
mp1 warm-up: Set Up the Virtual SAPC environment

Assigned: 26 January 2023

Due: 2 February 2023 Midnight


I. OBJECTIVES:

The purpose of this assignment is to set up the virtual SAPC environment and run a simple program on the VM. Students can follow the steps shown in the class lecture02.

II. WARM-UP STEPS:

1. Download and install the corresponding version (PC, MAC or LINUX) of VMWare on your computer.
2. Follow the instructions on Blackboard for PCs or MACs to download, install and configure the tutor VM and the tutor-vserver VM on your computer.
3. SSH into users.cs.umb.edu using your UNIX credentials. Go to /nobackup/faculty/hefeiqiu/ directory and copy examples folder to your cs341 folder using:

```
cp -r /nobackup/faculty/hefeiqiu/examples/ .
```



Space between / and .
4. If you cd examples/test_dir and issue command ls, you should see the makefile and test.c
5. Follow steps in the lecture02 to build test.lnx, transfer it to tutor-vserver VM, download it to tutor VM and run it.
6. For grading purposes, create a typescript file on your PC by issuing the script command after you log into the vserver VM. This will start recording all terminal commands and responses into a file “typescript”. Run mtip, download test.lnx to the tutor VM, and run it. After running the program, issue 2 cntl-C’s and return to the LINUX prompt in vserver VM. Issue the exit command and a typescript file will be generated on your PC.
7. Since the grader has no access to your PC, you have to transfer the typescript file to your examples/test_dir/ directory at the UMB server by:

```
scp typescript your_username@users.cs.umb.edu: cs341/examples/test_dir/.
```

mp1: Understanding Tutor, a stand-alone monitor program

Assigned: 2 February 2023

Due: 23 February 2023 midnight

I. OBJECTIVES:

The purpose of this assignment is to gain some experience with our LINUX system, the Stand-Alone PCs (SAPCs), and makefiles. At the same time, you will learn how to understand and modify existing C code.

1. Read, compile, run, understand, and modify a C program that runs a monitor program "tutor". We call it tutor because it mimics the Tutor monitor/debugger that we have installed on the SAPCs. You will build a LINUX version as well as a version that runs on SAPC VM.
2. Use your monitor program to explore memory locations on the PC – including the locations in which your program resides.

II. INTRODUCTION:

You should work on this assignment alone and turn in your own individual report and source code including test results. The LINUX dates on your files will determine whether the work was completed on time or not.

You should already have a cs341 logical link in your LINUX home directory. Type in command: `cd cs341` to go into your homework directory. Use that subdirectory for all your work in this class. It has the right protection setup so that other students cannot access your files but instructors and graders can via group cs341-1G.

Copy files from `/courses/cs341/s23/hefeiqiu/mp1/` to your cs341 folder using:

```
cp -r /courses/cs341/s23/hefeiqiu/mp1/ .  
cd mp1  
ls
```

Space between / and .

You should see all the files that are supplied for this project. Let's call mp1 your project directory in what follows. Use your project directory for all work for this assignment. All files referred to below are in that project directory unless otherwise specified.

***NOTE*: YOU MUST USE THE DIRECTORIES AND FILE NAMES SPECIFIED SO THAT THE GRADERS AND I CAN FIND YOUR FILES AND TEST YOUR HOMEWORK, IF NEEDED. IF YOU DO NOT DO SO, YOU WILL BE PENALIZED AND YOUR OVERALL PROJECT GRADE WILL BE LESS.**

III. DESCRIPTION:

You will write your own "tutor" program, which mimics Tutor. It's a tiny single user terminal monitor system, which we will compile and run both on the SAPC and on the LINUX system. In later assignments you'll add to its capabilities. For now it should accept the commands:

md <hexaddress>

(SAPC and LINUX) Memory Display

Display contents (in hex) of 16 bytes at the specified address.

(Present in the same format as the “real” Tutor program – 16 pairs of hex characters followed by 16 characters interpreting each byte as a printable character for ASCII codes 0x20 through 0x7e or a fixed ‘.’ for all other (non-printable) ASCII codes values.)

ms <hexaddress> <new_val>

(SAPC and LINUX) Memory Set

Stores byte new_val (two hex characters) at specified address.

h <cmd>

(SAPC and LINUX) Help

Help on the specified command, for ex., "h ps", or all commands if no argument.

s

(SAPC and LINUX) Stop

Stop running your tutor and return to the regular SAPC TUTOR (or back to the shell if running on LINUX).

The files you need to build tutor are found in my directory /courses/cs341/s23/hefeiqiu/mp1/. Most of the program has been written for you for two reasons. First, by reading the code provided, you can learn how a standard "table-driven" command processor design works. Second, providing you with a framework allows you to concentrate on the part of the programming that's important in this course.

The main program for tutor is in tutor.c. That driver calls a lexical analyzer (parser) called slx.c. The parser finds the command on the command line and calls the appropriate procedure by consulting the command table. File cmds.c contains that table and the code that implements the various commands. The file makefile builds them into an executable file.

Start by copying all files to your project directory. The makefile will build the executable files from your local directory. Read the makefile and learn how it does that.

a) Build and run the LINUX version:

pe15: make tutor

creates an executable tutor to run on LINUX

pe15: ./tutor

runs the program locally under LINUX

pe15:

b) Build the SAPC version:

pe15: make

creates a tutor.lnx to download to the SAPC

Use the Virtualized SAPC environment to debug the SAPC version. Log in into the tutor-vserver VM using the provided credentials. Transfer the SAPC version to tutor-vserver VM using:

vserver\$ scp username@users.cs.umb.edu:cs341/mp1/tutor.* .

vserver\$ mtip -f tutor.lnx

always use board #1

Machine Project Assignment

click on the “Send Cntl-Alt-Del” button or shutdown and restart at the tutor VM to reset the SAPC VM. Hit <CR> at the tutor-vserver VM to get the tutor prompt. Then download the tutor program to the SAPC VM and run it:

```
Tutor> ~d                # download tutor.lnx
Tutor> go 100100         # start your program on SAPC VM
```

IV. MODIFY PROGRAM:

Read tutor.c, slex.h, slex.c, cmds.c, and the makefile in order to understand the structure of the tutor program. (We will be going over that structure in lecture.) When you're ready,

A. edit cmds.c to implement the required new commands (md, ms and h)

Use sscanf to convert the argument passed md or ms from a character string of hex digits to an integer. Use the return value from sscanf to check if it really worked. Study the makefile -- be sure you understand how it works. Study the output of make as it runs -- what are the program-building steps?

