

COM1032 Operating Systems

Lab 3

OS Process Scheduling and Communication

Purpose:

The purpose of this lab session is to familiarize yourself with Operating System Process Scheduling Algorithms in general using a simulator, Linux/Raspbian Scheduling algorithms particularly, and familiarize yourself with some process communication protocols.

Aim

By the end of the lab you will be able to:

- Differentiate the different process scheduling algorithms and their performance.
- Monitor process scheduling in Raspberry Pi and control programmatically using delays commands.
- Identify Java Process Communication APIs.

Process Scheduling Simulator

Download from SurreyLearn in week 3 lab section the process scheduling simulator (pss.jar). The simulator takes an input file describing the processes to add to the ready queue, and simulate their running using the selected scheduling algorithms, and generate some performance metrics and charts. You are required to configure the input file based on your URN numbers and run your own simulations. Try to understand the resulting chart for each algorithm explaining the resulting performance based on your understanding of the input configuration parameters and the selected algorithm. These scheduling algorithms are as follows:

- - First come first serve (FCFS)
- - Shortest job first (SJF)
- - Round robin (RR)
- - Highest Priority First static priority, preemptive (HPFSP)
- - Weighted round robin (WRR)

The format of the input file is as follows:

1. First two lines are for you to write comments
2. Third line is the number of processes, we will identify as x. Configure your simulation to run for 8 processes.
3. For every process from the first to x, include a line in the input file. This file format require each line to contain the following items separated by spaces:
 - a. Serial number for the process beginning from 1 up to x in the last line.

- b. The time the process arrived at the ready queue with accuracy up to one digit after the decimal point. This would be a.b. We can use the following values for the 8 processes: 6.2, 2.4, 8.5, 9.7, 1.2, 4.5, 7.4, and 5.8 for the last process.
 - c. The time the process is expected to run for (CPU Burst): y. For this simulation, we will consider the following values, 60, 20, 40, 10, 90, 80, 30, and 50 for the last process.
 - d. The process priority p, For this simulation, we will consider the following values, 8, 6, 2, 5, 7, 4, 3, and 9 for the last process.
4. From the GUI screen choose your quantum value q, such as 10.

Hints: The graph shows the number of processes on the y-axis. The timeline of the processor is shown in the x-axis. A horizontal line in the graph is the set of points from the time the process is scheduled (added to the ready queue), to the time it is either terminated or interrupted. Each of these lines are horizontal in the graph and corresponding to the process number and represent a visualisation for the time the process is running in the processor. Try to understand the performance that you simulated. Focus on examples of the process turnaround time for some algorithms and check if you can manually calculate the resulted average turnaround time for the algorithm and the total order of execution. Try to manually visualise what would have changed if half of your processes blocked for I/O once during their execution giving an example of any order. For the 8 processes, if the first 4 processes stopped for I/O half-way through their execution, how the average turn-around time and final order of execution will change?

PSS generates three text files in the output folder grouped in subfolders named after the algorithm abbreviated name. The first file is just an output of the configuration parameters. This file can help you verify that you configured your input file and initial values of the context switching and quantum correctly. The second file is a log of all state transitions occurred to every process. The third results file is where the performance metrics are calculated. It shows the timing every process started on the processor, and the time it finished, and hence calculates the turnaround time as the arrival time subtracted from the finish time. The weighted turnaround time for every process is its turnaround time divided by the expected run time (CPU Burst) that you defined in the input file as y. The average turnaround time is the total turnaround time for all processes divided by the number of processes. Same about average weighted turnaround time is the total for all processes divided by the number of processes.

Process Scheduling in Raspbian OS

From the command line you can list the running processes as you did in Week 1 lab to avoid kernel mode code such as the one included in the second lecture. The `ps` (short for process status) command is used to list processes currently running on your Raspbian system. It can accept a lot of options that can come in handy when troubleshooting your system. Check these options using `man ps`. For example: `ps -A` lists all processes, `ps aux`¹ to list all processes along with the username of the process's owner, CPU loads, the starting time of the process, the command that initiated the process, etc.

