CS 1550

Week 2 – Setting up for Project 1

& Lab 1 cont.

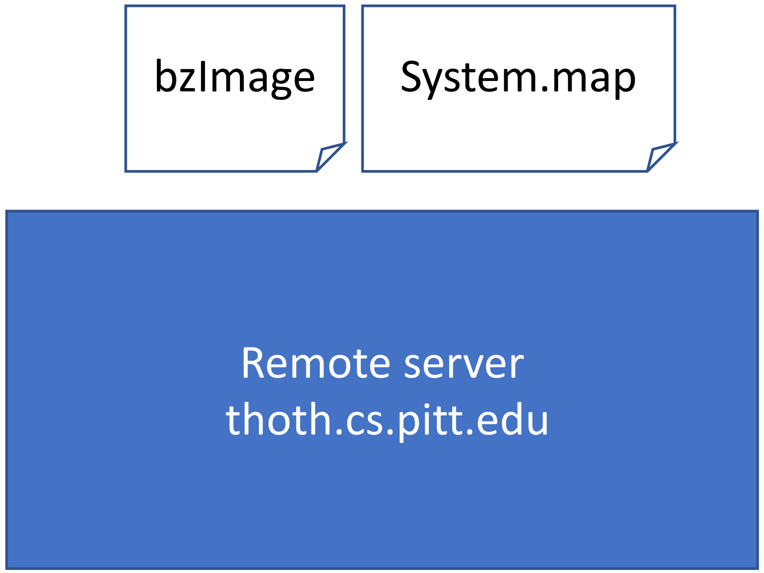
TA: Jinpeng Zhou jiz150@pitt.edu

Setting up environment for Project 1

- We will modify **Linux** source code and build a Linux distro on thoth server, then run & test our build on a VM

Setting up environment for Project 1

- We need to build the new kernel **on thoth server (thoth.cs.pitt.edu)**

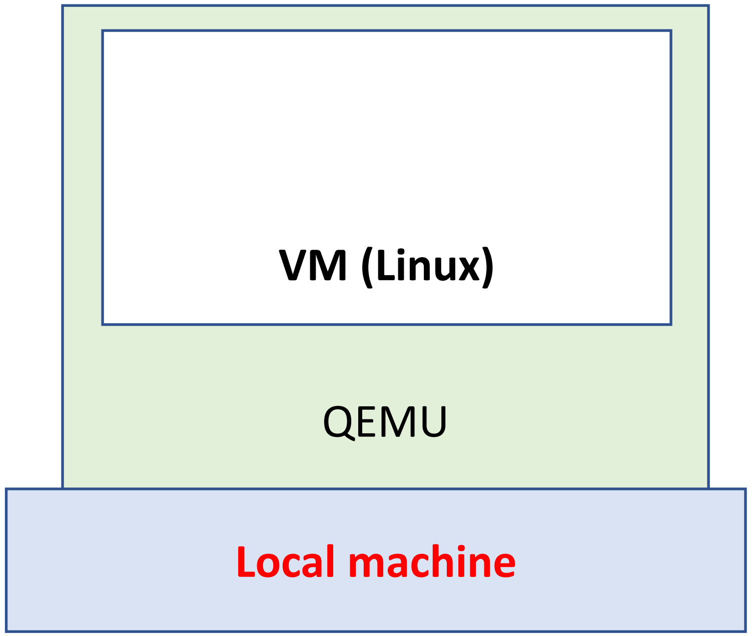


Setting up environment for Project 1

- We need to <u>run our build on **a QEMU Virtual Machine (VM),**</u> e.g., a VM on your local machine or on thoth
  - 1st, <u>**launch**</u> the default VM
  - 2nd, <u>**download**</u> our build (bzImage, System.map) to the VM
  - 3rd, <u>**configure**</u> the VM such that we can <u>**reboot**</u> it to our own kernel