Sample Output

```
pe15: ./tutor
  command      help message
  md           Memory display: MD <addr>
  ms           Memory set: MS <addr> <value>
  h            Help: H <command>
  s            Stop
```

```
LINUX-tutor> md 10000
00010000 7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 .ELF.....
LINUX-tutor>
```

B. play with the tutor programs that you've built

This is a central part of the assignment. If your code is elegant and perfect but you do not test it, exercise it, and think about the results it generates you will have learned less than half of what you should - so will get less than half the credit.

You must create a file discussion.txt. Write clear, grammatically correct English. Consider showing drafts to your friends (or your English teacher) to find out if your writing is clear.

V. DISCUSSIONS:

In your discussion.txt file, answer all of the following questions:

- (1) Describe how you tested your code. Try the following experiments. Write several sentences about what you found or learned working on each, and on any other similar kinds of questions that occur to you.

What happens if you call md for an address that does not correspond to a physical memory address? What if you write to an address that's part of your tutor code? Do these questions have the same answers on LINUX and the SAPC?

(2) Read the makefile to see where it puts the symbol table (nm output) for your tutor code. Use that symbol table to figure out:

(a) the address for test global variable xyz, which has value 6. Use tutor with that address to verify the value in memory.

(b) the address of the pointer variable pxyz. This address should be close to the one you determined in a, but not equal to it, since pxyz is the next variable in memory after xyz. Find the value of pxyz in memory. This should be equal to the address you found in (a) because of the initialization of this variable to &xyz. Note that you need to get 4 bytes of data for the value here. See IMPORTANT NOTE just below.

(c) the address of the cmds array. Use this address to determine the very first pointer in the array, the string pointer to "md". Then find the 'm' and 'd' and terminating null of this string.

(d) change the stop command from 's' to 'x' while the tutor program is running. Can you change the tutor prompt the same way?

IMPORTANT NOTE: When you try to access a 32-bit value (a pointer for example) in SAPC memory via Tutor, you need to reverse the displayed byte order, because of the little-endian representation of numbers in the Intel architecture.

Example: Suppose a pointer (or any 32-bit numeric quantity) is at address 0x100200 with value 0x00100abc. It would show up as:

```
Tutor> md 100200
100200  bc 0a 10 00
```

because of "little-endian" storage of numbers in memory in the Intel architecture. See the lecture notes.

To help with displays, there is an "mdd" command (memory-display-doubleword) in the "real" Tutor monitor. This command reorders the bytes for you and displays four bytes at a time as a doubleword value:

```
Tutor> mdd 100200    (for same memory contents as with md command above)
100200  00100abc ...
```

(3) Read the nm output to determine where in memory the code resides, on SAPC and LINUX. Hint: code symbols are marked t or T. Similarly determine where the data (set of variables) resides.

Machine Project Assignment

(4) Try to change the code itself so that tutor crashes (any random change that actually takes effect should do this). What happens on SAPC? on LINUX?

(5) You can't find the program stack using the nm output, but you can find it by looking at the stack pointer, called %esp on the SAPC and LINUX. Record your observations. Use "i reg" (info on registers) to see %esp in gdb and "rd" to see registers in Tutor.

(6) (Extra credits) More questions you should consider answering. What other interesting things have you tried? What did you learn from this project? Was it worth the time it took? What parts were the hardest, what parts the easiest, what parts most surprising, most interesting? What idiosyncrasies of C or LINUX or the SAPC or our installation slowed you down or helped you out? How might the assignment be improved?

VI. TURN-IN FOR GRADING:

To generate a typescript file, execute the LINUX command "script". This will start recording all terminal commands and responses into a file "typescript". At a minimum, you must show all of the following in your typescript file:

- ls -al to show your user name
- cat your discussion.txt file
- cat the sources of all of your .c files
- execution of make clean
- execution of make to create all program executable files
- a sample run of each test case that is required in the assignment

NOTE: In your typescript file, as the first step, you are required to include a fake command using your "first_name last_name your_unix_account_username", all lower case. Be aware of the space in between. This fake command will fail for sure. But it serves the purpose of showing both your name and the unix account username. The submission will ONLY be graded when there is this information.

Leave your discussion.txt file, typescript file, .c files, and a working version of tutor and tutor.lnx in your mp1 directory for grading. Do not modify any of these files after you turn in. The graders or I will pick up your homework from the directory and grade it. We may also login to this directory to check the LINUX dates on these files and to run them or test them ourselves. We may recompile your cmds.c with another main program as an extra test case. Please remember to close your typescript file by issuing the command "exit".

A rubric sheet for grading mp1 is included. In the event that you are unable to correctly complete the entire assignment by the due date, do not remove the work you were able to accomplish. Submit what you have completed - partial credit is always better than none.

mp2 Warmup Instructions

Study the lecture notes on the tools and instruction set. Then follow along with this document. Make sure everything works for you as it is shown here and that you understand *everything*. Turn in your work on this "warmup" along with the rest of your mp2 assignment in the cs341/mp2 folder. Requirements for turn-in are described under the section of TURN-IN FOR GRADING in mp2 instruction.