¹ `ps aux` options are: a for all, u for user friendly output, and u for including processes not started from a user terminal.

```

pi@new-hostname ~ $ ps aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.1  0.3  2148  1360 ?        Ss   12:03   0:01 init [2]
root         2  0.0  0.0      0     0 ?        S    12:03   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    12:03   0:00 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   12:03   0:00 [kworker/0:0H]
root         7  0.0  0.0      0     0 ?        S    12:03   0:00 [rcu_preempt]
root         8  0.0  0.0      0     0 ?        S    12:03   0:00 [rcu_sched]
root         9  0.0  0.0      0     0 ?        S    12:03   0:00 [rcu_bh]
root        10  0.0  0.0      0     0 ?        S<   12:03   0:00 [khelper]
root        11  0.0  0.0      0     0 ?        S    12:03   0:00 [kdevtmpfs]
root        12  0.0  0.0      0     0 ?        S<   12:03   0:00 [netns]
root        13  0.0  0.0      0     0 ?        S<   12:03   0:00 [perf]
root        14  0.0  0.0      0     0 ?        S    12:03   0:00 [khungtaskd]
root        15  0.0  0.0      0     0 ?        S<   12:03   0:00 [writeback]
root        16  0.0  0.0      0     0 ?        S<   12:03   0:00 [crypto]
root        17  0.0  0.0      0     0 ?        S<   12:03   0:00 [bioset]
root        18  0.0  0.0      0     0 ?        S<   12:03   0:00 [kblockd]
root        19  0.0  0.0      0     0 ?        S    12:03   0:01 [kworker/0:1]
root        20  0.0  0.0      0     0 ?        S<   12:03   0:00 [rpciod]
root        21  0.0  0.0      0     0 ?        S    12:03   0:00 [kswapd0]
root        22  0.0  0.0      0     0 ?        S    12:03   0:00 [fsnotify_mark]
root        23  0.0  0.0      0     0 ?        S<   12:03   0:00 [nfsiod]
root        29  0.0  0.0      0     0 ?        S<   12:03   0:00 [kthrotld]
root        30  0.0  0.0      0     0 ?        S<   12:03   0:00 [VCHIQ-0]
root        31  0.0  0.0      0     0 ?        S<   12:03   0:00 [VCHIQr-0]
root        32  0.0  0.0      0     0 ?        S<   12:03   0:00 [VCHIQs-0]
root        33  0.0  0.0      0     0 ?        S<   12:03   0:00 [iscsi_ah]

```

Here is a description of each column:

USER – the user who owns the process (the user pi in this case).

PID – process ID of the process (2570).

%CPU – the CPU time used divided by the time the process has been running.

%MEM – the ratio of the process’s resident set size to the physical memory on the machine.

VSZ – virtual memory usage of entire process (in KiB).

RSS – resident set size, the non-swapped physical memory that a task has used (in KiB).

TTY – controlling tty (terminal).

STAT – multi-character process state.

START – starting time or date of the process.

TIME – cumulative CPU time.

COMMAND – the command used to start the process (tail -f /var/log/messages).

The Linux kernel exposes some process metadata as part of a virtual file system. Let’s look in the /proc directory on your Linux system:

```

cd /proc
ls

```

You should see a list of directories, many of which will have names that are integers. Each integer corresponds to a pid, and the files inside these pid directories capture information about the relevant process. For the full list of the content of these directories, execute `man 5 proc` at a Linux terminal prompt. The /proc/[pid] files are not ‘real’—look at the file sizes with `ls -l`. These pseudo-files are not stored on the persistent file system: instead, they are file-like representations of in-memory kernel metadata for each process.

The virtual files associated with a process in /proc/[pid]/ include the following list:

`cmdline` The textual command that was invoked to start this process

`cwd` A symbolic link to the current working directory for this process

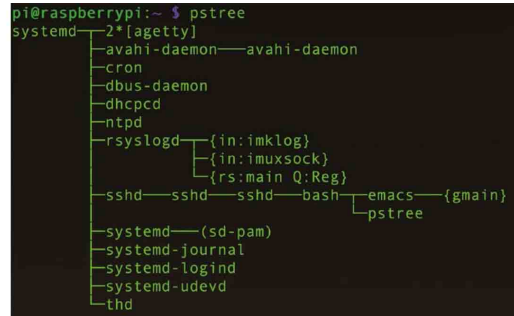
`exe` A symbolic link to the executable file for this process

`fd/` A folder containing file descriptors for each file opened by the process

`maps` A table showing how data is arranged in memory

`stat` A list of counters for various OS events, specific to this process

The `ps` tool is another example utility—it displays similar information to our process family tree code outlined above, but `ps` uses the `/proc` pseudo-files rather than expensive system calls. The `ps` utility is part of the `psmisc` Debian package, which you may need to install explicitly. The following figure shows typical output from `ps`, for a Pi with a single user logged in via `ssh`.



