50.002 Computational Structures

Information Systems Technology and Design

Devices & Interrupts

Natalie Agus (Fall 2018)

1 Asynchronous Handling

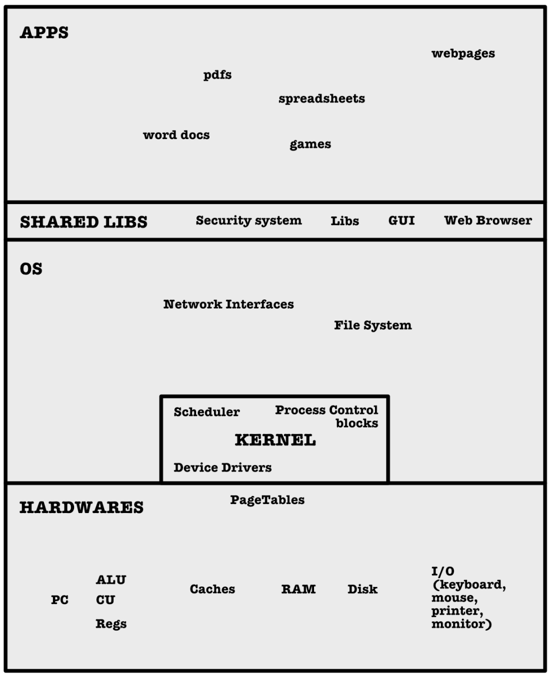


Figure 1

Our machine (computer) can only do 1 thing at a time: execute 1 instruction, or fetch one I/O, basically do only 1 program at a time. The Operating System (OS) allow "multitasking", by **rapidly switching contexts** between apps, thus giving each app the illusion of being in its own *virtual machine* with its own *virtual memory*.

Recall that each program "sees" itself as having its own memory (hence called virtual memory). Initially, every program was on disk (i.e: as binary executable). The OS can be seen as the first occupant of the empty RAM upon start up, loading in pagetable, and all other necessary component for the machine to run for use. The OS also know the directory of each file that lives in the machine's disk (secondary storage), so for example, it is able to show you files you can open in your desktop / home screen. How OS manages file system is beyond the scope of this course. It is sufficient at this point to know that if a program A is clicked or compiled or run, it will be actually loaded to the physical memory, which also in turn updates the pagetable. As the program progresses, its stack or data grows and it thinks that it starts to fill up its own virtual memory space. If program B is also opened, the OS also loads it to the physical memory and update the pagetable, however program A does not sees this. Program A and B has different **contexts**, each having its own virtual memory, thinking that it occupies the entire address space.

1.1 The Kernel

The kernel (the main part of the OS) is a set of instructions that live in the main memory, and it manages the execution of all apps in the computer, as well as the hardware (including I/O). The Kernel also has its own buffer (a dedicated section in the main memory) to temporarily contain I/O inputs and outputs. Values are temporarily held there until the OS fetches it and give it to the app that requires it.
In short:

1. Kernel serves as an intermediary between hardware and program
2. It deals with hardware and hardware inputs from user fills the kernel buffer (e.g: when keyboard button is pressed or mouse is clicked)
3. User program calls the Kernel through appropriate SVC(XXXXX) to request for input

2 The SVC

The SVC (supervisor call instruction) is a 32-bit instruction that allow the user (the apps) to communicate with the kernel (hence interrupting its own operation and give control to the kernel). You will want to execute the SVC if you need to temporarily transfer control to the kernel to satisfy certain type of request. One example is to check for I/O (like for e.g: keyboard input). From Figure 1, the Apps do not have direct access to the hardwares. The hardwares are managed by the kernel as part of the OS.
The SVC instruction:

```
SVC(XXXXX) in assembly language, translated into:

000001 --------------------- XXXXX in 32-bit machine language
(OPCODE)                     (SVC Index: the trap to go to)
```

The SVC opcode **is a form of interrupt request**, which trigger the ILLOP **handler**, and the SVC index tells the ILLOP handler to **trap** this exception into a trap (like a ’catch’ code) code with index as indicated by XXXXX in the SVC instruction above.