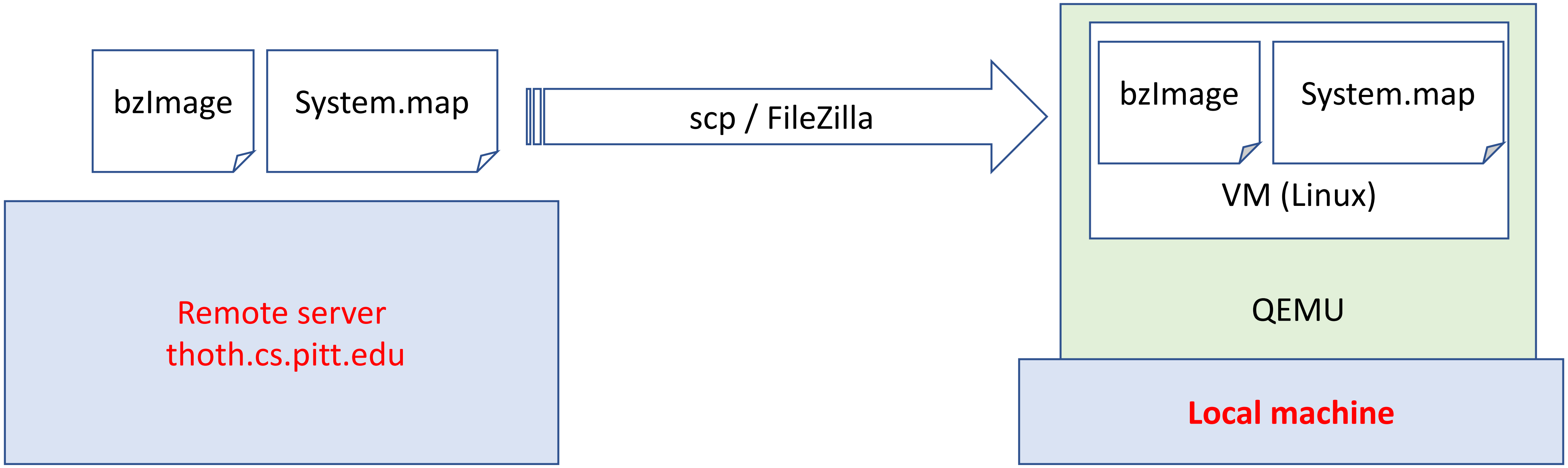
Setting up environment for Project 1

- 1st, **launch** the default VM: install qemu, then launch the VM image.



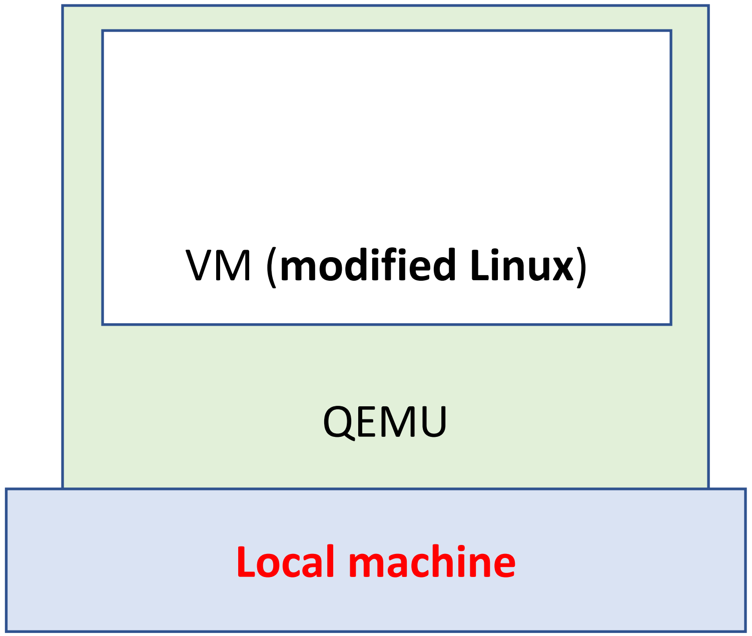
Setting up environment for Project 1

- 2nd, **download** our build (bzImage, System.map) to the VM



Setting up environment for Project 1

- 3rd, **configure** the VM such that we can **reboot** it with our build (a modified linux)

