Q1.

Flow of Code: All programs initialise 5 forks and 2 bowls accordingly. They initialise 5 threads simulating 5 philosophers.

Each thread has the following function. The philosopher begins by thinking, then goes on to pick up the left fork, then the right fork then starts eating. It picks up whichever fork it can find but can't start eating before picking up the other fork. When picking up the fork, it checks its availability.

A1. Strict ordering, in this program, the odd ones first reach for the right fork, and the even ones try to reach for the left fork. No semaphores have been used in this program; if the global fork is occupied, the thread is made to sleep. The rest of the code works similarly.

A2. Use of Semaphores. Global forks have been declared as semaphores and are technically assigned to philosophers. In this code, the philosophers simply check the availability of the forks through their semaphores. The rest of the code is similar. The semaphores are initialized before the launching of the threads and destroyed after joining.

B. Introduction of bowls. Similar to forks, global bowls are declared as semaphores and assigned to philosophers. The only change is before eating; the philosopher waits for a bowl.

In all cases, after the philosopher is done eating, he puts down all his utensils and goes back to thinking before getting another turn. (The Code is in an infinite loop to simulate the racing conditions of the philosophers trying to get their forks and bowls).

Q2.

All of these codes start by initialising the main array and the buffer array. A random generator function fills up the main array with the strings. Then a buffer array is created with an extra space for storing the index of the function. Then the IPCs are initialized and a for loop is run 10 times for sending 5 strings at a time. The IPC firsts sends the strings, waits for the largest index sent and repeats the process until all strings are sent.

FIFO: A self defined function loads the buffer with the 5 strings, the FIFO is initialized as write only, and then the strings are sent and the FIFO is closed. On the other side in the second program, a buffer is already created, the fifo is initialized as read only and it reads the strings, it copies the message into the buffer, prints out the details of the buffer and is closed. It is then initialized as write only and it sends the max index back to the first program and is closed. The FIFO is initialized in the first program read only and reads the max index which is then used to send the next 5 strings.

SOCKET: Similar methods followed as FIFO, Socket is first initialized, binded, connected, then first it writes the strings, waits for the max index repeats the process. The other programs connects with the socket, reads the strings, prints it out, then sends back the max index.

SHARED MEMORY: This program is also similar but it first writes the strings on to the "shm", then sleeps, the other program in this while reads the transfer, prints its details, sends back the max index and then sleeps. In this while the first program now reads the max index and repeats the process.

In all prgrams, time is calculated by initialising a variable before the starting of the loop and after it's end. There is a 3rd program which links both programs together as parent and child so split terminals need not be used.

Q3. The kernel module can be built in any directory at any location.
The kernel module is written as a C file with including necessary headers like <linux/module.h> and <linux/kernel.h>. To perform operations on the task structure the header file <linux/syscalls.h> is included. Inside the module there are two macros, __init which initialises the module and __exit which removes it. The operations required to be done by the module is written in the __init macro and those while de initialising are written in the __exit macro. They are called by the module_init and module_exit functions after definitions.
In particular case, *module_param()* is used to pass the process pid as a command line argument while initialisation. Inside the __init function, the module finds the task_struct of the given pid and then prints out it's details. The __exit function just contains an unloading message.
Since the module is stored in any directory, it is added to the /lib/modules directory by the line *make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules* in the makefile which adds this module as a target to the kernel makefile inside the modules directory by passing its working directory as a variable.
Once the module is compiled it is inserted with the process pid variable like so:
insmod ./module_name.ko pid="pid"
This loads the module and it can be confirmed by the command *lsmod* and its outputs can be seen on the kernel log by the command *dmesg*
To unload the module, write *rmmod module_name* and its results can also be verified by the *lsmod* and the *dmesg* commands.