Here's your first assembler program. It is written in Intel assembly language:

```
        .globl _start
_start:
    movl $8, %eax
    addl $3, %eax
    movl %eax, 0x200
    int $3
    .end
```

I've added the "int \$3" to trap back to Tutor at the end. Note also that I have used the .end to tell the assembler that this is the end of the code to be assembled.

Let's see how to get this to run on the tutor VM. Since it only uses registers and a memory location, it doesn't need any "startup" module. We just have to get these instructions into memory and execute them. Steps are as follows:

1. You can find tiny.s in mp2/warmup/

Copy the entire mp2 directory including the warmup one to your cs341 folder using:

```
cp -r /courses/cs341/s23/hefeiqiu/mp2 .
cd mp2/warmup
```

2. Build a 32-bit executable

Build a 32-bit executable by running the assembler as --32 and then the loader ld -m elf_i386. Normally we would put these commands in a makefile, but here you want to become familiar with the individual steps.

```
-----
pel15$ as --32 -al -o tiny.o tiny.s

1          # tiny.s: mp2warmup program
2
3          .globl _start
4  _start:
5 0000 B8080000    movl $8, %eax
5      00
6 0005 83C003     addl $0x3, %eax
7 0008 A3000200   movl %eax, 0x200
7      00
8 000d CC        int $3
9          .end
```

```
pe15$ ld -m elf_i386 -N -Ttext 0x100100 -o tiny.lnx tiny.o
```

Here the -N flag tells ld to make a self-sufficient, simple executable, and the "-Ttext 0x100100" tells it to start the code area at 0x100100

3. We can look at the contents of tiny.lnx with objdump

To get the hex contents as well as the disassembly, use "-S" option:

```
pe15$ objdump -S tiny.lnx
```

```
tiny.lnx:      file format elf32-i386
```

Disassembly of section .text:

```
00100100 <_start>:
 100100:      b8 08 00 00 00      mov     $0x8,%eax
 100105:      83 c0 03            add     $0x3,%eax
 100108:      a3 00 02 00 00      mov     %eax,0x200
 10010d:      cc                int     $3
```

From the disassembled output, we can tell:

```
b808000000      is at locations starting at 0x100100; mov is 5 bytes long
83c003          is at locations starting at 0x100105; add is 3 bytes long
a300020000      is at locations starting at 0x100108; mov is 5 bytes long
cc              is at location 0x10010d; int is 1 byte long
.end            program ends at location 0x10010e
```

Later, we will cover how to encode instructions in bits, but for now it is interesting to find the 0x200 address hidden in the movl %eax, 0x200 instruction, and the 08 and 03 in the first two. Surprisingly, the 08 takes up 4 bytes but the 03 only one. The instruction set is optimized to be able to add small numbers into registers very quickly. The instruction size is important to speed because each instruction must be read out of memory before it can be executed.

4. Run tiny.lnx and use tutor to debug program

We download and run tiny.lnx on the tutor VM, executing one instruction at a time to see how the registers change. To execute one instruction at a time, use the "t" command in Tutor, for "trace". To get started, set the EIP to 100100, pointing the CPU to address 100100 as the next instruction to execute.

Logon to tutor-vserver VM using credentials provided. Transfer the tiny.lnx file from users.cs.umb.edu to the VM using scp:

```
tutor-vserver$ scp username@users.cs.umb.edu:cs341/mp2/warmup/tiny.*
tutor-vserver$ ls
```

space



You should see all the tiny.* files. Download the tiny.lnx file from tutor-vserver VM to the tutor VM using mtip:

```
tutor-vserver$ mtip -f tiny.lnx
For command help, type ~?
For help on args, rerun without args
Code starts at 0x100100
Using board # 1
(restart tutor VM and hit <CR> at vserver VM)
```

```
Tutor> ~downloading tiny.lnx          //enter ~d
.Done.
Download done, setting eip to 100100
```

```
Tutor> md 100100                      //Look at the code: same as above
00100100  b8 08 00 00 00 83 c0 03 a3 00 02 00 00 cc 90 90 .....
```

```
Tutor> go 100100
Exception 3 at EIP=0010010e: Breakpoint
```

```
Tutor> rd
EAX=0000000b EBX=00009e00  EBP=000578ac
EDX=00101b88 ECX=00101bac  ESP=003ffff0
ESI=00090800 EDI=00101d5c  EIP=0010010d
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
```

```
Tutor> md 200                        //Check target area using md or mdd
00000200  0b 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```
Tutor> ms 200 00000000                //Clear target area(8 0's for 32-bit write)
```

```
Tutor> md 200                        //Check again--OK
00000200  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```
Tutor> rs eip 100100                 //Set initial EIP to start addr
```

```
Tutor> t                             //Trace: execute 1 instruction
```

```
Exception 1 at EIP=00100105: Debugger interrupt
Tutor> rd                            //See EIP at 100105 (i.e. offset 5), and
EAX=00000008 EBX=00009e00  EBP=000578ac    //8 now in EAX
EDX=00101b88 ECX=00101bac  ESP=003ffff0
ESI=00090800 EDI=00101d5c  EIP=00100105
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
```

```
Tutor> md 200                        //Check target area: nothing yet
00000200  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```
Tutor> t                             //Execute 2nd instruction
```

```
Exception 1 at EIP=00100108: Debugger interrupt
Tutor> rd                            //See b in EAX, EIP to offset 8
EAX=0000000b EBX=00009e00  EBP=000578ac
EDX=00101b88 ECX=00101bac  ESP=003ffff0
ESI=00090800 EDI=00101d5c  EIP=00100108
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
```

```
Tutor> md 200                        //Check target area: nothing yet
00000200  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```
Tutor> t                             //Execute 3rd instruction
```

```
Exception 1 at EIP=0010010d: Debugger interrupt
Tutor> rd                            //Only EIP has changed in regs
EAX=0000000b EBX=00009e00  EBP=000578ac
EDX=00101b88 ECX=00101bac  ESP=003ffff0
ESI=00090800 EDI=00101d5c  EIP=0010010d
EFLAGS=0302 (IF=1 SF=0 ZF=0 CF=0 OF=0)
```

```
Tutor> md 200                        //Check mem--yes, 0b now in 0x200
00000200  0b 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