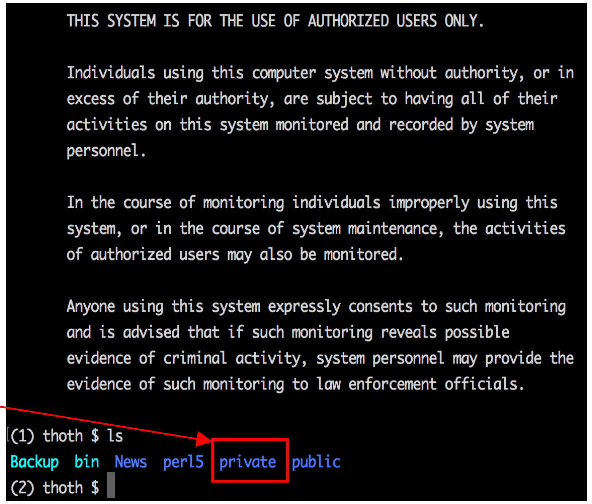


Setting up environment for Project 1

1. First log to your **thoth.cs.pitt.edu** account
    - SSH

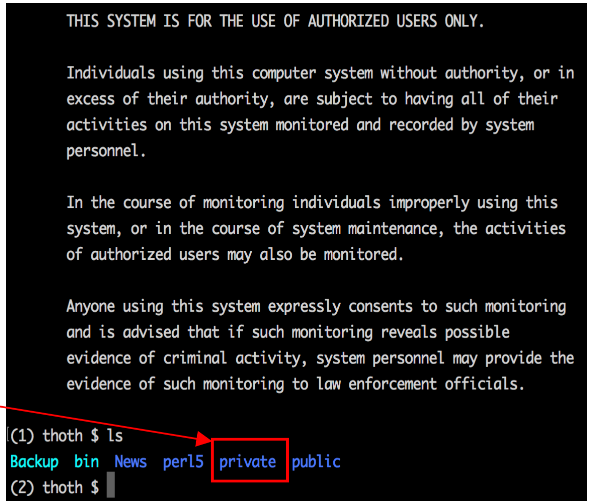
Setting up environment for Project 1

1. First log to your **thoth.cs.pitt.edu** account
   - SSH
2. Navigate to **/u/OSLab/username**
   - "/u/OSLab/username" will be your working directory. You can also use "~/private" as your working directory.
     - "cd /u/OSLab/ABC123" or "cd ~/private"



Setting up environment for Project 1

1. First log to your **thoth.cs.pitt.edu** account
   - SSH
2. Navigate to **/u/OSLab/username**
   - "/u/OSLab/username" will be your working directory. You can also use "~/private" as your working directory.
     - "cd /u/OSLab/ABC123" or "cd ~/private"
   - Copy linux source from /u/OSLab/original/linux-2.6.23.1.tar.bz2
     - cp /u/OSLab/original/linux-2.6.23.1.tar.bz2 .



This dot means current position. Since we just run "cd", it will be inside the working directory

Setting up environment for Project 1

3. Extract files locally
   - Run tar xfj linux-2.6.23.1.tar.bz2

Setting up environment for Project 1

3. Extract files locally
   - Run tar xfj linux-2.6.23.1.tar.bz2
4. Enter into **linux-2.6.23.1/**
   - Run cd linux-2.6.23.1

Setting up environment for Project 1

3. Extract files locally
   - Run tar xfj linux-2.6.23.1.tar.bz2
4. Enter into **linux-2.6.23.1/**
   - Run cd linux-2.6.23.1
5. Copy the .config file
   - Run cp /u/OSLab/original/.config .

Current position: WORKING_DIR/linux-2.6.23.1

Setting up environment for Project 1

3. Extract files locally
   - Run tar xfj linux-2.6.23.1.tar.bz2
4. Enter into **linux-2.6.23.1/**
   - Run cd linux-2.6.23.1
5. Copy the .config file
   - Run cp /u/OSLab/original/.config .
6. Build linux source code
   - Run make ARCH=i386 bzImage
7. Compile test programs
   - Run gcc -m32 -o trafficsim -I /u/OSLab/USERNAME/linux-2.6.23.1/include/ trafficsim.c