Now you can observe your processes with the `ps` command. Use the `watch` tool to see how the states change over time.

```
watch ps u
```

You should see that some processes are running (R) and others are sleeping (S), waiting for I/O (D), stopped (T), or Zombie (a dead Process) (Z). Press `CTRL + c` to exit the `watch` program.

This quantum time value is specified on Raspberry Pi as 10ms. You can check the default value on your Linux system with:

```
cat /proc/sys/kernel/sched_rr_timeslice_ms
```

Raspberry Pi Expansion Kit

To run the following exercise, we will need to start using the Raspberry Pi Expansion Kit that you received in lab 1. You can do this using 2 different libraries that you can use. There is a lower level C library “bcm2835” that works directly on the GPIO and other IO functions on the Broadcom BCM 2835 chip directly that is faster. The details to follow the `bcm2835` installation and syntax is given in the document “lab3_bcm2835” document in SurreyLearn. WiringPi has both C and Python interface and is supported by the Freenove tutorials provided with the expansion kit, and it is what follows below.

You need to do the following steps to get started:

1. Download the Guide zip file from: <http://freenove.com/tutorial.html>. You have been given “FNK0019” kit, so this is what you need (Please note if you have another kit):

FNK0019 Freenove Super Starter Kit for Raspberry Pi [View](#) [Download](#)

Or by:

```
cd ~
git clone --depth 1 https://github.com/freenove/Freenove_Super_Starter_Kit_for_Raspberry_Pi
mv Freenove_Super_Starter_Kit_for_Raspberry_Pi freenove
```

2. In the extracted folder “freenove”, you will find “Read Me First” pdf require that you remove all chips and modules inserted into the breadboard before use.
3. From Chapter 7 “AD/DA” of Tutorial.pdf, you will need to enable I2C interface in raspberry pie. Type command in the terminal:

```
sudo raspi-config
```

4. Choose “5 Interfacing Options”->“P5 I2C”->“Yes”->“Finish” in order and restart your RPi later. Then the I2C module is started.
5. Type a command to check whether the I2C module is started:

```
lsmod | grep i2c
```

6. Type the command to install I2C-Tools.

```
sudo apt install i2c-tools
```

7. I2C device address detection:

```
i2cdetect -y 1
```

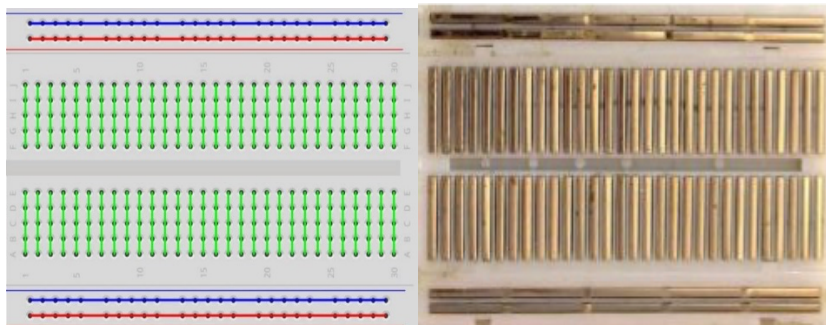
8. WiringPi is a GPIO access library written in C for the BCM2835/BMC2836/ BMC2837 used in the Raspberry Pi. It’s released under the GNU LGPLv3 license and is usable from C, C++ and many other languages with suitable wrappers (See below) It’s designed to be familiar to people who have used the Arduino “wiring” system. (for more details, please refer to <http://wiringpi.com/>). It should be in your system, but if not, install it as follows:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install wiringpi
```

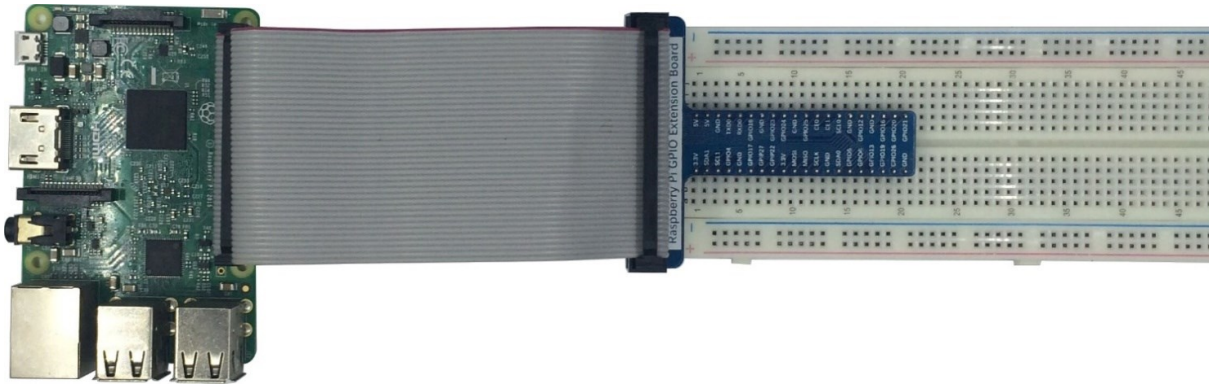
9. Run the gpio command to check the installation:

```
gpio -v
```

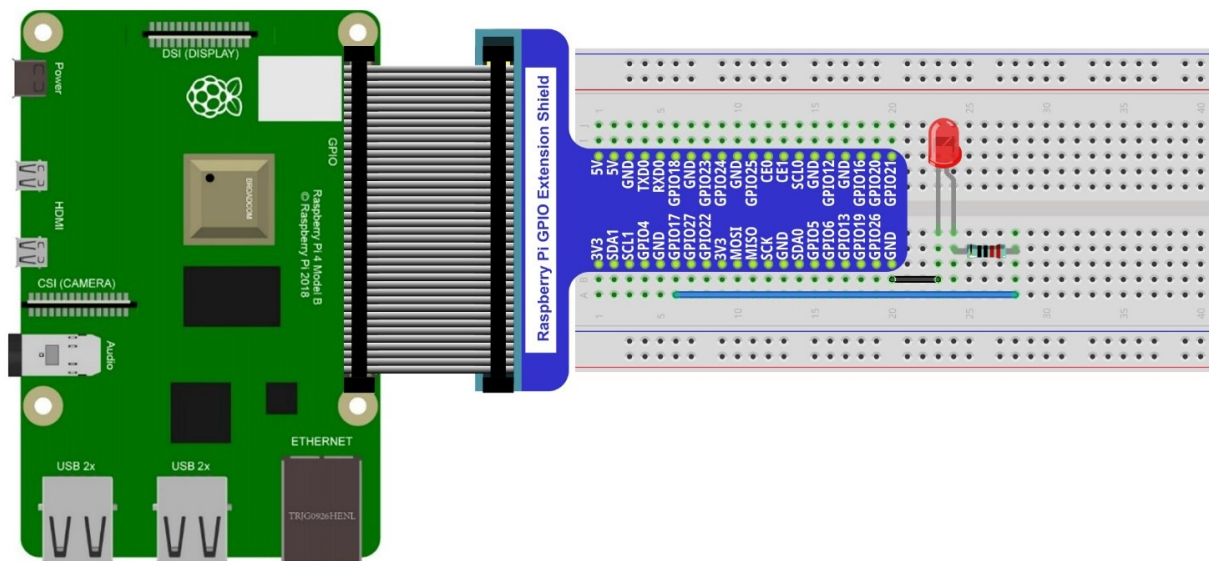
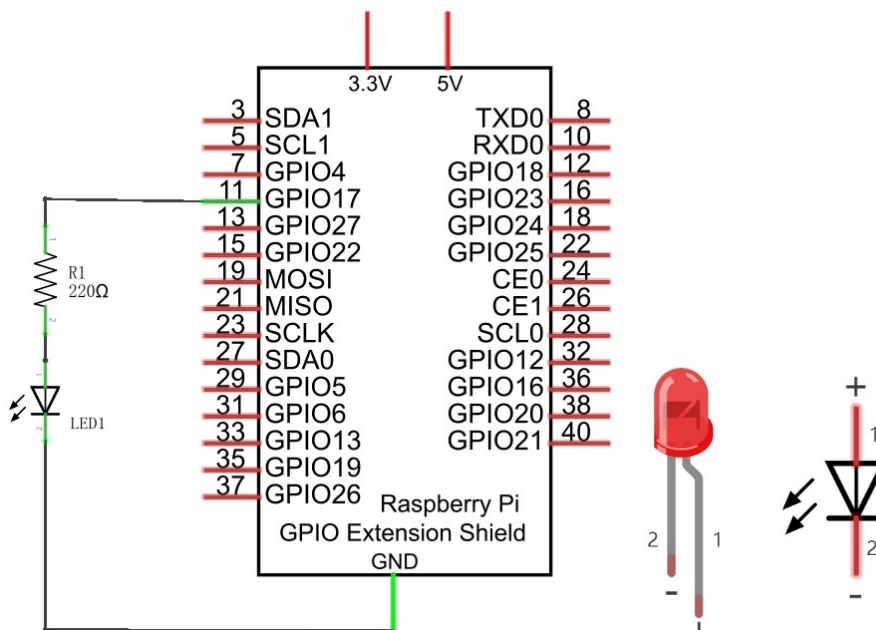
10. In the extracted folder, there is Code folder, in which there is C_Code subfolder, in which a number of C examples exist. There is also Python_Code subfolder for python examples.
11. From the C code, run the 01.1.1_Blink example by changing to its folder to find the source file.
12. First Connect the circuit as shown in the diagrams. First the board is internally connected as shown in the right-hand side picture:



13. Shut down Pi, and connect the expansion board as the diagram shows:



14. Continue the circuit as following diagram shows:



15. If you need to learn the Geany editor, instead of nano or vi, use the following command to open the Geany in the sample file "Blink.c" file directory path:

