ReadMe.txt

To run a program in the background, prepend the shell command with &. Ex: &ping 192.168.7.4

Programs:

- ping <ip>
- TestProgram
    - This the window manager. It is run automatically in the background.
- np <dest ip> <dest port> <listen port> (must be single digit ports)
    - must be run on two qemus; controls – WASD; moves a line in the window on the other instance of 'np' in the other qemu – W – up; A – left; S – down; D – right;
- TestGame <server ip>
- TestServer

For instructions on how to test anything (it's not simple), read to the end of this readme.

TestProgram (window manager)

The window manager starts automatically when qemu is run (the shell is forced to exec it) and runs in the background.

Controls:

- **p:** cycle foreground window.
- **i, j, k, l:** move foreground window left, right, up, down

A new resource called **ScreenBuffer** was created for this. The screen buffer allows a process to write to VGA memory. Any process may request a screen buffer – the first process to request a screen buffer resource is granted a lock on writing to vga memory. A process only (and automatically) creates a window if it requests a screen buffer resource. This instance of screen buffer is also tracked globally. Subsequent processes requesting a screen buffer still receive one. But their buffer writes (via the WriteScreenBuffer() syscall) do not go directly to vga memory, but instead to the screen buffer's personal memory. The screen buffers are stored as children of the main buffer. The process owning the main buffer can request the buffer memory of its screen buffer resource's child screen buffers, and then compose the buffers as it wishes. This allows the window manager process to correctly combine the displays of overlapping windows.

When a new process requests a screen buffer resource, the request action is added to a queue in the main screen buffer. The owner of the main screen buffer can use the syscall GetNewWindowRequests() and GetBufferRequestCount() (these functions have the side effect of removing the item as well from the queue) to query for new request actions in this queue. The window manager uses these functions every update frame so that it can create new window objects with the process id of the child process and manage them.

The window manager keeps a list of these window objects, and each frame, requests the video buffer memory from each child (with the syscall GetChildBuffer(char* buffer, isn't processId)). It then writes the child buffer into its own buffer. Finally, after copying the data from every child buffer, it uses the WriteScreenBuffer() syscall to write to VGA memory.

The window manager maintains an ordered list of process windows. The foreground window is always rendered on top.

The syscalls LockScreenBuffer(int id) and UnlockScreenBuffer(int Id) allow the buffer to be locked so that it is not read and written at the same time. A process writing to a buffer should lock it first.

Keyboard Input:

The window manager also manages routing key input to the foreground process window (the qemu program window must be selected – input from the console is not routed). The controls for managing windows (p, I, j, k, l) are intercepted by the window manager and not routed).

The syscall GetKeyPresses(char* buf, int bufferLength) has the keyboard interrupt bounded buffer be copied into a buffer to be read by the requesting process and clears the buffer. In this case, only the window manager calls this function.

The syscall QueueChildKeyInput(int childProcessId, char key) allows the window manager to pass the key input it reads from GetKeyPresses() to the current foreground window processes's key input buffer. Each process can use the syscalls GetQueuedKeyPressCount() and GetQueuedKeyPresses(char* buf, int bufferLength) to retrieve key input from their own process's local key input queue).

Files:

Kernel:

- ScreenBuffer.h
- ScreenBuffer.cc

User:

- TestProgram.cc
- DesktopWindowManager.h
- DesktopWindowManager.cc

Syscalls:

- GetScreenBuffer(int width, int height) //create screen buffer resource
- WriteScreenBuffer(int resourceId, unsigned char* buf) //copy user video buffer to system
- GetNewWindowRequests(int* buf) //get process ids and dimensions for process who have   requested a window to be created and clears the window request queue

- GetBufferRequestCount() // get the number of processes who have requested a window since the request queue was last cleared
- GetChildBuffer(unsigned char* buf, int processId) //get the last written graphics data written by a particular process
- LockScreenBuffer(int resourceId) //lock read / write access to buffer
- UnlockScreenBuffer(int resourcdId) //unlock read / write access to buffer

Networking:

Files:

- Network.h
- Network.cc
- NetworkProcess.h
- NetworkProcess.cc
- Socket.h
- Socket.cc

Syscalls:

- TestDraw() // initializes the network
- Ping(unsigned char destIP[4]) //ping
- OpenSocket(int protocol, int port) //open socket resource for process, using specified protocol to listen on specified port
- ReadSocket(int socketDescriptor, unsigned char srcIP[4], unsigned char* buffer, unsigned int bufferSize) //read up to bufferSize bytes from the the next queued packet on the calling process's received packet queue.
- WriteSocket(int socketDescriptor, const unsigned char destinationIP[4], const unsigned char* const buffer, int bufferSize, int port) //send bufferSize bytes to dest IP:port using protocol of specified socket

Driver:

We implemented a driver for the RTL8139 network card. We use DMA and IRQ interrupts to handle sending and receiving data.

P439 protocol:

The P439 protocol is similar to UDP, but the only data in its header field is a port number.

Protocols:

We supported ARP, ICMP via IPv4 and P439 via IPv4.

The class **Network** contains the low level code for interacting directly with the network card and for handling the details of each particular protocol, and our ARP cache. The class **NetworkProcess** handles the high-level communication between the network and other processes.

NetworkProcess:

Network process is a new process that runs forever in the background to manage the destinations of network packets. Packets are never sent or received directly by processes. They are sent to the NetworkProcess process, which queues them, and during NetworkProcess's run(), it handles sending and receiving the queued packets.

When a new packet is received, it is dissected by the Network class. If it is found to be a packet whose protocol uses the socket table, it is passed to *Process::networkProcess->QueueNetworkReceive(packet);* and queued to be later sent to the destination process. At this point, the packet data itself has been stripped of its header data.

Process::networkProcess is a static NetworkProcess variable.

Each time the NetworkProcess Run()'s, it directs the packets in its receive queue to the correct receive queues of the destination process the packet is intended for. The correct destination process of a packet is determined by looking at the port number in the header of the packet. The port number is an index into the static socket table. The user program can use syscalls to read its own receive queue.

When a new packet is sent using a syscall, it is sent into NetworkProcess's send queue. When NetworkProcess Run()'s, it empties its send queue into Network::SendPacket().

Packets sent from syscalls contain no headers. It is the job of Network::SendPacket() to finally build the header immediately before the packet is sent.

If, when attempting to build the header, the Network::SendPacket() finds a miss in the ARP cache, failure is returned with the side effect that SendPacket() queues its own ARP request packet, and the NetworkClass requeues the failed packet for a send attempt 100 milliseconds later.

For sending packets to 127.0.0.1, packets are routed directly to the receive queue of NetworkProcess and skip the network card.

How to get anything to run at all.

To run anything at all in our code requires some configuration.

Create file with any name and paste these contents:

http://git.yoctoproject.org/cgit.cgi/poky/tree/scripts/runqemu-gen-tapdevs

Create file called runqemu-ifup in same directory as script and paste these contents: http://git.yoctoproject.org/cgit.cgi/poky/tree/scripts/runqemu-ifup

example on how to run script: `sudo 1000 1000 3 /`

//might be necessary for the script

install tuctl (uml-utilities)

change to sbin (if it complains about something not found in bin)

//To see data we are sending to QEMU

```
tcpdump -xx -e -i tap0
```

//Current best run command for our kernel

```
qemu-system-x86_64 -net nic,model=rtl8139 -net tap,ifname=tap0 --serial mon:stdio -hdc kernel/kernel.img -hdd fat439/user.img
```

//To run the game

1. create taps

2. run the first qemu: qemu-system-x86_64 -net nic,model=rtl8139 -net tap,ifname=tap0 --serial mon:stdio -hdc kernel/kernel.img -hdd fat439/user.img

3. run the second qemu: qemu-system-x86_64 -net nic,model=rtl8139,macaddr=52:54:00:12:34:65 -net tap,ifname=tap1 --serial mon:stdio -hdc kernel/kernel.img -hdd fat439/user.img

4. in the second qemu: &TestServer

5. in the second qemu: &TestGame 127.0.0.1

6. in the first qemu: &TestGame 192.168.7.4

7. type "t" in both windows and the game will start

If you are running this on two emu on the same computer, you can only control one game at a time. You have to switch between each to move the paddle.