Setting up environment for Project 1

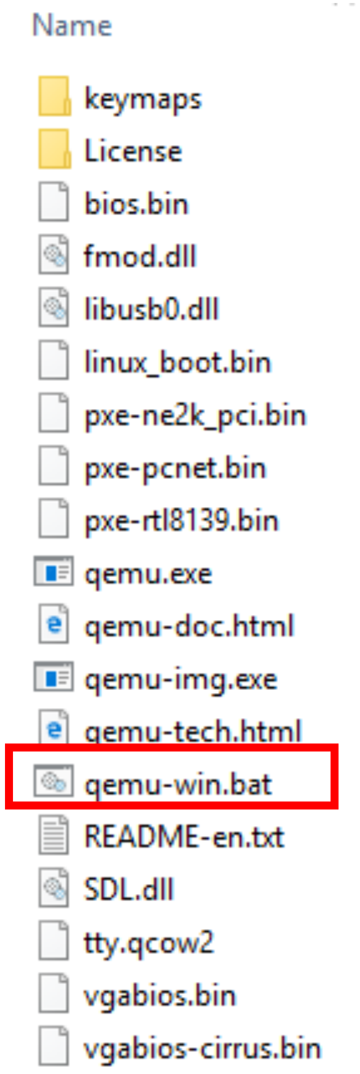
- After change the kernel code, we need to rebuild the Kernel
  - Run again make ARCH=i386 bzImage

Configuring QEMU

- We need a x86 version of QEMU (username and pass is **root**). It will be available in the "Assignment" on canvas.

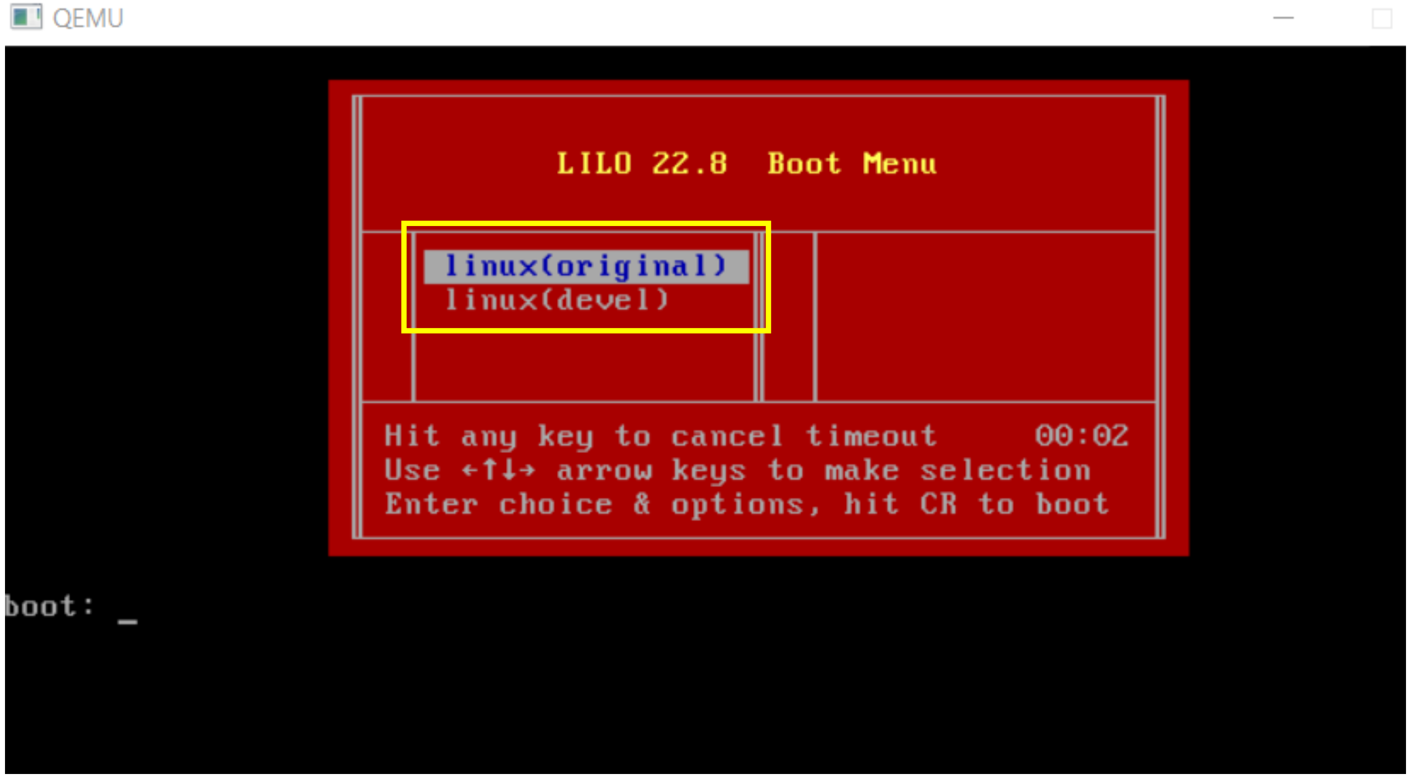
- Windows:
  - double-click/execute qemu-win.bat
- Mac/Ubuntu
  - Run the script:

    ./start.sh



Configuring QEMU

- Choose Linux(original): we will boot into “original”, download our build, connect it to the “devel”, then reboot to “devel”
  - User ‘**root**’ as user and password



Copying files from Linux to QEMU

- Now we need two files from the Linux we just built
  - Kernel File **bzImage** from:
    - linux- 2.6.23.1/arch/i386/boot/
  - System call map **System.map** from:
    - linux-2.6.23.1/
- Please be sure about **the path** where **you copied** the **linux distro!**
  - If you follow the steps here the linux files should be in **/u/OSLab/username**

Copying files from Linux to QEMU

- **FROM WITHIN THE NEW QEMU**
  - Now, we are in the default directory (in vm) after we type in "root" as user&pwd
- Download the files from your compiled Linux:
  - **scp** USERNAME@thoth.cs.pitt.edu:**/u/OSLab/**username/linux-2.6.23.1/arch/i386/boot/**bzImage** .
  - **scp** USERNAME@thoth.cs.pitt.edu:**/u/OSLab/**username/linux-2.6.23.1/**System.map .**

Dot (current position):
The default directory

- Install the rebuilt kernel in QEMU (to connect our build to "devel"):
  - cp bzImage /boot/bzImage-devel
  - cp System.map /boot/System.map-devel