2.1 Example: SVC for keyboard input

- The app wants to check if there’s any keyboard input, e.g: raw_input() in Python
- This means that the kernel needs to tell the app if there’s any input
- The code raw_input() in the Python code is translated into SVC(XXXXX) in assembly language, having the 32-bit machine instruction as shown in the previous section. The value of XXXXX depending on the OS.
- The illegal OPCODE in SVC(XXXXX) traps to OS, now the OS controls the machine,
- The OS write the key input **from kernel buffer** in R0
- Return to the app and resume the app instructions
- The app can get the character from R0

I/O handling idea:

- Anytime the keyboard is pressed, it invokes the I/O interrupt handler
- The I/O interrupt handler read the input key ASCII character and put it in the kernel buffer
- SVC raw_input() fetches the item from the buffer and put it in R0

3 The Sleep Status

The ”sleep” status in the proctable in kernel memory (shown in blue in Figure 2 above) is useful to allow a particular program A to ”sleep” when it is waiting for a certain input from user, and hence allow the kernel to resume the operation of other programs instead until the required user input for program A arrives.
Consider the scenario where program A requests for user input, i.e: keyboard strokes:

1. **A bad approach:** wait and loop in kernel when buffer is still empty. This is bad because you **cannot interrupt a kernel**. If buffer never gets filled then the computer ”hangs”.
2. **A better approach:** if buffer is empty, returns to the user mode (the app) again and ask the user to request raw_input() again. However it is still a waste of time go to back and forth between app and kernel mode only to wait for the user to press the key.

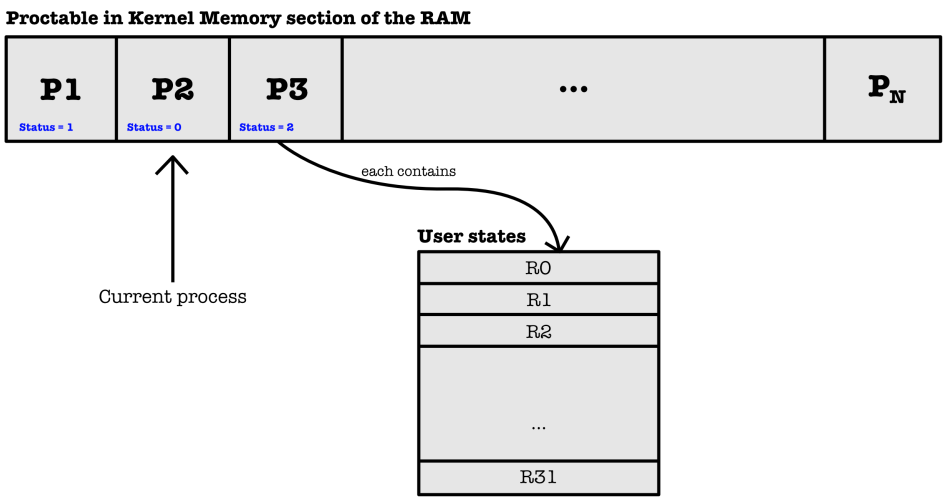


Figure 2

3. **A good approach**: OS performs scheduling by executing another program B first, and then returning to this particular program A with the *hope* that by then, the user has already pressed the key and this program A can continue.

4. **The best approach**: If buffer is empty, the kernel exits program A and execute another program B, and *will not return to A UNTIL there's a key pressed by the user*. **This is possible if one adds the sleep status in the proctable**:

   - The kernel decides that the process has to sleep if the keyboard buffer is empty using the command : **sleep(x)**
   - **x** is an integer and it is the program wake-up **trigger**
   - Sleep(x) transforms the status of the corresponding process into x
   - Status of 0 means that the program is **active**
   - The Kernel scheduler only rotate the execution among active programs, i.e: those with status of 0
   - When the buffer is finally filled, the IRQ code for I/O contains **wakeup(**x**)**, which will change the statuses of the processes with the same status x back into 0
   - This changes the processes that were "sleeping" into active mode, and will be executed by the scheduler

4 Scheduler and Real-Time I/O Handling

Figure 3 above illustrates the timeline of when a particular **interrupt request (could be system interrupt request, device and I/O interrupt request** is first invoked, then executed until it is done. An I/O interrupt request happens whenever any I/O

