Adding a Window Manager to xv6

Final Project Write-up

CS3210

Hunter Morrell

Andrew Ray

Maxim Mints

Intro

The XV6 interface is a relatively simplistic one, consisting solely of a terminal interface. XV6 is not user-friendly, especially when compared to most modern operating systems. It even lags behind the windowing systems of over thirty years ago! It lacks any sort of graphical interface that many are used to working with. Though the terminal interface is powerful and robust, it is not well suited to multi-tasking. This is because the current interface limits the user to run at most one command at a time. This is not acceptable, as it wastes the potential of XV6 and can sometimes make it difficult to wrangle it into doing exactly what you want.

We decided to fix this issue by implementing a windowing system inside of XV6, specifically a tiling window manager. This implies interfacing with the VGA hardware, gathering input from the mouse, and creating the userland support for drawing windows on the screen.

Background

Historically, window managers were created in order to resolve this exact problem. WMs allow for a level of abstraction that makes things easier and more convenient for users who may or may not be versed in all of the many intricacies of the terminal and the commands found there. Rather than having to spend time and effort looking up and implementing (now arcane) commands, one had the ability to simply operate within a familiar graphical interface and accomplish the same thing with much less time.

There are multiple types of window managers that were available to users over the years. Those window managers could generally be categorized into two categories: tiling window managers, and stacking window managers. Examples of tiling window managers include i3, and the “awesome” project. These window managers have fallen out of favor, but are still used by a niche community today. Most users are now more familiar with the stacking window managers, such as those that come with Windows, OSX, and many Linux/BSD distributions.

Tiling window managers, which is what we chose, are basic, yet still practical. Most importantly, they are functional. Instead of stacking windows that can overlap and overlay, like in Windows or OSX, tiling window managers instead have a grid that new windows fit into. Each window that gets added is slotted into this grid and everything is resized automatically to fit this new addition. This removes a lot of the complexity that comes with having to worry about floating windows that can potentially overlap and draw over each other.

In terms of how they are typically implemented, Stacking window managers use linked lists to store the position of the windows on the screen. This allows them to pop from anywhere in the list, and append to the tail. This makes the job of rendering the screen very easy for the rendering logic. All it has to do is to iterate through the list, rendering the windows from the back of the stack, to the front. However, this task is not so easy for tiling managers. Tiling managers have little choice but to use a binary tree to store the hierarchy of windows, since windows can be split beyond any practical use case. Using a binary tree makes this task relatively simple. Every window is found in a leaf node. Every time a window is split, a parent node is created, with one child containing the old contents of the window, and the new child being the newly created window. The parent node occupies the position of the window that was split.

The coincidental almost-namesake of our project, the X.org project that backs almost every window manager that Linux has ever had, is also a user-space program that accepts commands from other user-space programs through a socket, and updates the screen accordingly through a number of system calls. Not all graphics servers and window managers are implemented in user space. Windows uses a more kernel-heavy approach when implementing their graphics system; it uses an interface that Microsoft calls the “Windows Display Driver Model.” Part of the WDDM is implemented in the Windows kernel, and the rest of it is implemented in user-space.

Approach

We originally had four main goals that we wanted to achieve:

- Draw static images and shapes with VGA
- Move a cursor across the screen
- Display windows as graphical tiles
- Add ASCII text display capabilities to the server

Before discussing the approach for each goal, it’s necessary to describe some of the self-assigned constraints that make the implementation of the windowing system more interesting. Our group chose to implement the windowing system in user-space, which adds a considerable amount of complexity, as our access to hardware events becomes far more constrained. However, other user-space programs should still have access to the windowing system, and create arbitrary graphical elements on the screen. This creates two distinct domains of development: Our group had to implement both user-space features, and kernel-space extensions to make our end result.

In short, the kernel had to be modified to display VGA, create system calls to access the memory buffer that VGA reads from, implement the mouse drivers, and finally, create an event queue that could be readable from user-space through a system call. This event queue would funnel keyboard and mouse events to the caller.

For the display, the VGA display mode 13h was used. This is the linear 256 color mode, which was chosen for its simplicity so that there is no need to handle VGA planes explicitly. The screen is encoded in the VGA video memory (used in memory-mapped I/O) as a sequence of bytes, where each byte corresponds to one pixel of a specific color, where the color can be determined by indexing the color palette with the byte value. The color palette used was the web-safe 216-color palette, which defines 6 possible values for r, g and b, scaled between 0 and 255, which makes it easy to automatically generate an indexed pallette with. It is also very easy to find convert from an RGB value to the index reprsenting the closest color, which is employed by the server to draw BMP images.