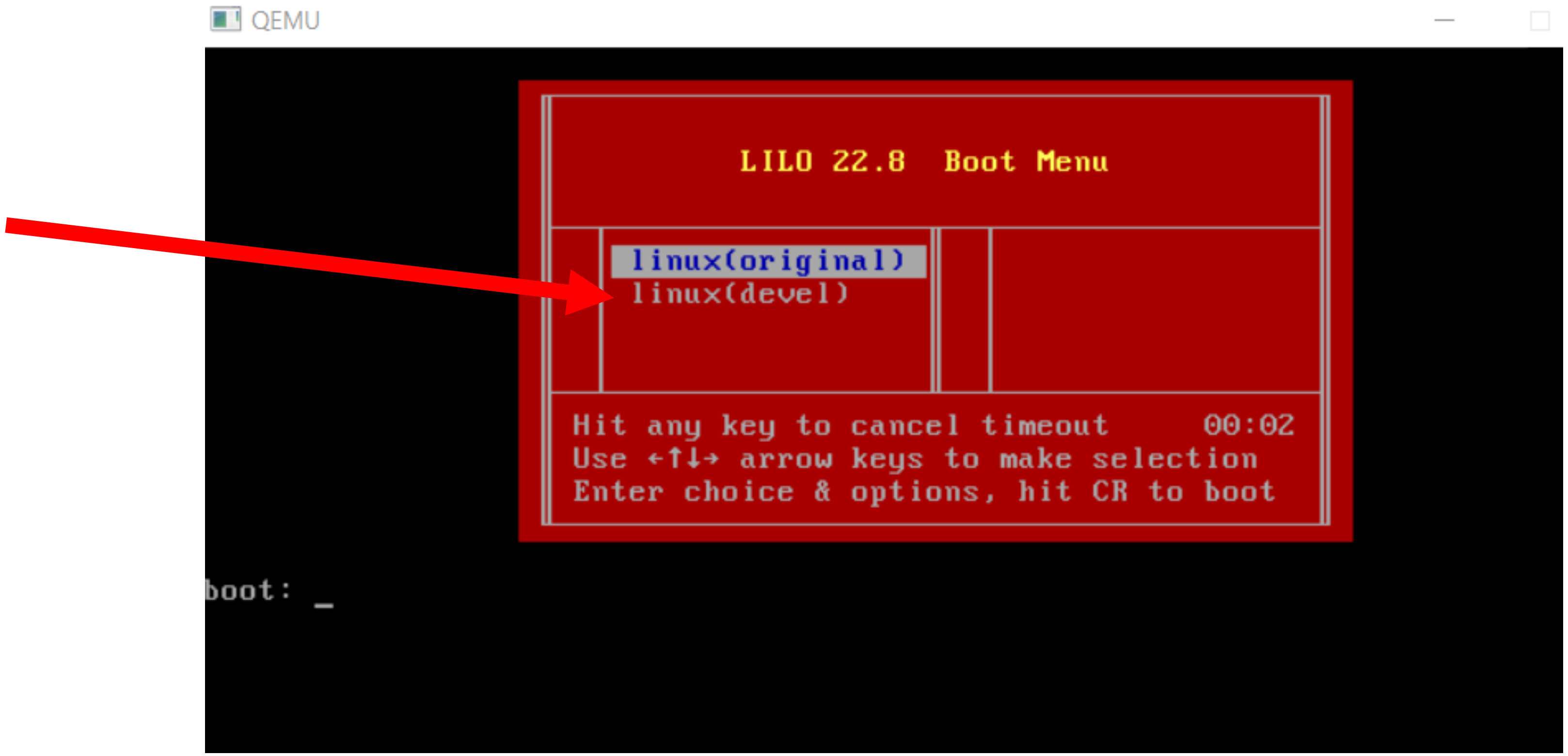
Copying files from Linux to QEMU

- After this run **li**nux **lo**ader command:
  - Run **lilo**
  - This will relink the new modified kernel you just copied

- Then reboot the system with the command:
  - Run **reboot**

Copying files from Linux to QEMU

- You will change to **linux(devel)** kernel
  - So to see changes always remind to choose it when opening Qemu



Run test program

Follow the same process:

- 1st , build the test program trafficsim **on thoth server**
  - **gcc -g -m32 -o** trafficsim **-I**/u/OSLab/USERNAME/linux-2.6.23.1/include/ trafficsim.c
    Exec name Header-file path Src file
  - For different tests, you only need to change src file and target executble name.
- 2nd, download the test program **to VM ("devel")**
  - After boot to the modified linux ("devel"), run scp to download tests
- 3rd, run it **inside VM("devel")**
  - ./trafficsim

Use GDB for project

You can run qemu on thoth server, then debug your kernel on thoth.

- Set up qemu on thoth
    1. cd /u/OSLab/USERNAME
    2. cp /u/OSLab/original/qemu.tar.gz .
    3. tar xzvf qemu.tar.gz
    4. cd qemu
    5. make qemu

Use GDB for project

You can run qemu on thoth server, then debug your kernel on thoth.

- We’ll need two connections (two terminals)
  - One is to launch the VM
  - The other is to launch gdb to debug the running VM

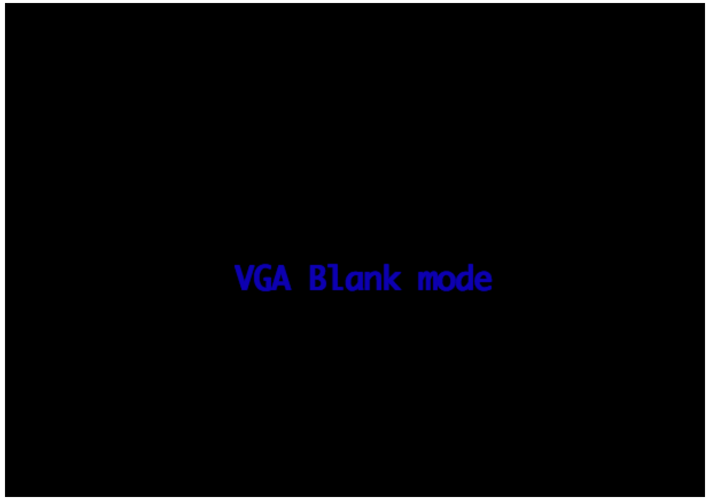
Use GDB for project

You can run qemu on thoth server, then debug your kernel on thoth.

