QUESTION 1
All files relevant to Q1 are located in the folder 'OS Program'

- 'Development.txt' : A text file containing platform development notes
- Foucher_S_OS_Ass2_Program.out : A Unix executable of the program
- Foucher_S_OS_Ass2_Source_Code.c : The c source code used to compile the program
- pass.txt : A file containing the login credentials used by the program (The syntax used is described in 'Development.txt'). Must be in the same directory as the program's execution directory.

QUESTION 2

a) A single user and single process computer.

The only thing that can be done in this situation is to reduce the OS overhead by optimizing the process swapping in and out as the OS comes back to service requests. Since it is a single user single process, it will be the only one causing I/O bursts and launching system calls, so it doesn't a small quanta.
A good implementation might be to let the process run until it requests an I/O or an OS service. Perhaps we could set a quanta of 0.1 second and have the OS come back to make sure that the process is not in an infinite loop or ready to be terminated.
Another good implementation would be to design the process as a Kernel function. By doing this, we would eliminate any task switch overhead and could let the process run forever. Since it is part of the OS the OS/Process could service itself.

b) A single user but multi-process computer.

To Maximize CPU utilization, we should implement a priority based Multi-Level-Queue scheduling. A single First-Come, First-Served (FCFS) queue minimizes the average waiting time for a given set of processes, but might cause non important processes to take up more CPU time than they should. To remedy this, we could implement a Preemptive SJF, however, estimating the length of the next CPU burst is inaccurate, time consuming and might cause indefinite postponement for longer processes.

When using Multilevel Queue Scheduling we can assign priority to processes based on their scheduling needs, and give each queue its own scheduling algorithm. When servicing the queues, to avoid indefinite postponement, instead of having a fixed priority scheduling, we can implement fixed CPU time slices allocated to each queue.

C) A multi-user and multi-process computer.

On such a system, we can implement the same priority Queues as described in b), but also distinguish between different users. The system could allocate more priority to Kernel processes, Batch jobs, the less CPU time to user processes.

D) A multi-user, multi-process and multi-processor computer.

First we separate processes by length using the criteria from SJF processing. Each CPU gets assigned processes of similar job length until IO burst. Afterwards, the processes get separated in a Queue based on their priority. Once running, every CPU has a set of similar length processes arranged in a priority Queue. The CPU quanta allocation is then done dynamically, based on the average length of the processes being serviced: CPUs servicing longer jobs assign longer quanta and CPUs servicing shorter jobs have smaller quanta. This reduces the Overhead of task switching, since the dispatcher will only come in roughly when processes request an OS service.

QUESTION 3:

A) Is it possible for a process to block itself to wait for an event that will never occur? How?

Yes. In a deadlock situation; a process can block itself while waiting for a second process to release a resource. If the second process uses the same mechanism and block itself while waiting for a resource the first process has acquired, both processes will remained blocked indefinitely, unless a third process detects the condition and fixes it.

B) Can the operating system detect that a blocked process is waiting for an event that will never occur? How?

Yes. The OS can look at a blocked process, figure out why it got blocked and analyze weather it will ever have access to the resources it needs to unblock itself. (At least 0(n^2) of time consumption)
Another solution would be to record the list of blocked processes at every OS cycle and distinguish between ‘computer time’ processes and ‘human time’ processes. If a ‘computer time’ process has remained blocked for more than 1 second or a ‘human time’ process has remained blocked for more than 1 minute, assume that it is in a deadlock state and deal with it.

C) What reasonable safeguard might be built into an operating system to prevent processes from waiting indefinitely for an event?

The OS could force the release of all resources before a process can enter a ‘blocked’ state.

Q**UESTION 4**:

A) Assuming that a system allowed such a process to run, what would the consequences be:

I. Assume that SPAWN generated a new process

If spawn() generates a new process, every time it is called, the OS will have to find space in RAM and load static data from the HDD, build a stack and heap, a PCB and insert it into the ready list. RAM will fill up and eventually the system should crash. It should take a bit of time before this happens because the spawning task requires a lot of steps. Also, since new processes are created every time, once the system crashes, the RAM will contain fewer process threads which are bigger because they all contain the parent processes' static data.

II. Assume that SPAWN generated a new thread

In this case, the creation of new thread will be faster since it takes less steps than creating full new processes, but RAM will be filled up more slowly, because threads will be sharing the static data from the parent Process.
The system will eventually crash, but with more copies of the process in RAM.

B) What are the consequences of this basic theoretical result from computer science on your ability to prevent processes like the above from running?

Since this computer science theory states that it is generally impossible to predict the path of execution of a program, it is also impossible to fully safeguard a system against such processes unless taking arbitrary safeguard measures (as described in the next question)

C) Suppose you decide that it is inappropriate to reject certain processes, and that the best approach is to place certain run-time controls on them.

I. What controls might the operating system use to detect processes like the above at runtime?

When spawning, the OS could scan all the 'family tree' of the process (i.e. all other processes sharing the same static data or all the child threads of a parent process) and not allow the operation if a certain amount of copies have been made (see how big the process is and not allow spawning if more than 70% of RAM allocated to processes is taken)

II. Would the controls you propose hinder a process's ability to spawn new processes?

No, unless the process needs to spawn enough process to reach the condition where the OS will deny the operation. Realistically, most processes shouldn't need that many threads and could operate normally.

III. How would the implementation of the controls you propose affect the design of the system’s process handling mechanism?

This implementation wouldn't affect the process run-time handling mechanism (processes swapping in and out of CPU) so there wouldn't be any performance decrease other then during process creation. Even when creating a new process, the only noticeable addition regular process would suffer is something like ''if(newProcess !spawned)proceed;'. Processes using the spawn function would suffer an OS slow down of servicing their request of spawning by a factor of O(n) (All the OS does more is count the processes having the same parents)