```
geany Blink.c
```

16. Use following command to compile "Blink.c" and generate executable file "Blink".

```
gcc Blink.c -o Blink -lwiringPi
```

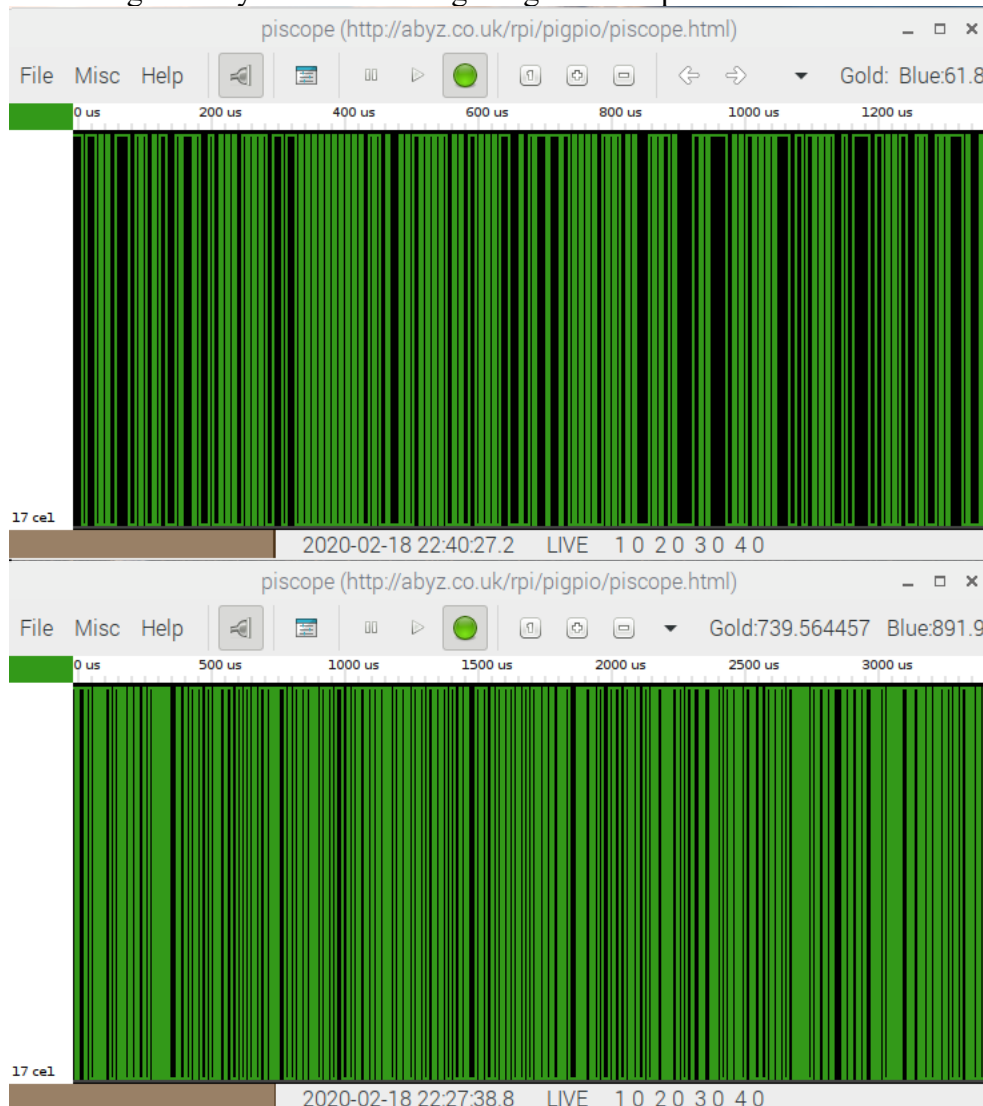
17. Then run the generated file "Blink".

