Homework 2

Hongxi Chen(hc2325)

Sep. 24, 2025

1. What is the difference between a Linux pipe and a FIFO (or a named pipe)? What can you do with a FIFO (named pipe) that you cannot do with a pipe?

A **pipe**, also known as an anonymous pipe, is a unidirectional communication channel that can only exist between related processes, such as a parent and its child process. The pipe is created in memory and does not have a corresponding file on the filesystem. It is temporary and is destroyed once the processes communicating through it terminate.

A **FIFO** (First-In, First-Out), or named pipe, on the other hand, has a presence in the filesystem. It is a special file that can be used by any number of unrelated processes for communication, as long as they have the necessary permissions to access the file. Because it has a name and exists on the filesystem, a FIFO is persistent and remains until it is explicitly deleted.

The primary capability of a FIFO that a regular pipe lacks is the ability to facilitate communication between **unrelated processes**. Since a pipe has no name, it can only be used by processes that inherit the file descriptor from a common ancestor. A FIFO's existence as a file allows any process that knows its path to open it for reading or writing.

2. List the key differences between using 'bash', 'source' and executing directly using './' when running a shell script.

- **bash script.sh**:
  - **Process**: This command explicitly starts a new instance of the `bash` interpreter, which then executes the script's commands. A new child process (a subshell) is created.
  - **Environment**: Since the script runs in a subshell, any changes to environment variables, functions, or the current directory are confined to that subshell and are lost when the script finishes. The parent shell's environment remains unaffected.
  - **Permission**: The script does not need to have execute permissions. It only needs read permission for the `bash` interpreter to read its contents.
- **source script.sh**:
  - **Process**: The script is executed within the **current** shell context. No new process is created.
  - **Environment**: Any environment variables, functions, or changes to the current directory made by the script will persist in the current shell after the script has finished. This is because the commands are run as if they were typed directly into the terminal.
  - **Permission**: The script does not need execute permissions, only read permission.
- **./script.sh**:
  - **Process**: This command executes the script as a separate program, which creates a new child process (a subshell).
  - **Environment**: Similar to `bash script.sh`, changes to the environment within the script do not affect the parent shell.
  - **Permission**: The script **must** have execute permissions (`chmod +x script.sh`). The first line of the script (the "shebang," e.g., `#!/bin/bash`) determines which interpreter is used to run the script.

3. List all the pins on the R-PI GPIO connector used by the piTFT screen. Explain what each pin is used for by the piTFT.

Based on the official Adafruit documentation, the PiTFT screen utilizes the following pins on the Raspberry Pi's GPIO connector for its primary and optional functions.

Primary Function Pins:

These pins are essential for the basic operation of the screen and the resistive touchscreen.

Hardware SPI Pins (SCK, MOSI, MISO, CE0, CE1) These pins form the high-speed Serial Peripheral Interface (SPI) bus. This bus is responsible for the primary digital communication, sending image data to the display controller and reading touch coordinates from the touch overlay.

GPIO 24 and GPIO 25 These two general-purpose pins are used for additional control signals required by the display and touch controller, working alongside the SPI bus.

Optional Function Pins:

These pins are connected to optional features on the PiTFT board that a user can choose to implement.

GPIO 18 This pin is specifically designated for controlling the screen's backlight brightness using Pulse Width Modulation (PWM). This allows for dimmable backlight control. This feature is optional and can be disconnected by cutting a solder jumper on the board if the pin is needed for other purposes.

GPIO 17, 22, 23, and 27 These four pins are routed to pads on the PCB for adding optional tactile buttons. This allows for the creation of a simple, physical user interface directly on the device.

Reference: Adafruit PiTFT - 2.8" Touchscreen Display for Raspberry Pi..

4. For the R-Pi 4, Model B, list all possible GPIO pins that may be used for projects and labs. Identify the maximum set (when not using any special functions). Also, list the minimum set, when all special functions (and the Adafruit 2.8-inch piTFT) are used.