- Launch the VM in first terminal
  - 1st , copy the modified kernel from your working directory to qemu folder
    - cd /u/OSLab/USERNAME/qemu
    - cp /u/OSLab/USERNAME/linux-2.6.23.1/vmlinux .

Current position:
/u/OSLab/USERNAME/qemu

  - 2nd, lauch vm with gdb enabled
    - make qemu-gdb
    - You'll see the vm is halted by gdb upon boot:



Use GDB for project

You can run qemu on thoth server, then debug your kernel on thoth.

- Launch the gdb in second terminal
  - 1st, open a new SSH connection to thoth.
  - 2nd, launch gdb
    - cd /u/OSLab/USERNAME/qemu
    - gdb -iex "set auto-load safe-path ."

```
(5) thoth $ gdb -iex "set auto-load safe-path ."
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-64.el6_5.2)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gp
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copy
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
The target architecture is assumed to be i386
+ target remote localhost:29209
0x0000fff0 in ?? ()
+ symbol-file vmlinux
```

Use GDB for project

You can run qemu on thoth server, then debug your kernel on thoth.

- Launch the gdb in second terminal
  - 1st, open a new SSH connection to thoth.
  - 2nd, launch gdb
    - cd /u/OSLab/USERNAME/qemu
    - gdb -iex "set auto-load safe-path ."
  - You'll see gdb has attached to the vm
    - Now, you can use gdb operations, e.g.:
      - Set a breakpoint by "b sys_cs1550_down"
      - Continue by "c"
        - The halted vm (in first terminal) will continue and finish booting
        - If you run the traffisim inside the VM. It will trigger the invocation of sys_cs1550_down, then the kernel will be halted upon sys_cs1550_down

```
(5) thoth $ gdb -iex "set auto-load safe-path ."
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-64.el6_5.2)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gp
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copy
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
The target architecture is assumed to be i386
+ target remote localhost:29209
0x0000fff0 in ?? ()
+ symbol-file vmlinux
(gdb) b sys_cs1550_down
Breakpoint 1 at 0xc011dd92
(gdb) c
Continuing.
```

Use GDB for project

Debugging the user program trafficsim.

- 1st, we need to load the symbol table of trafficsim to gdb
  - Get the text section address of trafficsim (by running readelf in thoth terminal)
    - readelf -WS PATH_TO_TEST_EXEC/trafficsim | grep text
    - You'll see an output like this:

This is the address we need

```
[13] .text             PROGBITS        08048610 000610 000e1c 00  AX  0   0 16
```

Use GDB for project

Debugging the user program trafficsim.

- 1st, we need to load the symbol table of trafficsim to gdb
  - Get the text section address of trafficsim (by running readelf in thoth terminal)
    - readelf -WS PATH_TO_TEST_EXEC/trafficsim | grep text
    - You'll see an output like this:

This is the address we need

```
[13] .text             PROGBITS        08048610 000610 000e1c 00  AX  0   0 16
```

  - Load the symbol table to gdb (by running add-symbol-table inside gdb)
    - (gdb) add-symbol-file PATH_TO_TEST_EXEC/trafficsim 0x08048610

  - Now, you'll be able to add breakpoint to traffisim.c, e.g.:
    - (gdb) b trafficsim.c:main
    - After that, if you run the trafficsim inside the vm, it will be halted by gdb at its main()

Lab 1 cont.
Add a new syscall to XV6

CS 1550 – xv6 – Adding a custom Syscall

Implement syscall inside kernel space

New syscall number (syscall.h): **#define SYS_getcount 22**

Extend syscall table (syscall.c): **[SYS_getday] sys_getcount,**

extern int sys_getcount(void);