```

Tutor> t                                     //Execute int $3
Exception 3 at EIP=0010010e: Breakpoint
Tutor> ~q
Quit handler:
Killing process xxxx Leaving board #1
Tutor-vserver$
-----

```

5. Run tiny.lnx and use remote gdb to debug program

Try out remote gdb on tiny: See details in part 6 of VMWare-for-Tutor_PC_2022.pdf for PCs (or VMWare-for-Tutor_MAC_2022.pdf for MACs). For the VM environment, COM1 is for remote gdb and COM2 is for the console.

```

-----
At the tutor-vserver VM, enter:
Tutor-vserver$ mtip -f tiny.lnx  (always use board #1)
For command help, type ~?
For help on args, rerun without args
Code starts at 0x100100
Using board # 1
(hit <CR> here)

Tutor> ~d
.Done.
Download done, setting eip to 100100
Tutor> gdb
Setting gdb dev to COM1, starting gdb (CTRL-C to abort).
<---just let it hang here

```

In another window in your home computer, run putty in PC or ssh in MAC. Connect to the tutor-vserver VM's IP address. Logon to tutor-vserver VM using the same credentials provided. Enter the following in the ssh window:

```

Tutor-vserver$
Tutor-vserver$ gdb tiny.lnx
GNU gdb (GDB) 7.0.1-debian
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/tuser/cs341/mp2/warmup/tiny.lnx...(no debugging
symbols found)...done.

(gdb) tar rem /dev/ttyS0          <--set gdb to talk to COM1(ttyS0)
Remote debugging using /dev/ttyS0
0x00100100 in ?? ()
(gdb) set $eip=0x100100          <--set PC to point at 0x100100

(gdb) i reg
eax                0xb                11
ecx                0x6a894            436372

```

```

edx          0x0          0
ebx          0x9e00       40448
esp          0x578a8      0x578a8
ebp          0x578ac      0x578ac
esi          0x90800      591872
edi          0x51ffc      335868
eip          0x100100     0x100100
ps           0x302        770
cs           0x10         16
ss           0x18         24
ds           0x18         24
es           0x18         24
fs           0x18         24
gs           0x18         24

(gdb) x/x 0x200
0x200:      0x00000abc      <--old contents of memory at 0x200
(gdb) set *(int *)0x200 = 0 <--how to "ms" with gdb
(gdb) x/x 0x200
0x200:      0x00000000      <--check results
(gdb) set $eip = 0x100100   <--to run from start
(gdb) x/4i 0x100100         <--examine 4 instructions
0x100100 <tiny.o>:      movl    $0x8,%eax
0x100105 <tiny.o+5>:    addl    $0x3,%eax
0x100108 <tiny.o+8>:    movl    %eax,0x200
0x10010d <tiny.o+13>:   int3
(gdb) b *0x100105           <--set breakpoint at 2nd instruction
Breakpoint 1 at 0x100105
(gdb) c                     <--continue from 0x100100
Continuing.

Breakpoint 1, 0x00100105 in _start ()
(gdb) i reg
eax          0x8          8
ecx          0x6a894      436372
edx          0x0          0
ebx          0x9e00       40448
esp          0x578a8      0x578a8
ebp          0x578ac      0x578ac
esi          0x90800      591872
edi          0x51ffc      335868
eip          0x100105     0x100105
ps           0x216        534
cs           0x10         16
ss           0x18         24
ds           0x18         24
es           0x18         24
fs           0x18         24
gs           0x18         24
(gdb) b *0x100108
Breakpoint 2 at 0x100108
(gdb) c
Continuing.

Breakpoint 2, 0x100108 in _start ()
(gdb) i reg
eax          0xb          11

```

ecx	0x6a894	436372
edx	0x0	0
ebx	0x9e00	40448
esp	0x578a8	0x578a8
ebp	0x578ac	0x578ac
esi	0x90800	591872
edi	0x51ffc	335868
eip	0x100108	0x100108
ps	0x202	514
cs	0x10	16
ss	0x18	24
ds	0x18	24
es	0x18	24
fs	0x18	24
gs	0x18	24

(gdb) b *0x10010d

Breakpoint 3 at 0x10010d

(gdb) c

Continuing.

Breakpoint 3, 0x10010d in tiny.o ()

(gdb) i reg

eax	0xb	11
ecx	0x6a894	436372
edx	0x0	0
ebx	0x9e00	40448
esp	0x578a8	0x578a8
ebp	0x578ac	0x578ac
esi	0x90800	591872
edi	0x51ffc	335868
eip	0x10010d	0x10010d
ps	0x302	770
cs	0x10	16
ss	0x18	24
ds	0x18	24
es	0x18	24
fs	0x18	24
gs	0x18	24

(gdb) x/x 0x200

0x200: 0x0000000b

(gdb) q

The program is running. Quit anyway (and kill it)? (y or n) y

Tutor-vserver\$

Note: To everyone who may encounter this problem and ask:

Question: Why am I getting these error messages?

