SWE4211 Lab 4: The Game of Anticipation

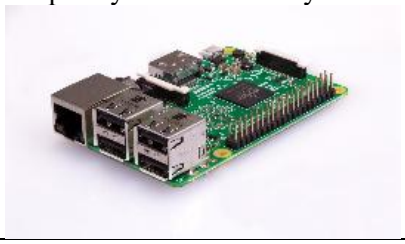
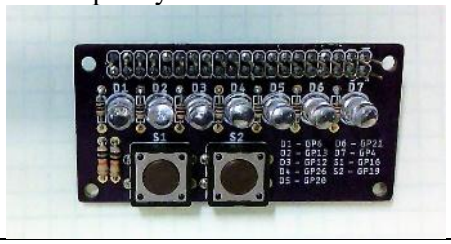
1 Introduction

Games are a source of joy and challenge. They give us the chance to test our skills against others, celebrate victories with grace, and show respect in defeat by honoring our opponents. It's no surprise that many software engineers are drawn to games—both as players and as enthusiasts. Yet, designing games is far from simple. The industry is fiercely competitive, demanding creativity, precision, and resilience. And perhaps that very challenge is what makes crafting games so exhilarating.

2 Lab Objectives

- Construct simple embedded software which controls GPIO devices.
- Use GPIO interrupts to manage a given software package.
- Construct software which uses multiple C++11 threads.
- Debug multithreaded programs using gdb on a remote host.
- Document C / C++ source code using Doxygen
- Generate documentation from your completed project

3 Laboratory Equipment Needed

One Raspberry Pi embedded systems board 	1 Raspberry PI Custom GPIO Hat 
Ethernet networking cables, as is applicable 	1 USB Power Supply 
	USB to Ethernet adapter 

4 Prelab

Before the lab, be sure to watch the video demonstrating the lab in action. (Yes, it's a thriller!)

In preparation, we'll need to install a couple of packages on our virtual machines. To begin, run the command **`sudo apt-get install doxygen`** to install Doxygen—a documentation tool for Linux. Make sure you're connected to the network, as the installation will take approximately 30 to 90 seconds.

Next, install Graphviz using the command **`sudo apt-get install graphviz`**. This package is necessary for generating certain diagrams. Each partner should complete these installations to ensure their VM is up to date.

5 Deliverables / Submission

Once you've completed the assignment, commit your final version of the code and push it to your GitHub Classroom repository. Be sure to include all commits. Your code must compile cleanly using the Makefile generated by CMake.

Next, record a brief video using your phone or PC that demonstrates the game functioning correctly. This video should capture a single execution of the game. Store the video in an accessible location and include a link to it in your submission.

Finally, submit a lab report as a single PDF file to Blackboard. Your report should include the following sections:

1) Title Page
   a) Assignment name
   b) Submission date
   c) Name of your GitHub Classroom repository
   d) Link to your demonstration video
2) Introduction
   a) Clearly describe the purpose of the lab in your own words.
   b) Explain what you aimed to accomplish, using multiple grammatically correct sentences.
3) Screen Captures and Data
   a) Include a screenshot showing your software building without warnings.
   b) Perform a make clean followed by make all in Eclipse or the terminal.
   c) Label and explain each screenshot.
4) Program Verification
   a) Provide a screenshot of the program running and measuring response time using the oscilloscope.
   b) Describe the steps you followed to verify timing accuracy.
   c) Include a screenshot of htop while the program is executing, showing CPU utilization.
   d) Use filters to limit visible tasks so the game process is clearly identifiable.
5) Discussion
   a) Answer each of the following questions in several well-written sentences:
      i) The game is designed for two players. Beyond hardware limitations, how challenging would it be to scale the game to four or more players from a software perspective?
      ii) Was it easier or harder to implement the game using multithreading compared to a single-threaded approach?
      iii) Where does the greater latency occur—in the software or in the human reaction to the LED?
      iv) Based on the system design, how does the software impact the measured latency? Where is latency introduced? Using class demos, estimate how much latency is due to the system versus human reaction time.
      v) How difficult was it to generate documentation using Doxygen? How does this compare to generating JavaDoc?
6) Successes and Challenges
   a) Describe what went well during the lab.
   b) Discuss any issues or obstacles you encountered.
7) Conclusion
   a) Reflect on what you learned from this experience.
   b) Suggest improvements for future iterations of this lab.