```
sudo ./Blink
```

18. Now, LED start blink.

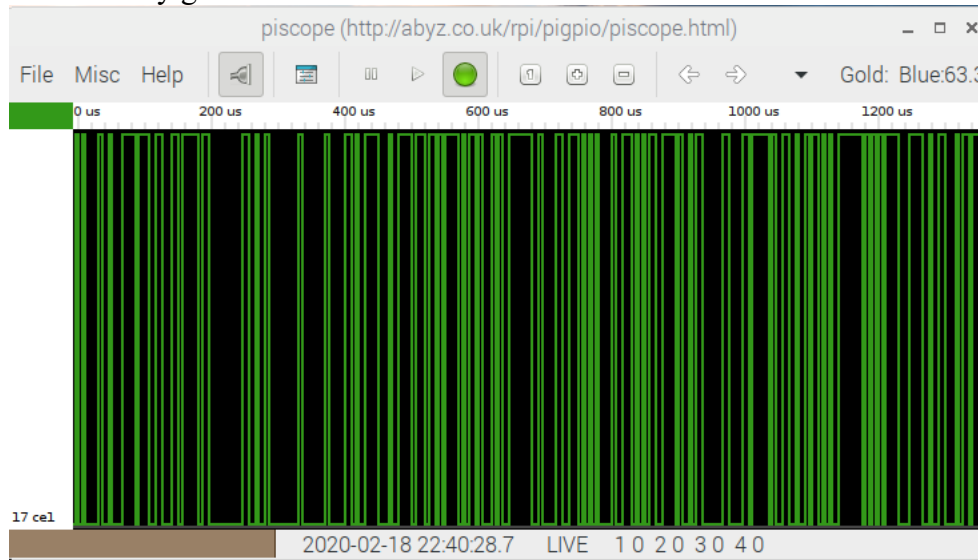
Use the Lab3_Monitoring" Document to install Raspberry monitoring packages such as RPi-Monitor, Piscope, and GTKWave. I used Piscope to produce the following screens to investigate how processes are interrupted by the OS.

Running the application at the command prompt and nothing running in the background gave something like any of the following images on the pin:



i.e. Very occasional process interruptions.

Now starting the game Minecraft and the browser in the background and any other tasks like another code that copies files from one location to another (I/O is usually slower), to make the RPi busy gave this:



This remained until the browser was fully loaded and less processes are running in the background at which point the pin output returned to the previous state.

Sleeping

Good applications tell the Linux scheduler when they are happy to be halted so another process can be run. The scheduler is a complex beast but we have found that simply using `delay(500)` in the WiringPi Library or `bcm2835_delay(500)` in the bcm2835 library in your programs main loop is enough to give the scheduler an opportunity to run another process and avoid it letting your application run but periodically halt it for a significantly longer time for many other processes to be run before handing execution back to your application. If your application needs constant fast monitoring of inputs or control of outputs then including this can help reduce the length of time your application is halted, assuming no other heavy process runs in the system (in which case the scheduler will decide how to split running time between you and the other process – e.g. switching every 10mS in the test above).

Guaranteeing a slice of time for critical IO operations

Another approach if you need to know you are not going to be interrupted by the scheduler, for instance say you want to generate RC servo PWM signals using GPIO pins, could be to just ensure you limit your operations to less than 10mS before calling `delay()` letting the scheduler know you are happy to sleep. This way you limit the maximum time your process needs to run before interruption to below the schedulers maximum time (often 10mS) and just accept that you don't know how long it will be before you get another slice of time, but at which point you are as sure as you can be that you will get the next block of time you need to complete another set of your IO operations. This may overcome some of the real time issues of working on top of an OS and its scheduler – 10mS is a lot of time to get IO operations done.

I will add some more on the configuration tools and options

Java Process Communication APIs

We will focus on Socket Programming and Message Passing.

Java Sockets:

This section of the lab will introduce you to socket programming, as an example for inter-process communication. As discussed in the lecture, other programming languages offer different syntax for the same concept. For example, Windows XP offer Local Procedure Call (LPC) to enable two processes on the same machine to communicate. Communication using sockets—although common and efficient—is generally considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application programmers to impose a structure on the data. Three other strategies for communication in client–server systems include sockets, remote procedure calls (RPCs), and Java’s remote method invocation (RMI). More on these APIs will be introduced in COM2038 Parallel Computing Module next year.

A socket is defined as an endpoint for communication. A pair of processes communicating over a network employ a pair of sockets — one for each process. A socket is identified by an IP address concatenated with a port number. In general, sockets use a client – server architecture. The server waits for incoming client requests by listening to a specified port. Once a request is received, the server accepts a connection from the client socket to complete the connection. For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a Web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the Web server. More on that will be introduced on COM2022 Computer Networking.

To explore socket programming further, we turn next to an illustration using Java. Java provides an easy interface for socket programming and has a rich library for additional networking utilities.

Java provides three different types of sockets. Connection-oriented (TCP) sockets are implemented with the Socket class. Connectionless (UDP) sockets use the DatagramSocket class. Finally, the MulticastSocket class is a subclass of the DatagramSocket class. A multicast socket allows data to be sent to multiple recipients.

The following example describes a date server that uses connection-oriented TCP sockets.

The operation allows clients to request the current date and time from the server. The server listens to port 6013, although the port could have any arbitrary number greater than 1024.

When a connection is received, the server returns the date and time to the client.

The date server source code is shown below. You need to start an eclipse project, and add this code in a file called “DateServer.java”;