```
itserver6$ cat tiny.s
```

```
# tiny.s
```

```
# mp2 Warmup
```

```
    movl $8, %eax
    addl $3, %eax
    movl %eax, 0x200
    int $3
.end
```

```
itserver6$ as --32 -o tiny.o tiny.s
```

```
tiny.s: Assembler messages:
```

```
tiny.s:4: Error: Rest of line ignored. First ignored character valued 0xd.
```

```
tiny.s:5: Error: invalid character (0xd) in second operand
```

```
tiny.s:6: Error: invalid character (0xd) in second operand
```

```
tiny.s:7: Error: invalid character (0xd) in second operand
```

```
tiny.s:8: Error: invalid character (0xd) in first operand
```

```
tiny.s:9: Error: Rest of line ignored. First ignored character valued 0xd.
```

Answer:

You must have used an editor such as notepad on your PC locally to create the .s file and used file transfer to put it on the LINUX system. Notepad has put a carriage return (CR) character 0x0d at the end of each line in addition to the normal LINUX new line (NL/LF) character 0x0a.

Here is an octal dump of the ASCII characters in hex form that are in your source file:

```
itserver6$ od -x tiny.s
```

```
0000000 2320 7469 6e79 2e73 0d0a 2320 4761 6c69
```

```
0000020 6e61 204f 736d 6f6c 6f76 736b 6179 610d
```

```
0000040 0a23 206d 7032 2057 6172 6d75 700d 0a0a
```

```
0000060 2020 206d 6f76 6c20 2438 2c20 2565 6178
```

```
0000100 0a20 2020 6164 646c 2024 332c 2025 6561
```

```
0000120 780a 2020 206d 6f76 6c20 2565 6178 2c20
```

```
0000140 3078 3230 300a 2020 2069 6e74 2024 330a
```

```
0000160 2020 2e65 6e64 0a00 0000167
```

```
itserver6$
```

Notice the 0d0a character sequence that occurs at the end of each line.

The GAS assembler (as --32) is not ignoring the carriage return character 0x0d at the end of each line and it gives an error. To fix this problem, you can use an LINUX editor such as vi to remove the carriage return (CR) characters or you can use the LINUX command tr to remove the 0x0d (or octal 15) characters and the command mv to rename the output file to the original one:

```
itserver6$ tr -d '\015' <tiny.s >output_file
```

```
itserver6$ mv output_file tiny.s
```

mp2: i386 C-callable Assembly Language Functions

Assigned: 2 March 2023

Due: Warmup + Part 1: 10 March 2023 midnight
Parts 2+3 : 26 March 2023 midnight

I. OBJECTIVES:

The purpose of this assignment is to gain some familiarity with writing C callable functions that use assembly language. Copy all files from /courses/cs341/s23/hefeiqiu/mp2/part1/ and /courses/cs341/s23/hefeiqiu/mp2/part2+3/ to your cs341/mp2/ directory. Use the provided makefile for all builds except where instructed otherwise. As in mp1, use the environment in users.cs.umb.edu to build all your .lnx executables and use the tutor-vserver and tutor VMs to run and debug your code.

II. DESCRIPTION:

Part 1: Program to count the occurrence of a user entered character in a string

1. Write a C callable assembler function that counts the number of characters in a string. The function should work for any string and character. The address of the string and character to be counted are passed as arguments according to the C function prototype:

```
int count (char *string, char c);
```

Since there are input arguments, your assembly function should use a stack frame. It should return the count to the calling C program in %eax. Use 32-bit quantities for all data. Even though chars have only 8 bits, they are stored in memory (and on the stack) as 32 bits in “little endian” format. If the 32-bit value is moved from memory to a register, the char value is available in the 8 least significant bits of the register.

You are given a C calling program (countc.c) in mp2/part1/ that calls the count function to count the number of times a user-entered character appeared in a user-entered string and prints the result. The C code “driver” is in countc.c. Create a new file with the name count.s and put your assembly code in count.s. Then build the executable using the provided makefile by invoking “make A=count”.

Capture a run of your program in the script file mp2_part1_typescript. The file is captured in the vserver VM and later transferred to your cs341/mp2/part1/ homework directory for grading. Please provide the script showing a run with a breakpoint set using either Tutor or remote gdb where the count is incremented, showing the count (in a register) each time the breakpoint is hit, just before the increment is made. In the script, show how you determined where to set the breakpoint, i.e., how you determined where the increment instruction is located in memory.

Part 2: Program to print binary characters

2. You are given a C calling program (printbinc.c) in mp2/part2+3/. Call your file printbin.s. The C caller for this program provides a char to the assembly function printbin. It asks the user for a

hex number between 0 and 0xff, converts the input from an ASCII string to a char value, passes it to `printbin` as an argument, and displays the returned string which should be the ASCII characters for the binary value, e.g. for entry of the hex value 0x3d, you will get the printout: “The binary format for character = is 0011 1101” You can see the function prototype for the `printbin` function in the calling C code.

The function `printbin` should be C callable using a stack frame and call an assembly language subprogram “`donibble`” that is not required to be C callable. Avoiding the use of stack frames is one way assembly code can be more efficient than C compiler generated code. The function `printbin` needs to declare space in the `.data` section for storage of a string to be returned and return the address of that location in the `%eax`. While processing the bits of the input argument, keep a pointer to the string in an available register. `printbin` and `donibble` can store an ascii character 0x20, 0x30, or 0x31 in the string indirectly via that register and then increment the pointer in that register until the entire return string has been filled in.

The `donibble` function handles one half of the char value producing the four ASCII character (0/1) for the bits in one hex digit. `Printbin` should call `donibble` twice once with each nibble to be processed in half of an available register, e.g. the `%al`. `Donibble` should scan the 4 bits of the register and move an appropriate ascii code into the string for each bit. `Printbin` adds the space between the two nibbles.

In file `mp2_part2_typescript`, show a run of `printbin` on the tutor VM to display 0xab, and then a run with a breakpoint set at the helper function, to prove that it is called exactly twice. You can use Tutor or remote gdb here as you wish.

Part 3: Program to copy n characters of a string

3. Write a C-callable assembler version of the library `strncpy` function called `mystrncpy` (to avoid conflicts with the library version) that copies the contents of one string to a user provided array and returns a pointer to the provided array. The function prototype is:

```
char *mystrncpy(char *s, char *ct, int n);
```

Write your code in a source file named `strncpy.s`. The provided C driver (`strncpyc.c`) in `mp2/part2+3/` takes user entered input for the source string and checks both the pointer returned and the effect of the copy. Choose test cases that exercise different possibilities in the logic of your code, e.g. a null string. What would happen if you choose a string longer than the destination array in the C driver?

In file `mp2_part3_typescript`, show a run of `strncpy` on the tutor VM to display the source string copied and the pointer returned.

III. TURN-IN FOR GRADING:

For `mp2_warmup`, you need to create a typescript file with the name `mp2_warmup_typescript`. It needs to show the work you did following all the steps in the instructions.

You need to create a typescript file for each part of mp2. Name them mp2_part1_typescript, mp2_part2_typescript, mp2_part3_typescript.

On users.cs.umb.edu, capture the execution of following commands: pwd, ls -lg, cat xxx.s, and the make build for the .lnx file. On tutor-vserver, capture outputs from the following commands: scp transfer of the .lnx file to the VM and mtip to run the .lnx file on tutor. After completing each part, use scp to transfer the script files to your mp2 directory at users.cs.umb.edu.

Leave working versions of the source files and the script files in your mp2 project directory. The grader or I may rebuild them and/or run them to test them. A copy of the rubric sheet for grading mp2 is included. In the event that you are unable to correctly complete this assignment by the due date, do not remove the work you were able to accomplish - partial credit is always better than none.

mp3: Accessing I/O Ports using assembly language

Assigned: 28 March 2023

Due: 11 April 2023 midnight

I. OBJECTIVES:

The purpose of this assignment is to gain some familiarity with writing C callable functions that do things that can't be done in C such as accessing I/O ports. Copy all files from /courses/cs341/s23/hefeiqiu/mp3 to your mp3 subdirectory of cs341. Use the provided makefile for all builds except where instructed otherwise. As in mp1 and mp2, use users.cs.umb.edu to build all your executables and use the VMs to run and debug your code.

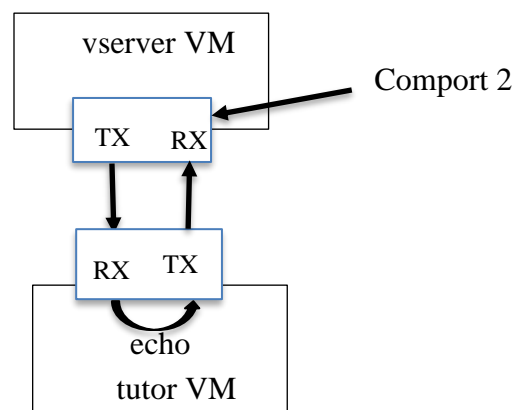
II. Program a UART directly in assembler

Please note that we consider this the old-fashioned way to do this programming. The current way is to use assembler only where it is required. Write an assembler function to read and echo characters. The C driver does the same initial logic determining whether or not the console is a COM port and if so which one, and if not, just return. In the new version, it then goes on to ask the user for the "escape character" that stops the echo loop. This can be CONTROL-A, but it doesn't have to be. The user types the actual character, not its ASCII code or whatever, to specify the escape character. Then the C driver calls the assembler echo function:

```
void echo(int comport, unsigned char esc_char)
```

The assembler code does the loop with in and out instructions to do the actual I/O, testing for the special esc_char and returning when it is seen coming from the user. Call your files echo.s and echoc.c. In file echo_typescript, show:

- 1) a run with "abc" followed by the esc_char,
- 2) a second run with "abcde<CR>xy" followed by the esc_char, and
- 3) a third run with "abcde<CR><LF>xy", then esc_char.



<CR> stands for carriage return which is the "Enter" key on your keyboard. <LF> stand for line feed. If your keyboard doesn't have a key marked line feed, use <control-J>. Since we haven't

asked for any special treatment for <CR>, the "xy" of the second test should overwrite the "ab", but the <LF> should prevent this in the third test.

FINAL NOTE:

Leave working versions of the source files and the script files in your mp3 project directory. The grader or I may rebuild them and/or run them to test them. A copy of the rubric sheet for grading mp3 is included. In the event that you are unable to correctly complete this assignment by the due date, do not remove the work you were able to accomplish - partial credit is always better than none.

mp4: Timing Interrupts

Assigned: 13 April, 2023

Due: 25 April, 2023, Midnight

I. OBJECTIVES:

The purpose of this assignment is to build a package of timing routines for the tutor VM and use it to time code on the tutor VM. This code does not run on UNIX (pe15). Read the Intel 8254 Programmable Interval Timer (PIT) Data Sheet which is under Start Here ---> Resources on Blackboard. Look at cs341/examples/timer_dir/timer.c for an example of C code for accessing the timer (suppose you have downloaded the examples folder to your course directory). Copy all the files needed for this project from /courses/cs341/s23/hefeiqiu/mp4/.

II. The timing package:

A package is a set of utilities that can be called by a program that wants to use them. When designing a package we carefully consider the things in it that must be visible to the caller, specify those, and require that everything else be internal and invisible. The visible parts of the timing package you will build are in "timepack.h" (function prototypes and comments are listed in the timing directory cs341/examples/timer_dir/). A customer for the package's services includes that header file (using #include) in the program for compilation and links his or her object code with "timepack_sapc.o" to build the executable. You are asked to modify "timepack_sapc.c" to provide a higher resolution timer service for programs running on the tutor VM.

a) Part 1: Existing Code

Every package should have a test program showing that it works by calling it. This test program is called a "driver" because it sits on top of the package and drives it like we test-drive a car – start up, do this, do that, stop, shut down. It is also called a "unit test" program because it tests just this one package separate from any other package in a bigger program. If you suspect something is wrong in a certain package, you'll try to make its unit test fail, and then you debug the problem in the relatively simple environment of the unit test, rather than in the bigger program.

The test programs for timepack is measure.c and charGen.c. You can build measure and measure.lnx using the makefile provided. File transfer the measure.lnx and the charGen.c file from the course directory to your vserver VM. **Build the charGen program on vserver VM using:**