Reminder: Submit all materials as a single PDF file to Canvas.

6 Designing our hardware

The first part of this lab involves building out our hardware. Your embedded software will be performing the role for the software shown in Figure 1.

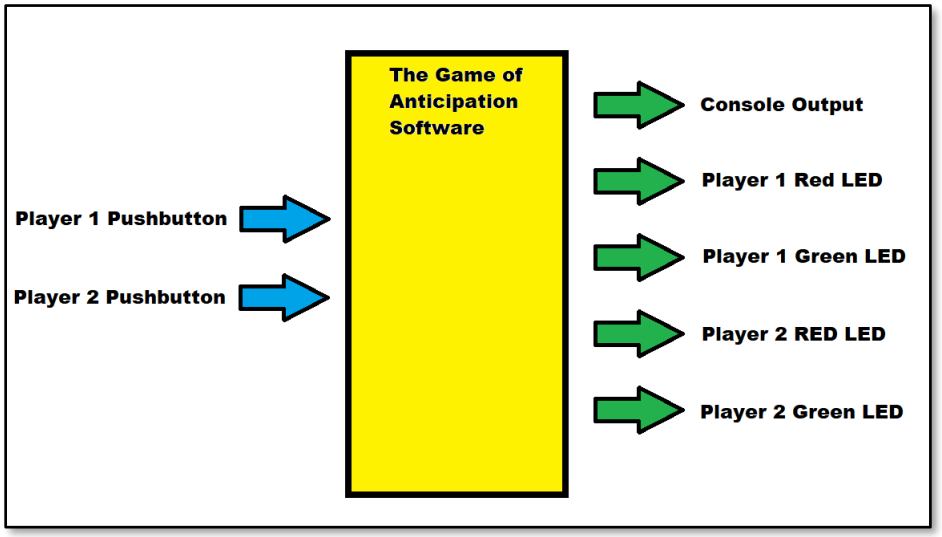


Figure 1: System Block Diagram

You will be using the same hardware that you used last week, namely the custom GPIO Hat shown in Figure 2. Note that the GPIO connections for each of the LEDs and switches are printed on the device.

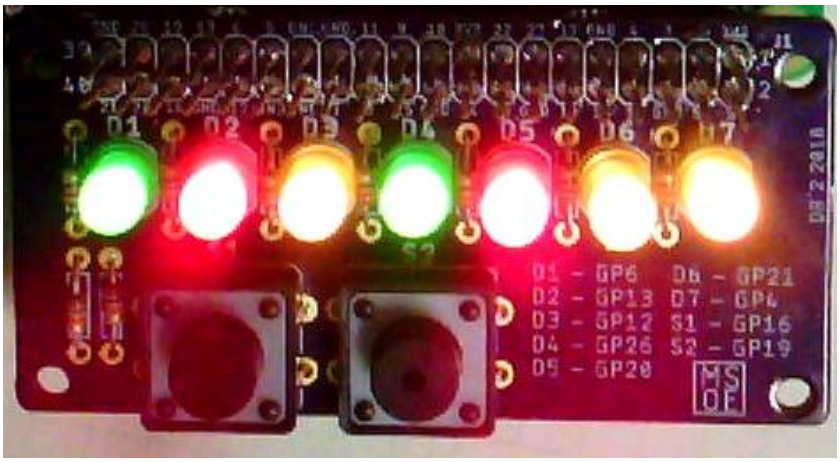


Figure 2: GPIO Board

7 The game specifics

The game begins with both LEDs turned off. Each player is assigned a red and green LED, along with a dedicated push button. As the program starts, both players' red LEDs will illuminate.

At a random moment, one player's red LED will switch off while their green LED turns on. That player must then press their button as quickly as possible in response to the green LED. The game records their reaction time, and once the button is pressed, the green LED turns off, the red LED turns back on, and the latency is displayed on the screen.

Meanwhile, the same logic runs concurrently in a separate thread for the second player. Any button presses made before the green LED activates are ignored. Button input should be handled using a blocking read approach to ensure accurate timing.