Implement syscall routine (sysproc.c):

```
int sys_getcount(void) {
    // return how many times
    // this syscall has been called
    // by the calling process
}
```

Provide user interface for user-space

User interface definition (usys.S): **SYSCALL(getcount)**

User interface declaration (user.h): **int getcount(int);**

CS 1550 – xv6 – Adding a custom Syscall

Implement syscall inside kernel space

New syscall number (syscall.h): **#define SYS_getcount 22**

Extend syscall table (syscall.c):

[SYS_getday] sys_getcount,

extern int sys_getcount(void);

Implement syscall routine (sysproc.c):

```
int sys_getcount(void) {
    // return how many times
    // this syscall has been called
    // by the calling process

    /* which syscall? which process? */
}
```

CS 1550 – xv6 – Adding a custom Syscall

Implement syscall inside kernel space

- New syscall number (syscall.h): **#define SYS_getcount 22**
- Extend syscall table (syscall.c):
  - **[SYS_getday] sys_getcount,**
  - **extern int sys_getcount(void);**
- Implement syscall routine (sysproc.c):

```
int sys_getcount(void) {
    // return how many times
    // this syscall has been called
    // by the calling process
    /* 1st, identify the calling process */
    struct proc * calling_proc = myproc();
    /* 2nd, retrieve the syscall num */
    int num;
    argint(0, &num);
    /* 3rd , return the corresponding counter */
    ......
}
```

User process will call getcount with one parameter: the syscall number N, to get how many times this syscall-N has been **called by this user process**.

But, inside kernel, all syscall routines takes void as parameter, so we need to **retrieve** the actual parameter (syscall number), by using argint()

CS 1550 – xv6 – Adding a custom Syscall

Implement syscall inside kernel space

New syscall number (syscall.h): **#define SYS_getcount 22**

Extend syscall table (syscall.c): **[SYS_getday] sys_getcount,**

extern int sys_getcount(void);

Implement syscall routine (sysproc.c): **int sys_getcount(void) {**

```
    // return how many times
    // this syscall has been called
    // by the calling process
    /* 1st, identify the calling process */
    struct proc * calling_proc = myproc();
    /* 2nd, retrieve the syscall num */
    int num;
    argint(0, &num);
    /* 3rd , return the corresponding counter */
    ……
}
```

Each process will have its own counters, recording the calling times for each syscall. E.g., xv6 by default has 21 syscalls, so each process needs 21 counters.

These counters should be per-process. So we define it as part of per-process metadata: **"struct proc" inside "proc.h".** After adding this new metadata, we also need to initialize it. We can initialize it upon the process creation: **check the "allocproc()" function inside "proc.c", and find the position to initialize it**

After that, we are able to **update** and **return** these counters during runtime

CS 1550 – xv6 – Adding a custom Syscall

Implement syscall inside kernel space

- New syscall number (syscall.h): **#define SYS_getcount 22**
- Extend syscall table (syscall.c):
  - **[SYS_getday] sys_getcount,**
  - **extern int sys_getcount(void);**
- Implement syscall routine (sysproc.c): **int sys_getcount(void) { }**

After that, we are able to **update** and return these counters during runtime

Where can we update the counters??
--Check the logic of the global syscall entrance function:
"syscall()" inside "syscall.c"

This is the position we actually call a syscall routine

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

CS 1550 – xv6 – Adding a custom Syscall

Test program (getcount.c) is provided in lab1 pdf:

```
main() {
    …
    getcount(N)
    …
}
```

1. Integrate it into XV6 by modifying makefile: UPROGS, EXTRA

2. Rebuild and launch xv6, run “ls”, now you can see the test program listed as a command. Run it and check the results. Expected results are given in the lab1 pdf.

: If you saw correct numbers from your own test but wrong numbers from autograder’s test. Most likely, the initialization of per-process counters is wrong. **Such inconsistency typically results from <u>missing</u> or <u>improper</u> initializations. If you met this problem in future labs/projects, check initializations first.

**: You can also use gdb to debug your labs. Similar as debugging projects, you need two connections. Please refer to gdb instructions in lab1 pdf.