Answers are as follows:

4.1 Maximum Available GPIO Set

When no special hardware interfaces (like SPI, I2C, UART) are enabled, the Raspberry Pi 4, Model B, provides a maximum of **26 GPIO pins** that can be used for general-purpose projects and labs. These pins are:
GPIO 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.

4.2 Minimum Available GPIO Set

The minimum set of available GPIO pins is determined under a heavy load scenario, with the following conditions applied simultaneously:

- The Adafruit 2.8-inch PiTFT is active (display, touch, and backlight).
- The four optional tactile buttons on the PiTFT are in use.
- All possible I2C interfaces shown on the pinout diagram are enabled.
- All possible UART interfaces shown on the pinout diagram are enabled.

Under these specific constraints, a total of **4 GPIO pins** remain available for other projects.
Final Available GPIO Pins The remaining, unallocated GPIO pins are:

- **GPIO 19**

- **GPIO 20**
- **GPIO 21**
- **GPIO 26**

Detailed Pin Allocation Summary The following table details the allocation of the 22 general-purpose GPIO pins that are consumed by the specified hardware and interfaces, justifying the final result.

GPIO Pin	Assigned Function / Device	Source Document
GPIO 2	I2C1 (SDA1)	Pinout Diagram
GPIO 3	I2C1 (SCL1)	Pinout Diagram
GPIO 4	UART3 (TXD3)	Pinout Diagram
GPIO 5	I2C5 (SCL5)	Pinout Diagram
GPIO 6	I2C4 (SDA4) / UART5 (RXD5)	Pinout Diagram
GPIO 7	PiTFT (SPI0 CE1) / UART4 (RXD4)	PiTFT PDF & Pinout Diagram
GPIO 8	PiTFT (SPI0 CE0) / UART4 (TXD4)	PiTFT PDF & Pinout Diagram
GPIO 9	PiTFT (SPI0 MISO) / I2C4 (SCL4)	PiTFT PDF & Pinout Diagram
GPIO 10	PiTFT (SPI0 MOSI) / I2C5 (SDA5)	PiTFT PDF & Pinout Diagram
GPIO 11	PiTFT (SPI0 SCLK)	PiTFT PDF
GPIO 12	I2C5 (SDA5) / UART5 (TXD5)	Pinout Diagram
GPIO 13	I2C5 (SCL5) / UART5 (RXD5)	Pinout Diagram
GPIO 14	UART0 (TXD0)	Pinout Diagram
GPIO 15	UART0 (RXD0)	Pinout Diagram
GPIO 16	I2C6 (SDA6)	Pinout Diagram
GPIO 17	PiTFT Tactile Button	PiTFT PDF
GPIO 18	PiTFT Backlight Control	PiTFT PDF
GPIO 22	PiTFT Tactile Button	PiTFT PDF
GPIO 23	PiTFT Tactile Button	PiTFT PDF
GPIO 24	PiTFT Control Pin	PiTFT PDF
GPIO 25	PiTFT Control Pin / UART4 (RXD4)	PiTFT PDF & Pinout Diagram
GPIO 27	PiTFT Tactile Button	PiTFT PDF

5. In Lab1, a shell script named "start_video" was created to run mplayer and video_control.py by using a single command. What is the correct ordering of operations within this script for this shell script to terminate correctly?

The correct ordering and management of operations are:

1. **Create the FIFO**: The named pipe (e.g., `video_fifo`) must be created first. This is the communication channel between `video_control.py` and `mplayer`.
2. **Run mplayer in the background**: `mplayer` should be launched in slave mode, reading commands from the FIFO. It must be run as a **background** process (using `&`). This allows the script to continue to the next command without waiting for `mplayer` to finish. If `mplayer` were run in the foreground, the script would block until the video playback is manually quit.

   Example: `mplayer -slave -input file=./video_fifo my_video.mp4 &`
3. **Run video_control.py in the foreground**: The Python script that listens for button presses and writes commands to the FIFO should run in the **foreground**. This script typically contains a loop that runs until a 'quit' button is pressed. When the user signals the script to exit, it should perform any necessary cleanup (like writing a 'quit' command to the FIFO for `mplayer`) and then terminate.