This sequence repeats 10 times for each player. After all rounds are completed, the player with the lowest total latency—aka the "fastest button pusher in the West"—is declared the winner, and their results are printed to the console.

8 Designing our software

The first step is to design your software. To make this process a bit easier, you've been provided with a basic UML diagram for the project, shown in Figure 3. You'll also need to determine how to measure time on the Raspberry Pi. *(Hint: look into* struct timespec*.)* You can explore its usage by typing **`man clock_gettime`** in the Unix shell.

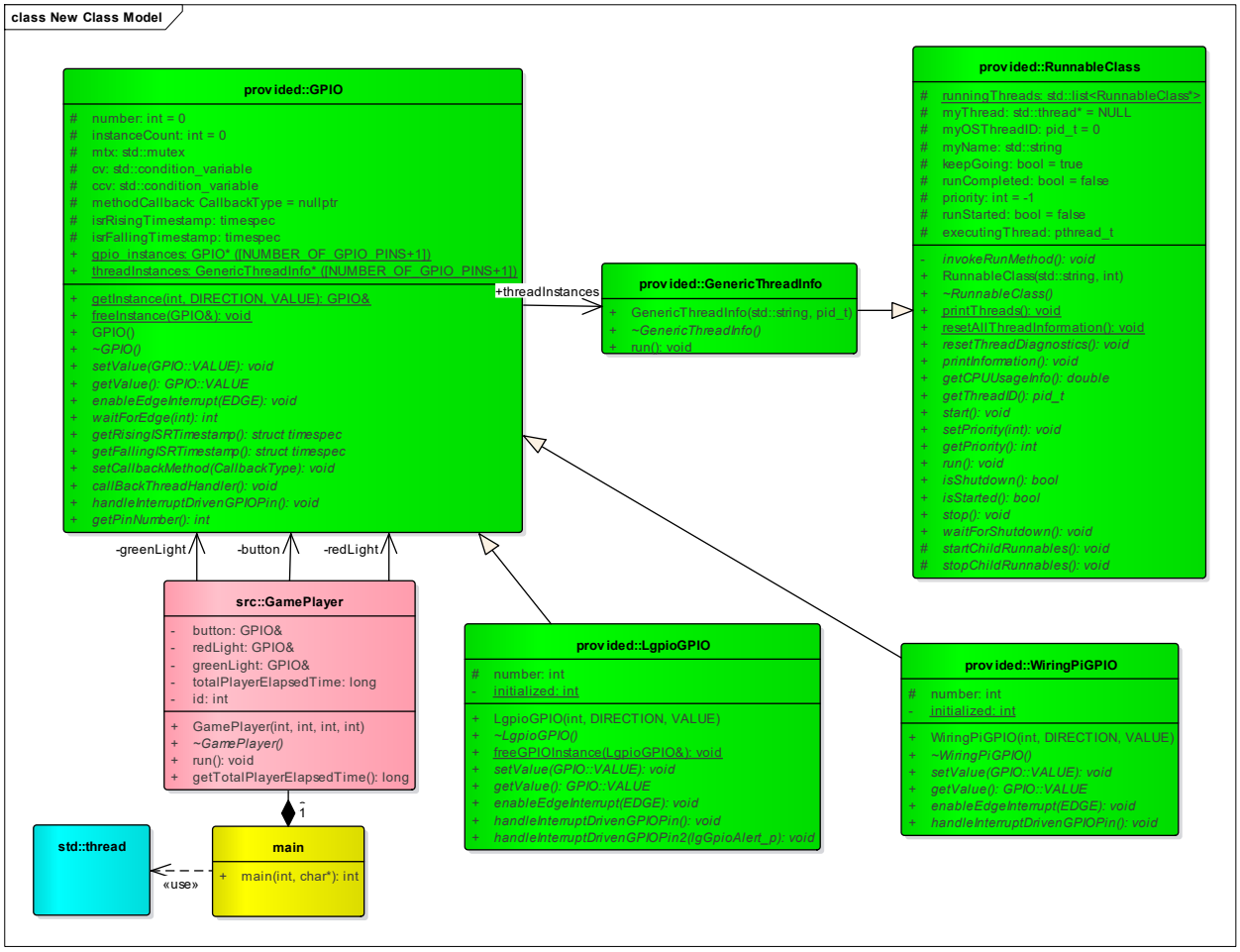


Figure 3: Basic UML for the project. Green files are provided to you. Pink files require you to implement. Yellow files may require modification. Blue files are part of std c++. (Note there may be a slight difference in names between this diagram and the template code for the class.)