```
gcc -o charGen charGen.c
```

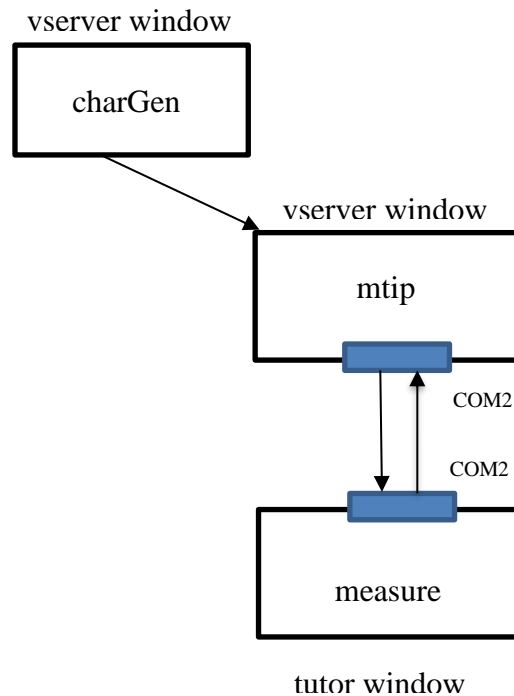
Load the measure.lnx into the tutor VM using mtip and run the program using:

```
go 100100
```

Open up a remote ssh window, connect to vserver VM, and run charGen to generate test characters to COM2 using:

```
sudo ./charGen
```

Use cs444 as your sudo password. Capture this in a typescript1 file (see an example on Blackboard). It shows that the timing package (as provided) can time things on the tutor VM to 55-ms accuracy, but not to the microsecond accuracy we want. The next step is to get your `timepack_sapc.c` fixed up for the higher resolution and the unit test executable `measure.lnx` will show it.

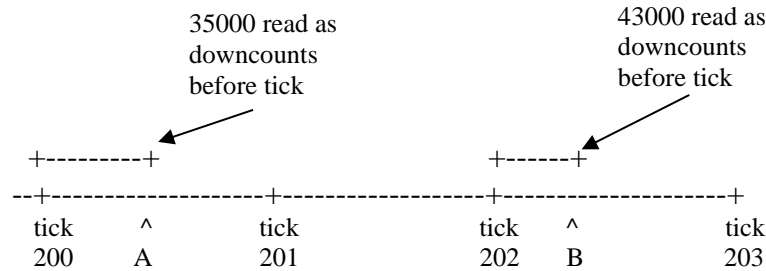


b) Part 2: Modified Code:

You're working directly with the hardware device, the Programmable Interval Timer (PIT) with its interrupt handler. This code has been provided to you in a). The `timepack_sapc.c` as provided can measure time in timer "ticks" at an 18.2-Hz (55 ms/tick) standard PC tick rate. To use the PIT to measure higher precision, you make use of "downcounts" within the timer chip. What you need to do is determine how many counts have down counted in the timer *since* the last tick and compute a higher accuracy time. By doing this at both the start and the end of the time interval being measured, you can compute the elapsed time accurate to a few microseconds, a very respectable timer service. You'll need to modify the `timepack_sapc.c` to achieve this.

There are $64K = 64 \cdot 1024$ downcounts within 1 tick and the time for one downcount to take place is _____usec. You will use this value later for your calculations.

Example: Event A happens 35000 downcounts before tick 201
Event B happens 43000 downcounts before tick 203



Since timer downcounts count down from 64K at the tick, you need to subtract the register value from 65536 to get the number of downcounts since last tick:

$$\text{number of downcounts since tick} = 65536 - \text{observed_count (in register)}$$

Thus the accurate time between A and B is

$$\begin{aligned} &= 202 \text{ clock ticks} + (65536 - 43000) \text{ downcounts} \\ &\quad - (200 \text{ clock ticks} + (65536 - 35000) \text{ downcounts}) \\ &= 2 \text{ clock ticks} - 8000 \text{ downcounts} \end{aligned}$$

where 1 tick = 64×1024 downcounts, so this whole thing can be expressed in downcounts, and then converted to usecs.

Note that you need to *convert* downcounts (an int) to usecs (another int). To do this, cast the downcount value to double, multiply by the correct double conversion factor, and then cast back to int usecs. The idea here is that the conversion factor is itself non-integral, so the needed multiplication by it needs to be in floating point, resulting in a floating point number accurate to about 1 usec, so we might as well cast it back to an int value since ints are easier to work with.

See the timer.c in the examples/timer_dir/ directory for a sample C program that reads the timer downcounts – you can take code from this as needed.

Don't leave any printf's in your final code that output during timing runs!! They take a really long time and ruin the data. Save the timing figures in memory and print them out after the stoptimer call.

Any private functions or variables you need should be declared static in "timepack_sapc.c", so that the calling program can never accidentally use a variable or function with the same name, or interfere with yours. Capture your improved result in a typescript2 file.

FINAL NOTE:

Leave working versions of the source files and the typescript files in your mp4 project directory. The grader or I may rebuild them and/or run them to test them. A copy of the rubric sheet for grading mp4 is included. In the event that you are unable to correctly complete this

assignment by the due date, do not remove the work you were able to accomplish - partial credit is always better than none.

Machine Project Assignment

mp5: Watchdog Timer

Assigned: 25 April 2023

Due: 10 May 2023 End of day

I. OBJECTIVES:

Embedded systems use a watch dog timer to recover from program malfunctions. During faulty software operations (e.g. a program gets stuck in an infinite loop), expiration of a watchdog timer causes the system to reboot. In embedded applications, these timers can be implemented in either software or hardware. This assignment teaches students on how to implement a watchdog timer in software. Copy files for this exercise from /courses/cs341/s23/hefeiqiu/mp5/ to your cs341/mp5 directory. You will use timing functions provided in timepack_sapc.c.

II. Operations of a Watchdog Timer:

One can imagine if an embedded system installed in a remote area goes haywire, it may be impossible or very costly to send a service personnel to reset the system. See an example application on the [NASA Deep Space Probe](#) and a more recent one on the [Mars helicopter flight](#). A watchdog timer can automatically reset the system without human intervention in the event of a system failure.

In theory, a [watch dog timer](#) is a timer that is set to expire at a pre-determined time interval. During a faulty condition (e.g. software is stuck in an infinite loop), timer expiry reboots the entire system. If the watchdog timer is implemented in external hardware, its logic will send a signal to the reset circuit (e.g. power up restart) of the embedded system when the timer expires. If implemented in software, the interrupt service routine of the timer will run a reboot function that restarts the application from scratch.

Since rebooting an application causes a major interruption to the operation of the system (similar to the powering up your computer), you don't want to reboot it if this is not needed. In most cases, we don't know exactly when the problem will occur, but we may have some estimate on the time when the problem will likely to occur. Then you set the watchdog timer to expire at or a little after that estimated time. For those times that the software is running without fault, you reset the watchdog timer before it expires to prevent interrupt from happening. This action is known as "kicking the dog"

III. Software implementation of a Watchdog timer:

a) Part 1: Simulating the coding problem:

We use a similar test set up as in mp4. The test program for this mp is charGen1.c and is provided in the cs341/mp5/ directory. scp the file to your vservers VM and **build the charGen1 executable at the vservers VM** using:

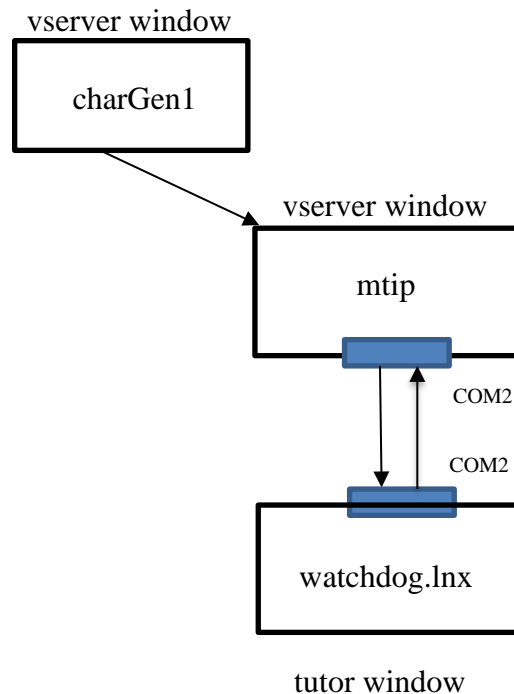
```
gcc -o charGen1 charGen1.c
```

Machine Project Assignment

Build the software application watchdog.lnx on users.cs.umb.edu (pe15) using the provided makefile, watchdog.c, reboot.s and timepack_sapc.c files. scp the watchdog.lnx to your vserver VM and load it into the tutor VM using mtip and ~d. Run the programs in this sequence:

- i) Start the watchdog.lnx using tutor command: go 100100
- ii) Open up a remote ssh window at vserver VM and run charGen1 to generate test characters to COM2 at 1 second intervals using:

```
sudo ./charGen1  
(use cs444 as your sudo password)
```



The provided code in watchdog.c will print out characters received from the charGen1 driver. After 10 characters, the system will get stuck in a simulated infinite loop and no characters will be printed out. However, the serial port interrupts are ongoing. The program continues to execute the serial port ISR and process the received characters, but the main() is not printing them out.

Capture a run of your program in the tutor VM with a typescript file named script_part_1.

Machine Project Assignment

b) Part 2a: Reset the system using a watchdog timer

Code the watchdog timer using functions provided in `timepac_sapc.c`. Set the count value = 0 to generate an interrupt every 55msec (this is longest duration between interrupts you can set with a 16-bit counter hardware). When the watch dog timer expires, the ISR calls a reboot function (the provided `reboot.s`) that restarts the `watchdog.lnx` program. If you want to wait a longer time to do `reboot()`, you have to modify the ISR software as follows:

```
void irq0inthandc()
{
    ....
    tickcount++;
    if (tickcount % 300 == 0) reboot(); // this will do reboot every 300* 55ms
    ...
}
```

Cat your programs (`watchdog.c`, `timepack_sapc.c` and `reboot.s`) and capture the run of your program in the tutor VM with a typescript file named `script_part_2a`.

c) Part 2b: Implement the “Kick the dog” function

In `watchdog.c`, at the end of the `do_work()` function, implement a `kick_dog()` to call the `set_timer_count` function to reset the count back to 0. This will prevent the timer from expiring under normal circumstances. If the program gets stuck in the middle of `do_work()` and `kick_dog()` is not run, then the timer will expire and the program will get rebooted at the timer ISR.

Cat your programs (`watchdog.c`, `timepack_sapc.c` and `reboot.s`) and capture the run of your program in the tutor VM in a typescript file named `script_part_2b`.

Any private functions or variables you need should be declared static in "`timepack_sapc.c`", so that the calling program can never accidentally use a variable or function with the same name, or interfere with yours.

FINAL NOTE:

Leave working versions of the source files and the script files in your `mp5` project directory. The grader or I may rebuild them and/or run them to test them. A copy of the rubric sheet for grading `mp5` is included. In the event that you are unable to correctly complete this assignment by the due date, do not remove the work you were able to accomplish - partial credit is always better than none.