Homework 7
Operating Systems, V22.0202
Fall 2007, Professor Yap

Due: Dec 21, 2007.

Demand Paging in the Toy OS

This homework is our final project. It concerns the important topic of demand paging.

It is a fairly demanding (pun intended), but the fact that we have no classes next Monday and Wednesday should be used for working on this project.

You will continue to work as a team, and use the same CVS repository for storing your project. In particular, all your work for this homework should be stored in a directory called `hw7` under your TEAMCODE directory.

1 What is new in TOS3

We want you to modify the version of THE/TOS found in the CVS module `class/os/2007/stm/stm1.3`. This version is called `tos3`. You can run `tos3` by typing `make t11`. Please do this first step right away.

After you download `stm1.3`, you can explore the various sample files and targets `t1, t2, t3, t4`, etc. It has our enhanced I/O that allows string I/O (to see this, make the target called `hello`).

What is new in stm1.3 is that traps are now implemented through a trap vector. This is defined in THE Machine, but the individual traps are implemented by the Toy OS (this is as it should be). So you do not have to modify THE Machine to implement new traps. This makes it easy to implement new traps. Note that in stm1.3, we distinguish two classes of traps: system traps and user traps. System traps are numbered 0 to 19; the rest are user traps.

2 Implementing Demand Paging

Your goal is to design `tos4` which has demand paging. You can design it from scratch, but perhaps the fastest way to do this is extend `tos3` by adapting the code of Jurney and Savarin (from homework 6 and found in `stm/stm2`).

To do paging, you need a secondary memory. So, in addition to main memory (mem[ ]) we have another array called **disk**. Jurney/Savarin only implemented asynchronous memory operations. You need to also implement asynchronous disk operations.

The following are some of the key elements that you will need to think about. Note that these are general suggestions, as your own design may diverge from it.

- You will need to define some constants. Here are some typical numbers:

  #define VASIZE 262144 /* 2^{18} virtual address space */
  #define RAMSIZE 16384 /* 2^{14} main memory */
  #define DISKSIZE 524288 /* 2^{19} disk memory */
  #define PAGESIZE 128 /* 2^{7} page size */
  #define MAXFRAMES (RAMSIZE / PAGESIZE) /* frame table size */
  #define MAXPAGES (DISKSIZE / PAGESIZE) /* page table size */

- We need to declare the process table, and for each process, we need to declare a page table. There is also a page table for the current process:

```
struct pageTableEntry {
int frame, /* frame number (frame=-1 means not loaded)*/
    disk, /* relative page number in disk */
```

```
        rbit, /* read bit */
        mbit, /* read bit */
        timestamp; /* load time */
}
    struct processTableEntry {
        char name[NAMESIZE];
        int regs[NUMREGS];
        int status; /* 0=ready, 1=blocked, 2=terminated */

        struct pageTableEntry pageTable[MAXPAGES]; /* page table of process*/
        int basePage; /* first page in disk for process */
        int pageLimit; /* max number of disk page for process */ }
    struct processTableEntry processTable[MAXPROCS];
    struct pageTableEntry pageTable[MAXPAGES];
                /* page table of current process*/
```

You would want procedures to initialize a pageTable, and to load a pageTable from the processTable into the current pageTable. (For simplicity, you do not have to put this table into the mem[ ].)

- We need a FIFO called readyQueue (holding the processes ready to run).
- To keep track of the available frames, we have

```
int freeList[MAXFRAMES]; /* freeList[n]=1 is occupied, 0 if free */
```

Assume a routine to find a free frame (return -1 if no more free frames).

- We need some global variables

```
int disk[DISKSIZE]; /* disk */
int pageFault = -1; /* if pageFault is not -1, it is the
    page that must be loaded to memory */
int clock; /* system clock */
```

You need procedure to initialize the disk, and to load a process into the disk.

- To simulate asynchronous disk and memory I/O, we have a list of disk requests. Each request transfers a single page between disk and memory. We can assume that each process make at most one disk request at a time, and so this list can be a table with MAXPROCS entries.

```
    struct diskRequest {
        int diskPage; /* page number in disk */
        int memPage; /* frame number in main memory */
        int write; /* write=1 if writing, write=0 if reading */
}   struct diskRequest RequestTable[MAXPROCS];
        /* RequestTable[n] is request of Process n */
```

You need functions to (1) post a disk request, (2) process a disk request, (3) read from Disk, and (4) write to Disk.

- Here is the algorithm to convert a relative address to a physical (memory) address.

```
int physAddr( int relAddr, int writeBit);

NOTE: writeBit=1 if we want to write, writeBit=0 if we want to read

1. Make sure that relAddr is between 0 and VASIZE.  Else segment fault.

2. Initialize pageFault=-1.   (This will change if we need a page fault)

3. page = ralAddr / PAGESIZE

4. frame = pageTable[page].frame

5. Update rbit, mbit, timestamp of pageTable[page]. (N.B. mbit=writeBit)

6. if (frame = -1)
         pageFault = page;     (indicate need for page fault)
   else
        return ((frame * PAGESIZE) + (relAddr % PAGESIZE))
```

- How do we use this physAddr(..)? In the exec_instr(..) function!
  Recall that function has a switch instruction:

```
exec_instr(op, ra, ad, ...)  {
    ...
    switch (op) {
    case 0:  pa = physAddr(ad, 0); /* 0=read request */
        if (pageFault == -1)
        regs[ra] = mem[pa]; /* load register ra */
        break;
    case 1:  ...
    ...
    }//switch
    ...
}//exec_instr
```

  Note that in case of page fault, the instruction is not carried out. That is because the current process would be context switched anyway, and block while waiting for disk I/O.