Once you've completed the initial setup, the next step is to define the behavior of each method. The run method serves as the core of the game - it's where the primary logic is executed. To manage concurrent execution effectively, you'll need to incorporate thread synchronization constructs. (Hint: std::thread::join was covered in your operating systems course.)

A good strategy is to first implement the full game logic for a single player. Once that's functioning correctly, you can extend the design to support a second player.

Comprehensive documentation for the program can be found in the baseline_docs folder within the repository.

9 Documenting your code

One of the key aspects of documenting C/C++ code is clearly describing parameters and methods. While there are several ways to do this, a widely used tool is Doxygen, which is very similar to Javadoc and shares many of its conventions.

Doxygen may not be installed on your virtual machine by default. To use it, you'll need to install it using apt-get, a command-line tool for managing packages on certain Linux distributions. Open a terminal and run the following command:

```
sudo apt-get install doxygen
```

Make sure you're connected to the network, as the installation typically takes around 30 to 90 seconds. Afterward, install Graphviz with the command:

```
sudo apt-get install graphviz
```

Graphviz enables Doxygen to generate visual representations of your source code, such as call graphs, dependencies, and other diagrams.

Once enabled, you can add Doxygen comments to your code just as you would with Javadoc in Java. In your projects, be sure to update the author tag for all code you write during lab. Example files are shown in Figure 7 showing source code comments. In your projects, you will want to make sure that the author tag is updated for all code that you construct during lab.

