newUserProcess();
        childStat.put(child.processID, child);
        this.childProcesses.add(child);
        if (!child.execute(fileName, args)) {
            return -1;
```

```
        }
        return child.processID;
    }
```

handleJoin

handleJoin by its name handles the joining of a process in an operating system. It first checks if the processID and statAdd are non-negative values, and if not, returns -1. It then gets the child process associated with the processID. A process ID global integer that indicates the ID of the process, it uses processCount which adds 1 every time a new process is created. If one does not exist, returns -1. Otherwise, it joins the thread associated with the child process and writes the child process' status to the virtual memory. Finally, it returns 0 if the status is -1 and 1 otherwise.

```
    private int handleJoin(int processID, int statAdd) {
        if (processID < 0 || statAdd < 0) {
            return -1;
        }
        UserProcess child = childProcesses.get(processID);
        if (child == null) {
            return -1;
        }
        child.thread.join();
        writeVirtualMemory(statAdd, Lib.bytesFromInt(child.status));
        if (child.status == -1) {
            return 0;
        }
        return 1;
    }
```

handleExit

The handle exit function has been modified from the last project. It will unload all the sections which were included in the last project, but this time unloadSections will actually close the sections that the load section used, set the status code to a global variable which is used in the handleJoin() function, close all the fileDescriptors from the last time, and reduce the aliveProcess. AliveProcess is very similar to processID, but one will only keep adding based on the amount of processes, and aliveProcess will actually reset when there are processes closed. When the last process is closed and there are no processes left, the system will be terminated. If not, it will only finish the current thread.

```
    private int handleExit(int status) {
        unloadSections();
```

```
        this.status = status;

        for (int i =2; i<fileDescriptor.length; i++) {
            if (this.fileDescriptor[i] != null){
                handleClose(i);
            }
        }

        exitLock.acquire();
        aliveProcess--;
        if(aliveProcess == 0){
            Kernel.kernel.terminate();
        }
        exitLock.release();
        //KThread.currentThread().finish();
        UThread.finish();
        return 0;
    }
```