- Let us now consider the page fault algorithm:

```
void execPageFault () {
        1.  frame = freeFrame(); // try to get a free frame
        2.  if (frame != -1) // found free frame!
                Update page table
                Return
        3.  Else find a frame to evict // this is the victim frame
        4.  If the mbit of victim is set,
                post a disk request to write
                post a disk request to read
                contextSwitch( 1 ) // 1=block current process
```

```
else
        post a disk request to read
        contextSwitch( 0 ) // 0=nonblock current process
```

- Note that in step 4, if the mbit of victim is set, it might appear as if you have to post two disk requests. This might appear impossible, given our `RequestTable` structure which allows each process at most one request at a time. But you can actually encode the request to write, and then to read, in the `RequestTable`. If you do not understand this, come talk to me.

3 Submission

- Your submission must include a README file telling us
  (1) what you have implemented and
  (2) what you have not implemented (but would like to).
  (3) It should also give a high level description of how you implemented various features. Specifically, I need to see
  (3a) the key data structures and what they mean
  (3b) the key functions and what they do.
- There should be a Makefile with the interesting targets for us to test!
- In particular, I would like the default target to compile the programs and run an interesting example right away.
- For Demand Paging, the targets should be "tos4" to compile TOS4, and "test" to show what your TOS4 can do.
- Remember the general rule: if it does not compile and work on simple tests, we cannot give any credit!

  So, it is more important to get your program to work on SOME easy features than to try to cover ALL the features in your project.
- I strongly suggest you stick to the data structures described in this document. Deviations from it should be explained in your README.
- We suggest that you break down the project into stages. Make sure that each stage works before you go to the next.
- At each stage, implement the simplest algorithm that wil achieve the functionality you need.

  E.g., for page eviction algorithm, choose the victim by finding the one with the oldest time stamp (basically a FIFO queue). This may not be the best algorithm, but probably simplest.
- The first stage might be called `tos4a` in which you simply adapt `tos3` to implement asynchronous memory operations from the Jurney/Savarin code. make sure that this works before going to `tos4b`, `tos4c`, etc.

4 ADDITIONAL SUGGESTIONS

The following are optional.

4.1 Implementing an accurate System Clock

You already know, from studying the the code of Jurney and Savarin in hw6, how the basic concepts asynchronous asynchronous memory operations can be implemented. There are three versions in their code, but it is enough focus on the `sos-multi.c`.

A key idea there is the notion of System Time, maintained by a `clock` that ticks with every operation (some operations causing many more ticks, especially memory operations). This clock is central to modern computers, as it is the basic many other functions.

It would be good to implement a very accurate system clock that includes both the memory and disk operations.

4.2 Implementing a Batch System

We would use THE/TOS to carry out various simulations. For this purpose, it is convenient to be able to do "batch jobs".

1. First, we want to allow stm programs to do I/O using files instead of from the terminal. To do this, `TOS` mut accept two new options at command line, indicated by `-i` and `-o`. For instance, suppose you type:

   ```
   >tos.exe -i infile -o outfile primes.stm
   ```

   Then the primes.stm program will read its input from [infile], and send its output to [outfile]. Of course, it can still take the input from the terminal if you do not specify the `-i` flag.

2. Second, we also allow the command line argument to be placed in a "batch file". You tell `TOS` where the batch file is by the flag `-b`.

   ```
   >tos.exe -b batchfile
   ```

   What should be in the batchfile? Well, each line of the batchfile indicates a program to run. For instance, it could have the following lines:

   ```
   #comment line begins with '#'
   primes.stm -i input1 -o output2
   gcd.stm -i input2 -o output2
   primes.stm -i input3 -o output3
   ```

   This would cause TOS to run three jobs.

4.3 Implementing Function Call Support

1. We want to implement 6 new traps to enable function calls:

   CALL/RET, PUSH/POP, GETSP/SETSP

   These manipulate the stack pointer register, SP. These traps are numbered 20 to 25 (so they are user traps). The specs for these traps are found in the updated Lecture 5 (in lecture directory).

   To take advantage of the SP register, you must also modify our stm loader so that it initializes SP to the maximum relative address of the stm program.

2. Write a new stml program called `gcd-rec.stm`. This is a recursive way of computing the gcd, using our new function call traps. Here is how the recursive program is to work, by repeated subtraction: assume m and n are non-negative numbers.

$$\texttt{GCD}(m,n) = \begin{cases} m & \text{if } n = 0, \quad (R0) \\ \texttt{GCD}(n,m) & \text{else if } m < n, \quad (R1) \\ \texttt{GCD}(m-n,n) & \text{else.} \quad (R2) \end{cases} \tag{1}$$

For instance,

$$\begin{aligned} \texttt{GCD}(4,6) &= \texttt{GCD}(6,4), && \text{by } (R1) \\ &= \texttt{GCD}(2,4), && \text{by } (R2) \\ &= \texttt{GCD}(4,2), && \text{by } (R1) \\ &= \texttt{GCD}(2,2), && \text{by } (R2) \\ &= \texttt{GCD}(0,2), && \text{by } (R2) \\ &= \texttt{GCD}(2,0), && \text{by } (R1) \\ &= 2. && \text{by } (R0) \end{aligned}$$

Moreover, you must assume that the arguments for this procedure is in R1 and R2 to begin. The final output (the gcd) should be in R1.

3. You must further write a driver routine in `gcd-rec.stm` to call the gcd routine.

   This driver will take read from input three numbers, m, n, k. Then it will call the gcd subroutine k times to compute:

$$\texttt{GCD}(m,n), \texttt{GCD}(m+1,n+1), \texttt{GCD}(m+2,n+2), \ldots, \texttt{GCD}(m+k,n+k).$$

4. You might also want to think about how to use these facilities to implement system libraries which user (stml) programs can invoke. You will need new traps, of course.