As previously stated, the video memory buffer was made writable by new system calls. A “backup” buffer was also added, which can back up the contents of the main buffer. The window server always draws to both buffers, however, the cursor, which is handled entirely in the kernel, is only drawn to the main buffer. This was done to make the cursor independent of anything drawn by the window server: whenever the cursor is moved, the area where it was previously located is copied over from the backup buffer to the main memory buffer. The cursor is drawn pixel-by-pixel with an image mask, which means it does not have to have a rectangular shape.

The PS/2 mouse driver was added separately from the keyboard driver. Both the Keyboard and the Mouse send input to the processor through the same port. However, both signals are accompanied by control signals that make it easy to sort a keypress from a mouse movement or a mouse event, which was used to synchronize the two drivers.

Once the mouse driver was implemented and tested, the events from the mouse and the keyboard were funneled into a newly created event queue that the Y window server would later access. An extra system call was necessary to pop from this queue. Mouse events were also sent directly to the cursor, which was moved as many pixels as the number of “mickeys” the mouse moved (this seems to make the cursor overly sensitive to some touchpads, however, we haven’t seen problems with normal mouse input).

Once all of the necessary kernel features were implemented, it became necessary to develop the user space.

The Y Window server is forked by the init process. Y then forks `sh` after adding two new file descriptors, corresponding to two pipes which any user program spawned through the shell can use to communicate with the server. User programs would communicate with Y by writing command messages into the write end of the pipe funneling commands to the serer (file descriptor 4), and read communication data from the server from the read end of the other pipe (file descriptor 3). Inside of Y, there is an event loop which processes any new mouse and keyboard events, and then any new user commands.

As previously described, the windows were implemented using a tree structure, and certain key combinations were listen for to create and remove windows (tiles). A tile can be created by using the key combination Ctrl ‘ or Ctrl , to split, correspondingly, horizontally or vertically. Clicking a tile with the mouse selects it, highlighting its borders in green. The selected window is referred to as the active window. Ctrl - closes the active window. The borders can be clicked with the cursor and dragged, which resizes them.

If any process spawned in a shell with the previously described pipes sends a message to the server via pipe 4, it is immediately assigned to the active window and the message is processed. For example, the imgviewer program included for demonstration purposes will use the drawbmp message to pass in a file name of a .bmp file to be drawn on the server (the example bmps which we fit into xv6 memory are diddy.bmp and y_u_no.bmp). This program, imgviewer, then blocks. If needed, another shell can be spawned at any time to launch a program concurrently in another window, using the key combination Ctrl =.

Each window is assigned its own image buffer, which is made the same size as the entire screen for consistency, where images can be drawn. Its contents are drawn to the main VGA display memory buffers and redrawn depending on window resizing events. The contents of a window’s display buffer are drawn/copied onto different areas of the main VGA buffers depending on the window’s size and offset in both dimensions.

Results

Only three of the four goals listed above were implemented; ASCII support was never added to the window manager, as it would ultimately not add anything substantial to the project. Since our graphics server was command-based, adding text support would have entailed finding a font online, and adding a command to display the image of the character at a specific location. Since this operation would have been trivial, no attention was paid to it. The goal was originally chosen because our group assumed that it would have been more difficult to implement. As such, we only defined one command message, drawbmp, to show that our proof-of-concept works as expected. This command would draw a bitmap image to the top-left corner of the window.

There was a minor oversight in our project which should be noted: we neglected to add support for using the window system to initiate and respond to program exits. To implement this feature, we would have had to make the program aware of its own PID (perhaps through a system call), and define messages that could be used by a process to: set its own pid (in case it should be killed by closing the window or opening another program in the same window); and to signal its closing (in case it terminates normally). The above would have been trivial to implement. Additionally, we would have wanted to add process exits to the event queue. It would not have been difficult to implement either.

Conclusions

We had four main lessons learned from this project.

First, making the window system a userspace program in lieu of a set of system calls may have been a bad design choice for XV6. Despite choosing a tiling window manager over the other options due to its simplicity, this approach actually increased the complexity of this. Even though the user-space approach gives us the benefit of using pipes as individual event queues for communication with the processes, we did not even end up using this extensively.

Second, the VGA 256-color palette is not large enough to give a good color rendition for images. Images draw successfully, but for ones with a large amount of colors, they often look faded due to the reduced color set.

Third, we encountered an odd issue with QEMU where it apparently lags at redrawing roughly ten of the pixel rows at the bottom of a 320x200x256 screen, causing screen tearing. To counter this, we simply stopped drawing somewhere above those rows, essentially changing our entire implementation as if the QEMU VGA screen was slightly smaller than it actually is.

Fourth and last, QEMU had another odd issue where it fails to capture the mouse correctly when running inside of a VM. When this occurs, it has the effect of locking the cursor to where it is initially drawn at the top left at {0,0} and preventing it from actually moving.

Notes

We decided to use an alternative approach to structuring this document, which was to combine the problem and intro sections into a single intro section. Our base problem was simplistic enough to introduce and explain with the rest of the introduction.