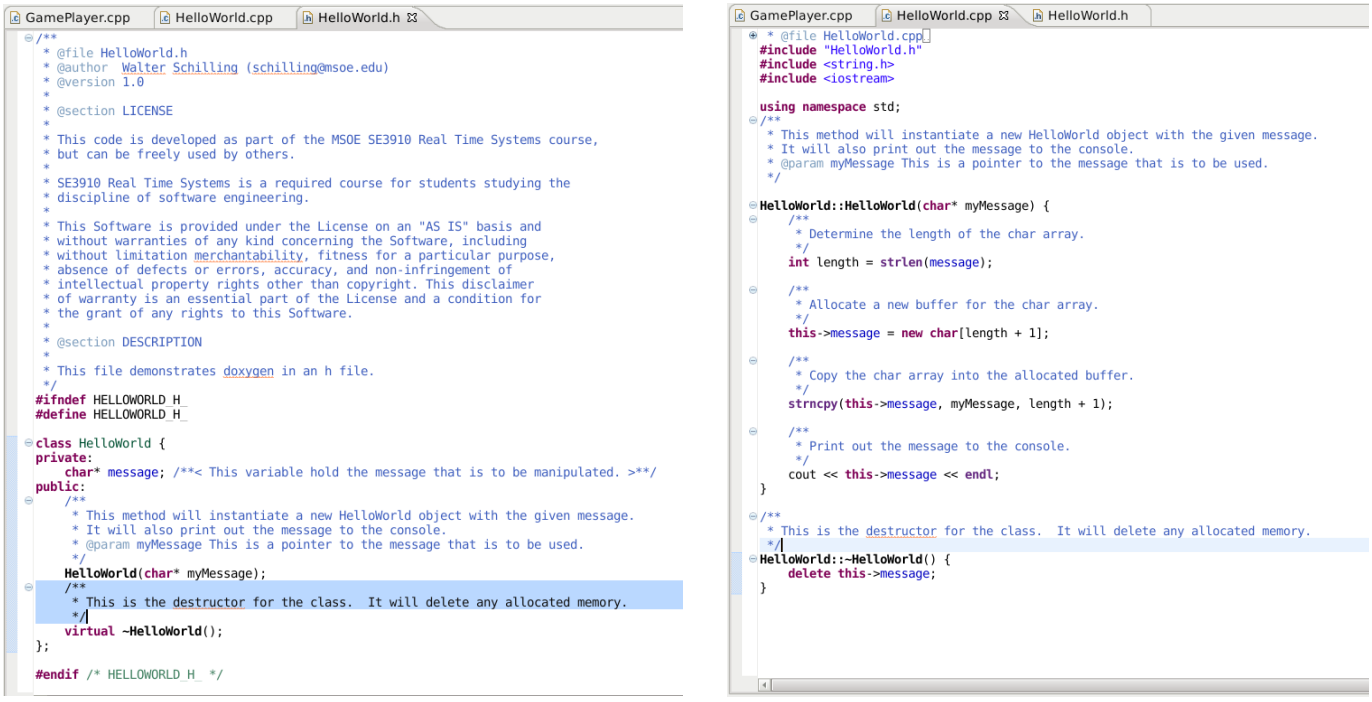


Figure 7: Sample files.

The cmake files for this project are automatically setup to construct the project in a manner that will allow you to easily create documentation on your project. To do this, open up a shell and change into the build directory of the project. From inside of this folder, issue the command `./createDocs.sh` which will execute the script that will make the documents

(internally the command `cmake --build . --target docs` runs). This will create a set of browsable html files for your project. Even though HelloWorld is not part of your formal project (i.e. you are not required to write the code for it), the generated documentation is provided to you as an example, and the file is given for reference purposes. You developed files should have generated comments that would closely match those provided to you in the documentation files. You may also find that in the build directory there is a script file which will create the documents for you. This usage of shell scripts saves you from having to remember somewhat obscure commands and options.

It is customary that the docs folder be added to your repository and committed once you have created it, as it defines how the project is structured. This way, future developers always have access to your documentation through a common location. On future assignments, you will be given the docs folder. Here you have been given the folder baseline_docs so you can see the generated documentation that your files should generate, except that names should reflect your name in all files where relevant.

10 Debugging your code[1]

10.1 Debugging on the Raspberry Pi

Until now you have built your code in the virtual machine and simply run it on the Raspberry Pi. In this lab you will need to do more than run it. You will need to stop the program in the middle of execution, look at the state of each thread, and step through code line by line. This section explains how that works and how to use the **pidebug.sh** script that sets it up for you.

10.2 How remote debugging works

Your program has to execute on the Pi, because only the Pi has the GPIO hat attached to it. Your source code, your symbol tables, and Eclipse all live in the VM. The debugger solves this by splitting itself into two halves that communicate over a TCP socket.

Piece	Runs on	Responsibility
gdbserver	Raspberry Pi	Launches your program, stops and resumes it, and reads and writes its memory, registers, and thread state. It knows nothing about your source code.
gdb (the cross debugger)	Your VM	Holds the symbol table from the binary you compiled, maps machine addresses back to source lines, and evaluates expressions. It never touches the hardware.
Eclipse	Your VM	A front end for gdb. Every button you click becomes a gdb command, and every value you see in a view came back from gdb.

Table 1: Division of labor during a remote debug session.

Because the two halves talk over an ordinary socket, anything that can carry bytes between the machines will work. We use an SSH tunnel. The option **-L 2345:localhost:2345** tells **ssh** to listen on port 2345 inside your VM and forward everything written there, through the encrypted SSH connection, to port 2345 on the Pi. Eclipse then connects to **localhost:2345** and never needs to know the Pi's address at all.

This is also why the script starts gdbserver as **gdbserver localhost:2345** rather than **gdbserver :2345**. The debug port is bound to the Pi's loopback interface, so nothing else on the lab network can connect to it. A debug port is a remote code execution service with no authentication of its own. The only way through is the SSH connection you authenticated.

Eclipse has built-in support for doing all of this itself, but its bundled SSH implementation no longer negotiates a key exchange that current Raspberry Pi OS accepts, so the connection fails before it starts. Rather than fight the IDE, we perform the three steps ourselves.