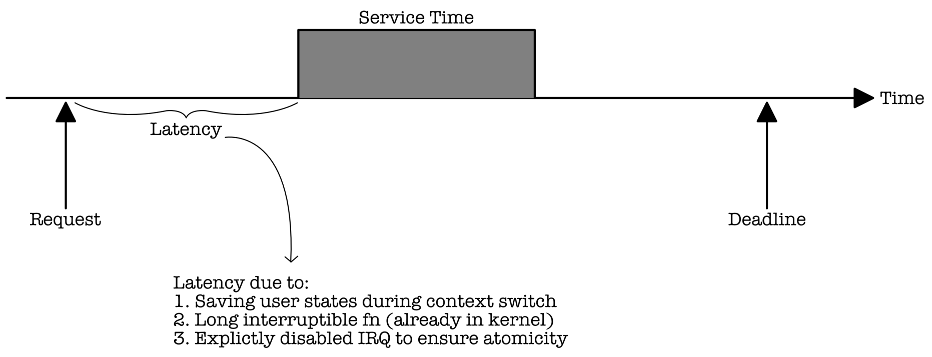

Service Time
Time
Latency
Request
Deadline
Latency due to:
1. Saving user states during context switch
2. Long interruptible fn (already in kernel)
3. Explictly disabled IRQ to ensure atomicity

Figure 3

devices is pressed by the user.

Typically, there is latency between the time when an I/O interrupt is first invoked, and when the machine actually does the work (service time) pertaining to that interrupt. A scheduler in the kernel has to ensure that a particular task is done **before deadline is reached**. And finally, the **latency** affects how ”real-time” the machine is for the user that handles the I/O. The shorter the latency, the more ”real-time” the machine seems.

4.1 Scheduling multiple interrupts

Our computer system is connected to multiple hardwares and I/O devices (disk, keyboard, mouse, printer, monitor, etc), some of which directly interacts with the user. Whenever an interrupt request is invoked, the scheduler has to decide how to handle these interrupt requests.

There are two methods to handle I/O interrupts:

1. **Weak, non-preemptive measure**:

    - The machine has a fixed ordering of device handling, for example: printer < keyboard < mouse, meaning that mouse has the highest priority and printer has the least priority.
    - Whenever an interrupt request for mouse arrives, its going to be the **next task to do**,
    - however the kernel **does not stop whatever task it was doing**.
    - This is called a non-preemptive measure: the kernel does not stop whatever it was doing, it allow reordering of the interrupt queue based a fixed ordering of device handling.

2. **Strong, preemptive measure**:

    - To have **priority level** for each device (use higher *p* bits of PC):

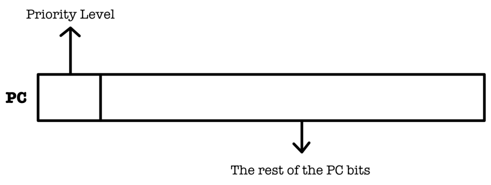


Figure 4

- The number of *p* bits required depends on how many priority levels you want the machine to have, e.g: 3 bits for 8 levels.
- Allow handlers with lower priority to be interrupted **only** by handlers with higher priority
- Take interrupt **iff priority level of requesting device** > **current PC's Priority Level**.
- If the requesting device has a higher priority, change the priority level of the PC with the device's priority level, and then PC goes to the device's IRQ handler.
- Same priority level can never interrupt each other

4.2 Recurring Interrupts in Strong, Preemptive Measure

If higher priority interrupts keep arriving at a high rate, the lower priorities keep getting interrupted,

- Interrupts for each priority level has a special block of handler
- Lower priority devices have to wait longer to complete its task, because its execution is spread out over time when not interrupted
- **Interrupting devices with the same priority level is NOT allowed**
- Same priority level can never interrupt each other

5 Calculations: determining latency

5.1 Case 1: Without scheduling

Consider a scenario where there's 3 devices that can invoke interrupt : Disk, printer, and keyboard, and that they can request interrupt in any order. Without any scheduling measure, **the worst case latency seen by each device is the total latency of the other devices**, as the interrupt requests can arrive in any order.