   Example: `python3 video_control.py`

Because the Python script is the last command running in the foreground, when it terminates, the shell script also concludes, and the command prompt returns. The background `mplayer` process will be terminated when the shell exits.

6. If you run the date command on the ECE5725 server, the date and time are accurate. The RPi does not have a battery-backed real-time clock so how does the RPi maintain accurate time? When would it be appropriate to add an external, battery-backed real-time clock to the RPi?

The Raspberry Pi maintains accurate time by connecting to the internet and synchronizing its system clock with a time server using the **Network Time Protocol (NTP)**. When the Raspberry Pi boots up with a network connection, the operating system contacts an NTP server to get the current date and time and sets its internal software clock accordingly. This process is repeated periodically to correct any drift.

It would be appropriate to add an external, battery-backed Real-Time Clock (RTC) to a Raspberry Pi in situations where the device will not have a consistent or reliable internet connection. An RTC is a small, battery-powered chip that can keep track of time even when the Pi is powered off.

7. For the following RPi GPIO circuit, describe why R2 is necessary in the figure. Describe a possible 'software situation' that would damage the GPIO without R2. How does R2 prevent the problem? Why is the value of 1k ohm selected for R2?

- **Why R2 is necessary**: R2 is a **current-limiting resistor**. Its primary purpose is to protect the GPIO pin from excessive current in case of a software misconfiguration or an accidental short circuit.
- **Software situation that would damage the GPIO**: A common software error is to configure the GPIO pin as an **output** and set it to a HIGH state, while the circuit is designed for it to be an input. If the pin is set as a HIGH output (driving 3.3V), and then the user presses "Switch 1," a direct path is created from the 3.3V GPIO pin to Ground. Without R2, this would be a short circuit, causing a very large amount of current to flow from the pin to ground, which can permanently damage the GPIO pin and potentially the Raspberry Pi's processor.
- **How R2 prevents the problem**: R2 is placed in series between the GPIO pin and the rest of the circuit. In the fault scenario described above, R2 would be between the 3.3V output and ground. According to Ohm's Law ($I = V/R$), the resistor limits the current to a safe level. With a 1kΩ resistor, the current would be limited to $I = 3.3V/1000\Omega = 3.3mA$. This small amount of current is well within the safe operating limits of a GPIO pin and will not cause damage.
- **Why 1kΩ is selected**: The value of 1kΩ is a common choice because it's a good compromise. It's low enough that it doesn't significantly interfere with the logic levels of the input pin (which has a very high internal impedance), ensuring that the pin can still reliably read HIGH or LOW states. At the same time, it's high enough to provide excellent current limiting protection in the case of a short circuit.

8. In Lab1 and Lab2, you use a python code video_control.py, for example, to respond to buttons connected to the RPi GPIO pins. video_control.py passes these events to an instance of mplayer, controlling the video under playback. The two processes communicate using video_fifo. Describe what would happen if start_video is run WITHOUT first creating the fifo video_fifo. In this special case, if you press a button, what it the response of video_control.py? What is the response of mplayer? Will video_fifo be correctly created? Will control of mplayer be correct?

If the `start_video` script is run without first creating the FIFO (`video_fifo`), the communication between `mplayer` and `video_control.py` will fail.

- **Response of mplayer**: When `mplayer` is launched with the argument `-input file=./video_fifo`, it will try to open this file for reading. Since the file does not exist, `mplayer` will immediately exit with an error, such as "Cannot open FIFO /path/to/video_fifo". It will not wait for the file to be created.
- **Response of video_control.py**: The Python script will likely start without an issue. When a button is pressed, the script will attempt to open `video_fifo` to write a command. This will also fail because the file does not exist. The program will throw an exception (e.g., `FileNotFoundError`) and likely crash, unless it has specific error handling to deal with this situation.
- **Will video_fifo be correctly created?**: No. Neither `mplayer` nor the Python script (in a standard implementation) will create the FIFO. A named pipe must be explicitly created using a command like `mkfifo video_fifo` before the communicating processes attempt to use it.
- **Will control of mplayer be correct?**: No. Since `mplayer` would have already exited with an error and `video_control.py` would crash upon a button press, there is no communication and therefore no control over video playback.

