Part 1:

a. The IOP sends an electric signal through a wire to the CPU, where it flips a bit in a specific place in memory that the CPU uses to acknowledge interrupts. At the end of every fetch/execute cycle the CPU checks if any of the bits in the location in memory have been flipped and if they have, it performs the following set of actions. First it saves its current context, so it can resume its previous process later and lets the IOP know that it has received the interrupt. It then identifies the interrupt by looking it up in vector table. This tells it where to jump the PC so that the relevant ISR can handle the interrupt. The ISR is a program stored in memory that can be run by the CPU to service the ISR's designated interrupt. The CPU with the ISR as the instructions preforms that actions that the interrupt was asking for. Finally, the CPU reloads the saved state from earlier and resumes the original process.
b. A system call is a request by a user program to the CPU, requesting an action that user programs cannot do under their current privileges. System calls are related to interrupts as they share a lot of characteristics such as interrupting the CPU's current task. A system call is executed similarly to an interrupt as system calls have their own spot in a vector table. This means that system calls share a lot of the infrastructure used for the interrupt system.
   Some examples are:
   Open: opening a given file.
   Write: writing data to a given location
   Exit: end a process
c. check if the printer is OK: when the printer receives this instruction it proceeds through a checklist. This includes checking the power supply, that a page is properly inserted, that the printer is not being actively used and that the printer has sufficient ink.
   print (LF, CR): printer receives instruction print(LF, CR). The driver looks up the code table for the commands LF and CR. It then has the motors shift the page so that the printing location is lowered by one line. Afterwards it has the motors shift the printing apparatus itself to the left margin. Finally, it notifies the IOP that it has finished its instruction.
d. Offline operation works by having the programmers prepare their programs separately from the device that would run it. They would do this by writing the program and its data on some form of data storage (first punch cards and later magnetic tape) and then submitting that to the operator of the computer system. The operator would then combine jobs with similar requirements (batching) and the computer with a program called the monitor would run them sequentially. The output of the programs would be then printed or stored and later returned to the programmers. An advantage of this system is that compared to the previous system of giving exclusive use of a computer to programmers one at a time, the computer could be used much more efficiently, which was a major consideration for the owners of the computing systems. A disadvantage is that the programmers would get no feedback on the performance or functionality of their programs until the results were returned to them. As the principal stakeholders of

early computing where the operators, the advantage significantly outweighed the disadvantage, and this system was quickly adopted.

e. Part 1: if the driver for the card reader does not parse the $ symbols, the data it passes to the CPU will be useless as it will not recognize the data as a job to be completed and will likely cause an error of some kind. A way to prevent would be to make the CPU recognize that it is being fed data lacking a critical piece (such as command markers) and warn the programmer.
   Part 2: If the operating system ran into a $END in the middle of a program, it would end the process prematurely. This can cause the program to return incomplete or incorrect data as the rest of the program is ignored. As this is a problem with the programmer and not the operating system, the operating system must simply assume that the $END was intentional and proceed as normal, ending the current job and looking for the next one.
f. A privileged instruction is an instruction that can only be run by the CPU while in kernel mode, not user mode. These are reserved for critical functions that cannot be trusted to the user. A pair of examples are:
   1. Disabling Interrupts: If the CPU needs to complete a task without interruption (such as modifying code related to interrupts) it needs to be able to do this. This is privileged as disabling interrupts for a significant amount of time will effectively prevent the CPU from completing any task.
   2. Halt or sleep: this is useful for the CPU to save power and/or extend the CPU's life. This is privileged as halting the CPU without proper consideration can cause unexpected behavior, data loss or other undesirable outcomes.
   3. In/out: reading from I/O ports. This is needed by CPU for basic functionality. This is privileged as programs should only read from or output to devices that they explicitly need.
   4. Modifying the Interrupt Vector Table: The CPU needs access to it so that it can add or remove vectors depending on what I/O devices are connected. No user program would ever need for this, this just prevents malicious programs from effectively stealing control of the computer.
g. $LOAD TAPE1: When this command is executed, the loader will take all the data on the cards from the language command (in this case FORTRAN) and the $LOAD command and load it into main memory, storing the address of the beginning of the program.
   $RUN: When this command is executed the CPU will find the address stored by the load command and jump to PC to that address. It will then set the CPU to user mode and then begin executing the user program.
h. Timed I/O:

Read = 1.3 sec

Print = 1.8 sec

Total time (s)	1.3	1.8	3.6	4.0	5.8	
	read					
		CPU op				
			print			
				CPU op		
					Print	

Loop time: 5.8 seconds
Total time: 1647.2 seconds / 27 minutes 27.2 seconds

Polling:
Read = 1.0 sec
Print = 1.5 sec

Total time (s)	1.0	1.5	3.0	3.4	4.9	
	read					
		CPU op				
			print			
				CPU op		
					Print	

Loop time: 4.9 seconds
Total time: 1391.6 seconds / 23 minutes 27.2 seconds
Interrupt:
Read = 1.1 sec (including interrupt latency)
Print = 1.6 sec (including interrupt latency)

Total time (s)	1.1	1.6	3.2	3.6	5.2	
	read					
		CPU op				
			print			
				CPU op		
					Print	

Loop time: 5.2 seconds
Total time: 1476.8 seconds / 24 minutes 36.8 seconds
Interrupt (with buffers):
Read = 1.1 sec (including interrupt latency)
Print = 1.6 sec (including interrupt latency)
This means that the CPU do the op after the first print while the print action takes place

Total time (s)	1.1	1.6	3.2	4.8	
	read				
		CPU op			
			print		
			CPU op		
				Print	

Loop time: 4.8 seconds
Total time: 1363.2 seconds / 22 minutes 43.2 seconds

Part 2:

Trace file 0-5 demonstrate the default behaviour of the simulator.

Trace file 19 contains the default version of a trace file that will be executed with various modifiers.

It has a duration of 3102 ms

Trace file 6 behaves as if the CPU is 50% slower than default, meaning that all CPU bursts take 1.5 times the time to execute.

It has a duration of 3295 ms, 106% the default time, meaning that the duration of the CPU burst has a minor effect on the total time taken by the program.

Trace file 7 has a context switching time of 20 ms instead of the default 10ms

It has a duration of 3162 ms, 101% the default time, meaning that the duration of the context switch has a minor effect on the total time taken by the program.

Trace file 8 has a context switching time of 30 ms instead of the default 10ms

It has a duration of 3222ms, 103% the default time, meaning that the duration of the context switch has a minor effect on the total time taken by the program.

Trace file 9 has the ISRs have a flat 100 ms increase in time taken.

It has a duration of 3702, 119% the default time, meaning that the ISR has a noticeable effect on the total time taken by the program.

Trace file 10 has the ISRs have a flat 200 ms increase in time taken.

It has a duration of 4302, 138% the default time, meaning that the ISR has a noticeable effect on the total time taken by the program.

Trace file 11 has the kernel mode operation time increased to 2ms (from 1ms).

It has a duration of 3108, a negligible impact on the time taken.

Trace file 12 has several I/O operations per CPU burst. This is to simulate an operation that, for example, takes many inputs for a given CPU action.

Trace file 13 has the opposite, with several CPU bursts per I/O cycle. This is to simulate an operation that, for example, has low amount of data, but needs several complex math operations applied.

Trace files 14-20 show the behaviour of the simulator in various situations with default settings.

In conclusion, in most operations most of the time taken by programs of this nature are taken by the ISRs of the I/O devices. Therefore, a good way of speeding up these programs is by minimizing the I/O calls or by having the CPU perform useful actions during the ISR (by way of multithreading).