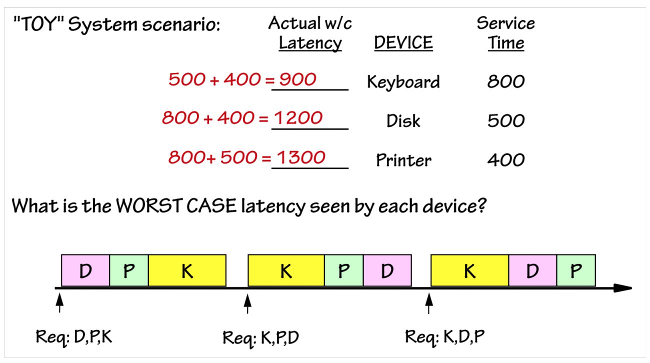


Figure 5

5.2 Case 2: Weak, non-preemptive measure

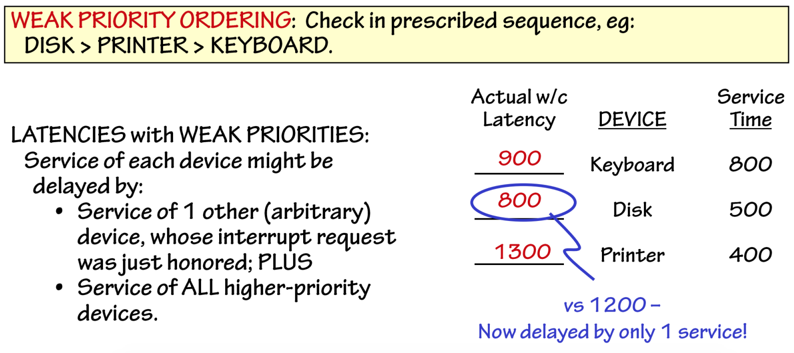

WEAK PRIORITY ORDERING: Check in prescribed sequence, eg: DISK > PRINTER > KEYBOARD.
LATENCIES with WEAK PRIORITIES: Service of each device might be delayed by:
• Service of 1 other (arbitrary) device, whose interrupt request was just honored; PLUS
• Service of ALL higher-priority devices.
Actual w/c Latency
DEVICE
Service Time
900
Keyboard
800
800
Disk
500
1300
Printer
400
vs 1200 –
Now delayed by only 1 service!

Figure 6

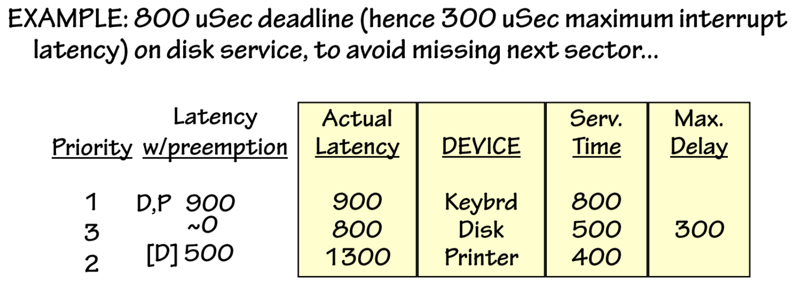
Assume now you have a priority ordering of: **Disk** > **Printer** > **Keyboard**, disk having the highest priority because you dont want to miss writing data in it.

- The **worst case** latency for disk is 800, which is the service time for the keyboard [1]
- The worst case latency for keyboard, with the lowest priority is the service time for disk + printer

5.3 Case 3: Strong, pre-emptive measure

The need for pre-emptive measure is apparent when there is a **deadline** imposed for the interrupt request: Looking at Figure 7 above, the disk, which has the highest priority, has a 800 uSec deadline, which means that the **maximum latency that it can have is 300 uSec**.

[1]recall in weak measure, you cannot interrupt whatever that is currently going on, but you can re-order of other interrupt requests



EXAMPLE: 800 uSec deadline (hence 300 uSec maximum interrupt latency) on disk service, to avoid missing next sector...

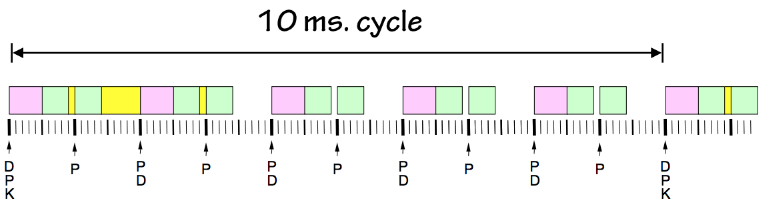
Priority	Latency w/preemption	Actual Latency	DEVICE	Serv. Time	Max. Delay
1	D,P 900	900	Keybrd	800	
3	~0	800	Disk	500	300
2	[D] 500	1300	Printer	400	

Figure 7

- The disk now has zero latency if pre-emptive measure is taken (when disk interrupt request arrives, the scheduler will stop other operations and attend to the disk)
- The keyboard with the lowest priority still has to wait for the disk and printer to complete if their interrupt comes, so its total latency is still 500 + 400 = 900.

