Part 1

Task 1

`#!/usr/bin/bash` - this tells the system that this is a shell script. `input_file="$1"` - this creates a variable - input_file and set it to the argument passed to the script. `[ ! -e "$input_file" ]` - returns true/false if the file set as input_file exists or not.

```
if [ ! -e "$input_file" ]; then
     echo "File not found: $input_file"
     exit 1
 fi
```

- this is an if then else, exiting if the file does not exist, continuing if it does.

`while IFS=',' ` - is Internal Field Seperator - basically here telling the script to use `','` to separate columns in line. `read -r id timestamp temp pressure;` uses IFS seperator to read a line and split it into these arguments. echo - dumps out the string inside "", and uses the argument values `"@argument"` in each print.

the file we caught earlier, done `< "$input_file"` injects it into the while loop so the read can execute on it.

Task 2

Very similar to Task 1, but the difference here is this: `tail -n +2 "$input_file" |` - does 2 things, tail takes off the header line of the file and starts the processing at the second line. then the `"|"` feeds this 'new' file to the while loop - so different from the done < earlier. in our loop this time, we intercept `@timestamp` before printing, and reformat it, creating a new variable - `formatted_time` - and setting it to the output of the date command - which makes the unix time more legible.

Task 3

Builds on Task 1 & 2 - but this time we have a second file - the output file we want to store the results of this script. `-z` checks if the user correctly specifies an output file, and `-e` then checks it doesn't exist - no overwriting. this while loop is similar in pattern, but this time we do some checking on the temp column.

We have a double pipe here, the tail from task 1 strips the header and passes the file without header to the awk. the awk processes this new data, and inspects col 3 - `$3` and filters out if the value is greater than 50 then we enter the while loop again as described earlier.

Task 4

this builds on task 3. but this time I inject the sort function, so the tail does as before - strips header. Then however the sort kicks into action. It needs a delimiter to identify columnn done via `-t ','` Once it knows how to identify columns we need to tell it what column to sort on - this is where `-k2,2n` comes in it says col `(-k2)` but only col 2 `(2n)` then once sorted it pipes it onto the awk which performs as described in task 3

Part 2

Instead of writing and reading to a pipe we could write and read to a file?

Yes - you could create a file, have the producer write to it and a consumer read from it. There would be challenges, described in answers below - but to this very specific question, the answer is yes.

What would be the difference? Why would be a pipe a better choice compared to a file? A pipe is better because it provides real-time IPC. Files don't. If you chose a file as a mechanism, you can dump data to a file, its persisted and the client can pick it up days later. A reboot, the data is persisted. With a pipe the data in it won't persist through a reboot. But if you want something more real-time a pipe is better. A client that opens is immediately informed when a producer pushes data into the pipe.

A pipe is also better because as the name suggests its FIFO. A client using a file will never know when data is pushed into the file. Nor will it know if a producer modifies by appending (FIFO-like) to a file or modifies adhoc(randomly across the file locations). Basically - order is not guaranteed.

Even if one agreed to some FIFO behaviour using a file as a pseudo-IPC, the performance will be far slower. Files have to be written to disk. pipes are in-memory.

Part 3

"A process of priority 0 should only run for 1 quantum, a process of priority 1 should run 2 quanta, a process with priority 2 should run 3 quanta and so on"

So, any process with a priority of x will get a quanta of x+1. Quanta is the work budget allocated to a process based on its priority. And while the integer used to describe process' priorities - 1,2,3.... might imply higher to lower, if you look at the quanta allocation x+1, it is the opposite - higher priority numbers implies higher and higher allocated quanta, ergo ability to do more work. For the question specifically process with priority 0 get 1 quanta and process with priority 1 get 2 quanta - circa twice the amount of budget for work - circa implying some time for the scheduler to context switch the processes.

Notes on the assembly attempt

You will find compiling MIPS. However, I did not complete this assignment fully. This is how far I got:

- created some variables to capture the quanta - this because I was getting a little lost in the code on my first attempts.

- extended the PCB to catch the new priority field, and kept the original 5 process limit
- updated the main to pass in priority in $a1 to now give each process a priority
- the syscall gets all the way down to s100_alloc, where I pluck that priority off a registering process and jam it into the new pcp location i created for it.
- that compiles, but further hacking around to get the scheduler working - well lest say the wheels fell of and I called it a day.

To be honest, between Python, C, Bash, Java, Mongo, NeoJS, SQL languages this year, MIPs was a bridge too far for me. The context-switch was too much - excuse the pun! As I mentioned I got a good way into this, but everytime I touched the scheduler the code broke, so I cut my time losses. Mea culpa.

Part 4

Why does a GPU provide a significant performance gain compared to a multi-core CPU when multiplying large matrices?

Taking a concrete example:

- Latest NVIDIA RTX 4090 has 16000 cores
- Latest Intel i9-14900K has 24 cores and 32 threads.

Matrix operations offer the opportunity to highly parallelize operations - as from the matrix.c example So the larger the matrix, the larger the opportunity to parallelize via thread, ergo the more cores available the more can be done in parallel. So in the above example a crude comparison would be an nxn matrix >32 would result in threads having to share the CPU, thus endure the over head of context switching as each may be interputed as they go through the CPU pipeline. That same n is 16K for the Nvidia example given. So for a 16k x 16k matrix, each thread gets their own core on the NVIDIA, while each thread has to share with 499 other threads on one of the 32 Intel cores.

Part 5

The counter is a shared memory location, accessible by all threads within the process. Unprotected by a mutex, multiple threads can update this same time - basically a race condition. The code itself is a read/increment/write pattern, so if two threads have a race condition, both will read the initial value of the counter, both will increment and both write the result n+1. However the issue is neither know about each other and ideally, the counter should have been updated before the second accessing thread read it - so the final result in this scenario is 1 less what it should be. Basically without searilizing the access one loses an increment.

What the mutex allows is the ability to ensure that only 1 thread can access the counter and modify it at a time - basically, while we have paralallelizm via threads, the mutex effectively seralizes the access to the shared counter - thus ensuring the lost increment is avoided.