10.3 The three manual steps

Every debug session requires the same three things to happen, in order:

1. Copy the newly built binary from the VM to the Pi, using **scp** or **sftp**.
2. Open the tunnel: **ssh -L 2345:localhost:2345 pi@yourpi**
3. Start the program under the debug server on the Pi: **gdbserver localhost:2345 ./yourProgram**

Only then can you attach Eclipse. None of this is difficult, but you repeat it after every single edit-and-compile cycle, and the first step is the one everybody forgets. Debugging a binary that is not the one you just compiled is a special kind of misery: breakpoints land on the wrong lines, variables hold impossible values, and you lose an hour blaming your code. A

[1] Note: This section was developed with the help of the Claude AI tool.

repetitive sequence that is easy to get wrong is exactly what a shell script should own. In many ways, these are the same arguments made for the earlier for **createDocs.sh**.

10.4 What the script does

The **pidebug.sh** script performs all three steps in a single command. Read through it; it is short, and every section of it exists for a reason.

1. It loads your settings. If a file named **~/.pidebugrc** exists, the script reads the hostname, user, remote directory, port, and key from it, so you never type them.
2. It parses command-line options with **getopts**, so any setting can be overridden for one run without editing anything.
3. It checks before it acts. The local binary must exist, and the local port must be free. Failing immediately with a clear message beats failing halfway through in a way you have to diagnose.
4. It transfers the binary, creates the remote directory, marks the file executable, and kills any gdbserver left over from a previous session. That leftover process is the most common cause of a mysterious "address already in use" error.
5. It builds the remote command, quoting any arguments you pass to your program with **printf %q** so that they survive being handed to a second shell on the Pi.
6. It opens the tunnel and starts gdbserver in one **ssh** invocation. Because both live on the same connection, they come up together and go away together; there is no half-running state to clean up by hand.
7. It installs a **trap** so that when you press Ctrl-C, the script reaches back over SSH and stops gdbserver before it exits.

The first line of the body, **set -euo pipefail**, makes the shell exit on an unhandled error, on the use of an undefined variable, and on a failure anywhere in a pipeline. Without it, a failed copy would be followed cheerfully by a debug session against last week's binary. It is worth putting at the top of every script you write.

10.5 Using the script

Do this part once. Make the script executable:

```
chmod +x pidebug.sh
```

Then create the file **~/.pidebugrc** containing the details of your own Pi, substituting your hostname and account:

```
PI_HOST=schillingpi3u61
PI_USER=pi
PI_DIR=/home/pi/debug
GDB_PORT=2345
```

After that, one command per debug session, run from your project directory:

```
./pidebug.sh build/AnticipationGame
```

To run the program as root on the Pi, add **-s**. To pass arguments to your own program, put them after a double dash:

```
./pidebug.sh -s build/AnticipationGame -- 10
```

The script stays in the foreground and prints the output of gdbserver and of your program. Leave that terminal open for the whole session. Pressing Ctrl-C in it ends the session, stops gdbserver, and closes the tunnel. Run **./pidebug.sh -h** for the full list of options.

10.6 Connecting Eclipse

Start the script first, wait for gdbserver to report that it is listening, and then configure and launch the debug session in Eclipse:

1. Select Run → Debug Configurations, then create a new C/C++ Remote Application configuration.
2. At the bottom of the Main tab, click "Select other..." and choose the GDB (DSF) Manual Remote Debugging Launcher. This is the launcher that connects to a debug server that is already running; the default one would try to copy and start the program itself, which is what we are doing in the script.
3. Set the C/C++ Application field to the same local binary you handed to the script. Eclipse reads your symbols and source line numbers from this copy, so it must be the build that was pushed to the Pi.
4. On the Debugger tab, set the GDB debugger to your cross debugger — the one that understands ARM binaries, not the VM's native **gdb**.
5. On the Debugger tab, open Connection and set the type to TCP, the host name to **localhost**, and the port to **2345**. You are connecting to your own machine; the tunnel does the rest.
6. Click Debug. Execution will stop at the start of main, and from there the debugger behaves exactly as it did in your Java and desktop C++ courses.

Because gdbserver exits when your program does, you run the script again for each debug session. Rebuild, rerun the script, reconnect.

10.7 Running as root

Access to GPIO interrupts generally requires root privileges, depending on which library back end your build uses and how the pins are exported. If your program works when you run it from an SSH session with **sudo** but fails under the debugger, this is almost certainly why: gdbserver launched it as an ordinary user. Add the **-s** option and gdbserver, and therefore your program, will run as root.