```
import java.net.*;
import java.io.*;
public class DateServer {
    public static void main(String[] args) {
        try {
            ServerSocket sock = new ServerSocket(6013);
            // now listen for connections
            while (true) {
                System.out.println("Server started");
                System.out.println("Waiting for a client ...");
                Socket client = sock.accept();
                System.out.println("Client accepted");
                PrintWriter pout = new PrintWriter
                    (client.getOutputStream(), true);
                // write the Date to the socket
```

```

        pout.println(new
        java.util.Date().toString());
        // close the socket and resume
        // listening for connections
        client.close();
    }
}
catch (IOException ioe) {
    System.err.println(ioe);
}
}
}

```

The server creates a `ServerSocket` that specifies it will listen to port 6013. The server then begins listening to the port with the `accept()` method. The server blocks on the `accept()` method waiting for a client to request a connection. When a connection request is received, `accept()` returns a socket that the server can use to communicate with the client. The server first establishes a `PrintWriter` object that it will use to communicate with the client. A `PrintWriter` object allows the server to write to the socket using the routine `print()` and `println()` methods for output. The server process sends the date to the client, calling the method `println()`. Once it has written the date to the socket, the server closes the socket to the client and resumes listening for more requests.

A client communicates with the server by creating a socket and connecting to the port on which the server is listening. Add a new class “`DateClient.java`” in your project and add the following code.

```

import java.net.*;
import java.io.*;
public class DateClient {
    public static void main(String[] args) {
        try {
            //make connection to server socket
            Socket sock = new Socket("127.0.0.1",6013);
            System.out.println("Connected");
            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));
            // read the date from the socket
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);
            // close the socket connection
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