9. While using the PiTFT buttons to control video, you may have encountered a problem with some of the buttons. It might appear that you had several 'hits' of the buttons when you pressed the button only once. There are several causes for this issue; what is one problem that could cause this issue?

One of the most common problems that causes multiple "hits" from a single button press is **switch bounce**.

When a mechanical button is pressed, the metal contacts inside do not make a clean, single connection. Instead, they "bounce" against each other several times on a microscopic level before settling into a stable, closed state. This bouncing happens very quickly (over a few milliseconds). A fast microcontroller like the one on the Raspberry Pi can read these rapid open/closed transitions as multiple, separate button presses.

Other potential causes include:

- **Floating Input Pin**: If the GPIO pin is configured as an input but is not connected to a pull-up or pull-down resistor, it is said to be "floating." In this state, the pin's voltage level is undefined, and it can be susceptible to electromagnetic interference, causing it to randomly register HIGH and LOW states, which could be interpreted as button presses.
- **Software Polling Issues**: If the software is checking the button's state in a very tight loop without any delay, it might read the button's state multiple times during the single, brief moment the user's finger is on the button, leading to multiple registered presses.

10. Describe pygame screen elements including a surface and a rect. How are these used to animate an image? Draw a step-by-step diagram illustrating the process you would use in PyGame to animate 2 frames of on the PiTFT.

- **Surface**: A Pygame `Surface` is a blank canvas or an image that you can draw on. Everything displayed in Pygame is drawn onto a Surface. The main screen itself is a special Surface created by `pygame.display.set_mode()`. Other Surfaces can be created for images, text, or other graphical elements.
- **Rect**: A `Rect` (or rectangle) is an object that stores coordinates and dimensions (x, y, width, height). It does not contain any pixel data itself but is used to define the position and size of a Surface on the screen.

These two elements are used together to animate an image. Typically, an image is loaded onto a Surface, and a Rect is used to control its position. In each frame of the animation loop, you would:

1. Fill the main display Surface with a background color to erase the previous frame.
2. Update the position of the Rect object.
3. "Blit" (draw) the image Surface onto the main display Surface at the location specified by the Rect.
4. Update the entire display to show the newly drawn frame.

Step-by-step Diagram for a 2-Frame Animation:
Description of the Diagram Steps:

1. **Initialization**:
   - Initialize Pygame and set up the display Surface (the screen).
   - Load two image files (`frame1.png`, `frame2.png`) into two separate Surface objects.
   - Create a list or array containing these two Surfaces.
   - Initialize an index variable (e.g., `frame_index = 0`) to keep track of the current frame.
2. **Start Main Loop**: This loop runs continuously to create the animation.
3. **Handle Events**: Check for user input, such as closing the window.
4. **Erase Screen**: Fill the main display Surface with a background color (e.g., black). This clears the previous frame's drawing.
5. **Select Current Frame**: Use the `frame_index` to select the current image Surface from the list of frames.
6. **Blit Frame**: Draw the selected frame's Surface onto the main display Surface at a specific position (defined by a Rect object).
7. **Update Display**: Call `pygame.display.flip()` or `pygame.display.update()` to make the changes visible on the screen.
8. **Update Frame Index**: Increment the `frame_index`. Use the modulo operator (`%`) to make it wrap around back to 0 after reaching the last frame. For a 2-frame animation, it would be `frame_index = (frame_index + 1) % 2`.
9. **Control Framerate**: Use a clock object (`pygame.time.Clock`) to control the speed of the loop, ensuring the animation runs at a consistent framerate. This introduces a small delay. The loop then repeats from step 3.