Be deliberate about this. Running under **sudo** means your program, bugs included, has unrestricted access to the machine, and a stray pointer can take down more than your process. Use it when the hardware requires it, not as a reflex, and make sure the hostname in your configuration file really is your own Pi.

10.8 Debugging threads, and a warning about timing

Your game runs each player in its own thread, and the debugger sees all of them. In the Eclipse Debug view, every thread appears with its own call stack, and clicking one switches the Variables view to that thread's context. In the gdb console, **info threads** lists them and **thread 2** switches between them. This is the fastest way to find out whether a thread is blocked on a read, waiting on a lock, or has quietly exited.

One caution matters more in this course than in any other. By default gdb runs in all-stop mode: when any one thread hits a breakpoint, every thread in the process is frozen. That is what you want for inspecting state, and it is fatal to any measurement. A reaction time recorded while the program sat at a breakpoint is not a measurement of anything. Use the debugger to find logic and synchronization errors; collect the latency values you report, and your oscilloscope captures,

from a normal run with no debugger attached. Mistaking instrumented timing for real timing is a classic and expensive error in real-time work.

10.9 Troubleshooting

Symptom	Likely cause and fix
The script reports that the local port is in use.	A tunnel from an earlier session is still open. Close that terminal, or run pkill -f "ssh.*2345", or pick another port with -p and use the same port in Eclipse.
Eclipse reports that the connection was refused.	gdbserver is not listening. Check the terminal running the script; the program may have exited, or the launch may have failed.
Breakpoints never hit, or stop on the wrong lines.	The binary on the Pi is not the one Eclipse has symbols for. Rebuild, then run the script again so the new binary is transferred. Do not use -n after a rebuild, and be sure the build is not stripped and was compiled with debug symbols.
The program fails on a GPIO operation only under the debugger.	Insufficient privileges. Rerun the script with the -s option.
gdbserver: command not found.	Install it on the Pi with sudo apt-get install gdbserver.
SSH warns that the host key has changed.	The Pi was reimaged. Remove the stale entry with ssh-keygen -R followed by the hostname, then reconnect and accept the new key.

Table 2: Common remote debugging problems.

11 Setting up passwordless SSH access

The debug script runs several commands on the Pi: it clears out any stale debug server, copies your binary across, and then opens the tunnel and launches gdbserver. If the Pi is authenticating you with a password, every one of those is a separate login and you will be asked for that password several times per debug session. Two mechanisms remove the nuisance entirely, and both are worth knowing well beyond this lab.

11.1 Public key authentication

Rather than proving who you are with a secret that both machines know, you prove it with a key pair. Generate one in your virtual machine:

```
ssh-keygen -t ed25519
```

Press Enter to accept the default location. You now have two files in **`~/.ssh`**. The file **`id_ed25519`** is your private key and should never leave the VM. The file **`id_ed25519.pub`** is the public half, and it is safe to hand to anyone. Copy that public half to the Pi:

```
ssh-copy-id pi@yourPi
```

That command asks for the Pi's password one last time and appends your public key to **`/home/pi/.ssh/authorized_keys`** on the Pi. From then on **`ssh`** proves your identity by signing a challenge with your private key. There is nothing for you to type and nothing crossing the network for someone else to capture. Do this before you try the script's **`-b`** option, which detaches from your terminal and therefore cannot prompt for a password at all.

If you give the key a passphrase, let **`ssh-agent`** hold the decrypted key for the length of your desktop session so you type the passphrase once:

```
ssh-add ~/.ssh/id_ed25519
```

Keep the private key where it was generated. Copying private keys from machine to machine is the most common way that key based authentication quietly stops being an improvement over passwords. If you need access from a second machine, generate a second key there and copy its public half to the Pi as well.

Neither of these techniques is specific to a Raspberry Pi. Key based authentication and connection reuse will serve you in every SSH based workflow you meet after this course, from pushing to a Git remote to reaching a build server or a cloud instance.

12 Testing your code

Once you've finished writing the program, it's important to verify the accuracy of your timing calculations. Use an oscilloscope to measure the interval between the moment the LED turns on and when the button is pressed. This measured time should closely align with the latency output printed by your code.

Be sure to capture a screenshot showing both the game in action and the oscilloscope display to demonstrate the match between the physical measurement and your software's output.