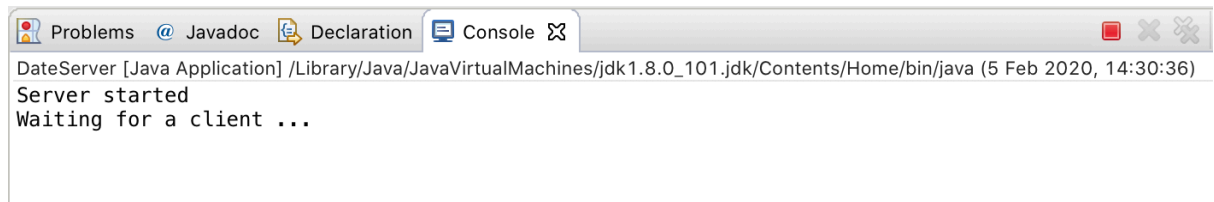
```

The client creates a `Socket` and requests a connection with the server at IP address 127.0.0.1 on port 6013. Once the connection is made, the client can read from the socket using normal stream I/O statements. After it has received the date from the server, the client closes the socket and exits. The IP address 127.0.0.1 is a special IP address known as the loopback. When a computer refers to IP address 127.0.0.1, it is referring to itself. This mechanism allows a client and server on the same host to communicate using the TCP/IP protocol. The IP address 127.0.0.1 could be replaced with the IP address of another host running the date

server. In addition to an IP address, an actual host name, such as `www.surrey.ac.uk`, can be used as well.

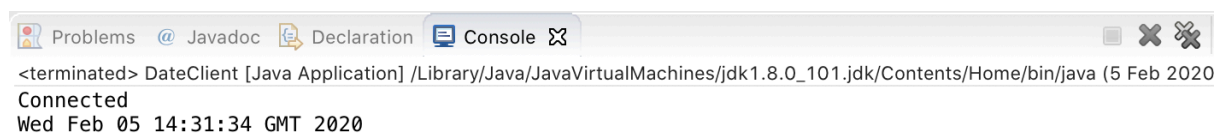
After writing the code for both client and server end, you can execute the server-side program first. After that, you need to run client-side program and send the request. As soon as the request is sent from the client end, server will respond back. Below snapshot represents the same.

19. When you run the server side script, it will start and wait for the client to get started.



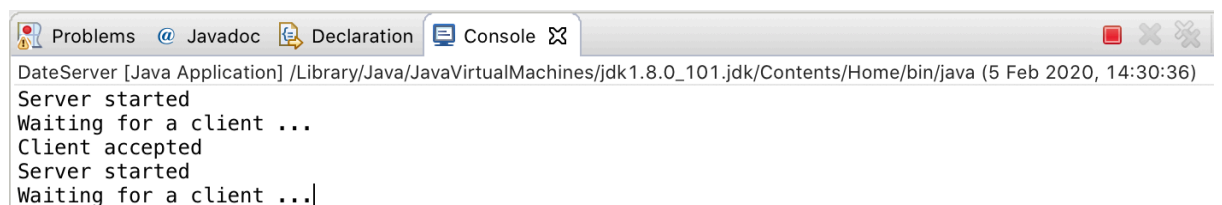
```
Problems @ Javadoc Declaration Console
DateServer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (5 Feb 2020, 14:30:36)
Server started
Waiting for a client ...
```

20. Next, the client will get connected and inputs the request in the form of a string.



```
Problems @ Javadoc Declaration Console
<terminated> DateClient [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (5 Feb 2020
Connected
Wed Feb 05 14:31:34 GMT 2020
```

21. When the client sends the request, the server will respond back.



```
Problems @ Javadoc Declaration Console
DateServer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java (5 Feb 2020, 14:30:36)
Server started
Waiting for a client ...
Client accepted
Server started
Waiting for a client ...|
```

Message Passing for Processes:

This section will provide approaches to the producer-consumer problem using shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. Some operating systems allow processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Although Java does not provide support for shared memory, we can design a solution to the producer-consumer problem in Java that emulates shared memory by allowing the producer and consumer processes to share an instance of the `MessageQueue` class.

You can implement a `Channel.java` interface with send and receive methods as follows:

```
// Send a message to the channel
public void send(E item);
// Receive a message from the channel
public E receive();
```

In `MessageQueue` class, implement the channel interface using a buffer that can be implemented using the `java.util.Vector` class, meaning that it will be a buffer of unbounded capacity. Since these are normal method calls, we can consider both the `send()` and `receive()` methods are nonblocking.

When the Producer generates an item, it places that item in the mailbox via the `send()` method. The code for the Producer is implemented in `Producer.java` class that you can download from SurreyLearn.

The Consumer obtains an item from the mailbox using the `receive()` method. Because `receive()` is nonblocking, the consumer must evaluate the value of the Object returned from `receive()`. If it is null, the mailbox is empty. The code for the Consumer is implemented in the `Test.java` class that you can download from SurreyLearn.

Exercise:

- Do a socket-based chat program that takes the address and port numbers as arguments, and you can start as many processes of them as you need to chat as pairs or groups.
- Hint: Download the skeleton code and follow the TODO comments to write the required code.