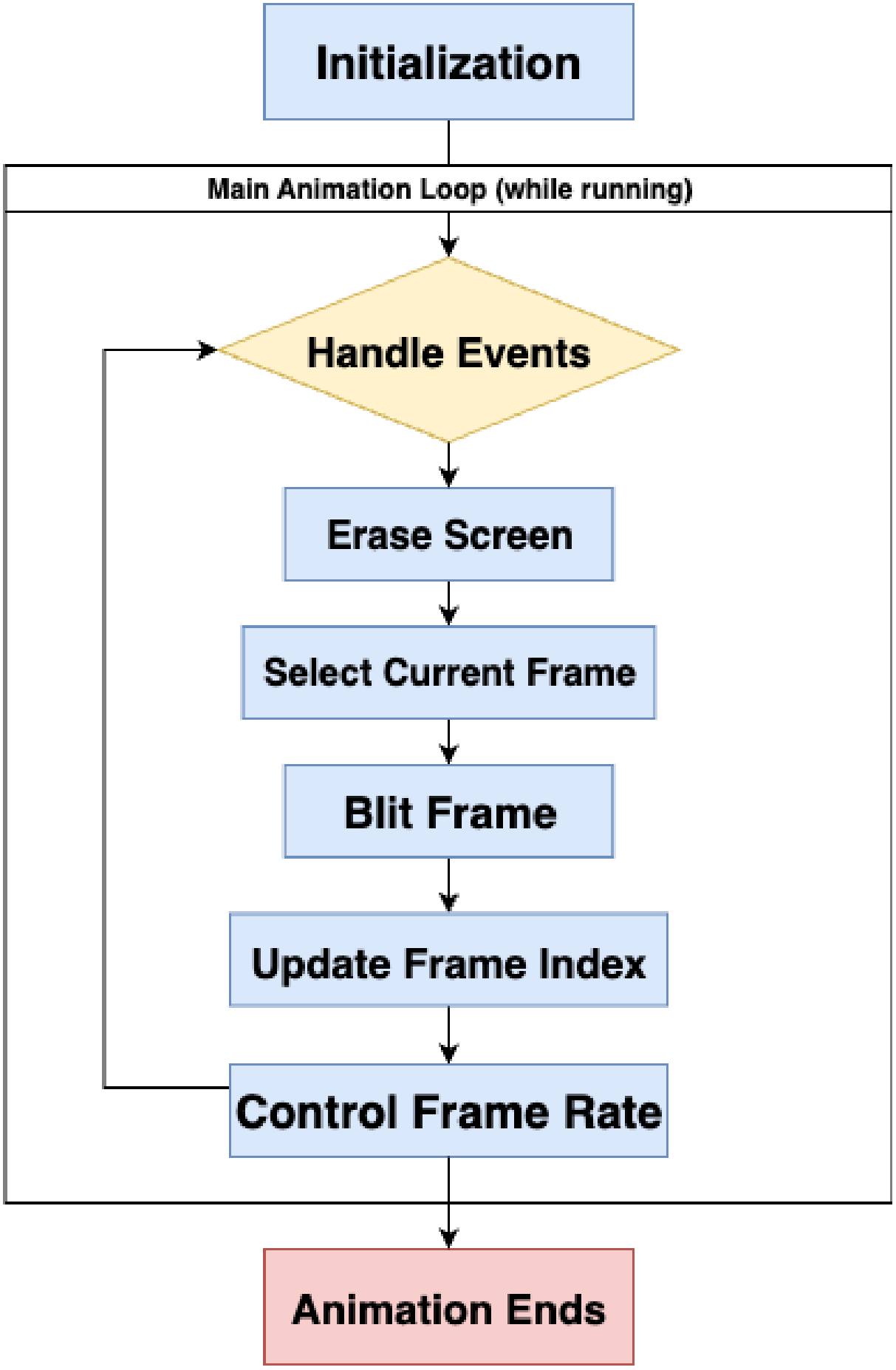

Figure 1: PyGame 2-Frame Animation Process