5.4 Case 4: Recurring interrupts with strong, pre-emptive measure

How much CPU time is consumed by interrupt service?



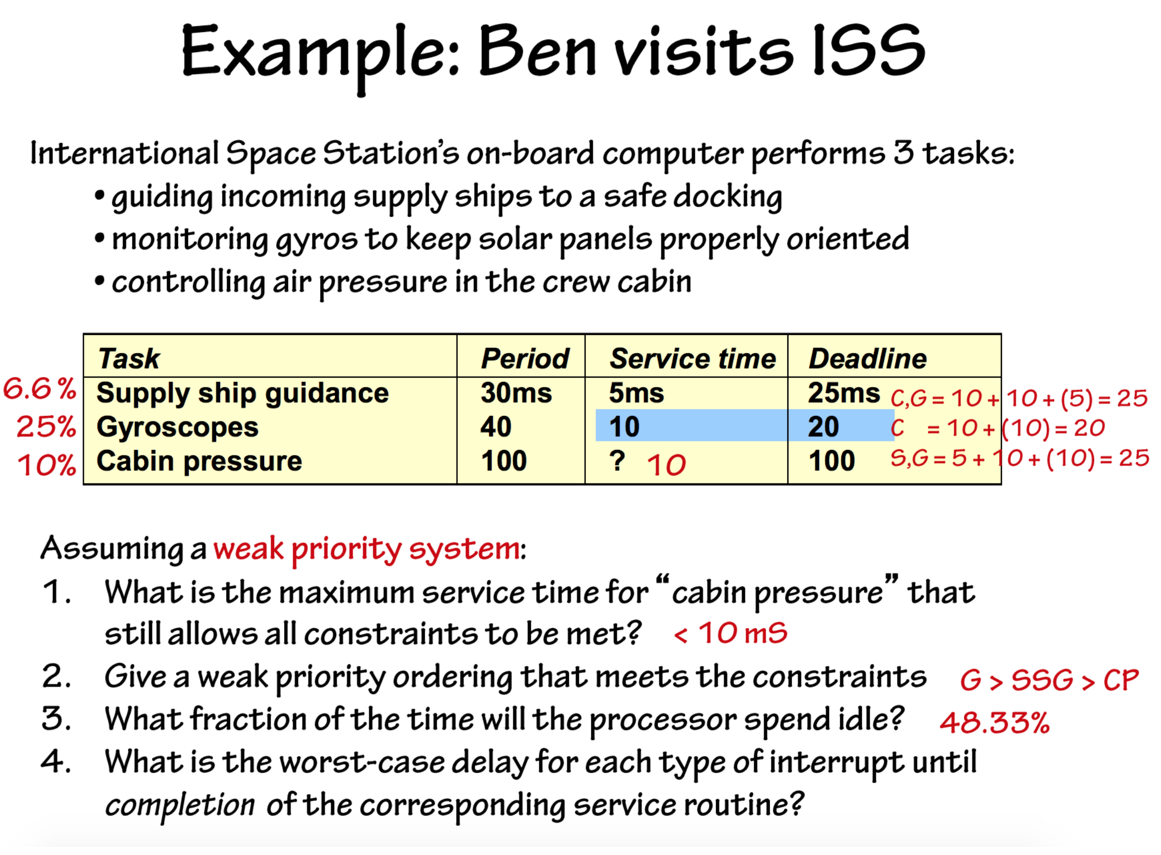
Actual Latency	DEVICE	P	Serv. Time	Max. Delay	Max. Freq	% Load
900	Keybrd	3	800		100/s	800u*100/s = 8%
0	Disk	5	500	300	500/s	500u*500/s = 25%
500	Printer	4	400		1000/	400u*1000/s = 40%

Figure 8

Now consider that the interrupt requests from these devices are recurring with certain frequency as shown in the Figure above.

- The keyboard service time is spread out (yellow region) due to interrupts from printer and disks
- The **CPU Load** is computed by **multiplying the max frequency of each device** with the **service time**
- The total load % in Figure 8 above is 73%
- The remaining fraction, 23 % is left over for the applications to use to do ”work”
- Your computer will not be able to execute any other application instruction if the CPU load is ≥100 %

6 Extra Example Question



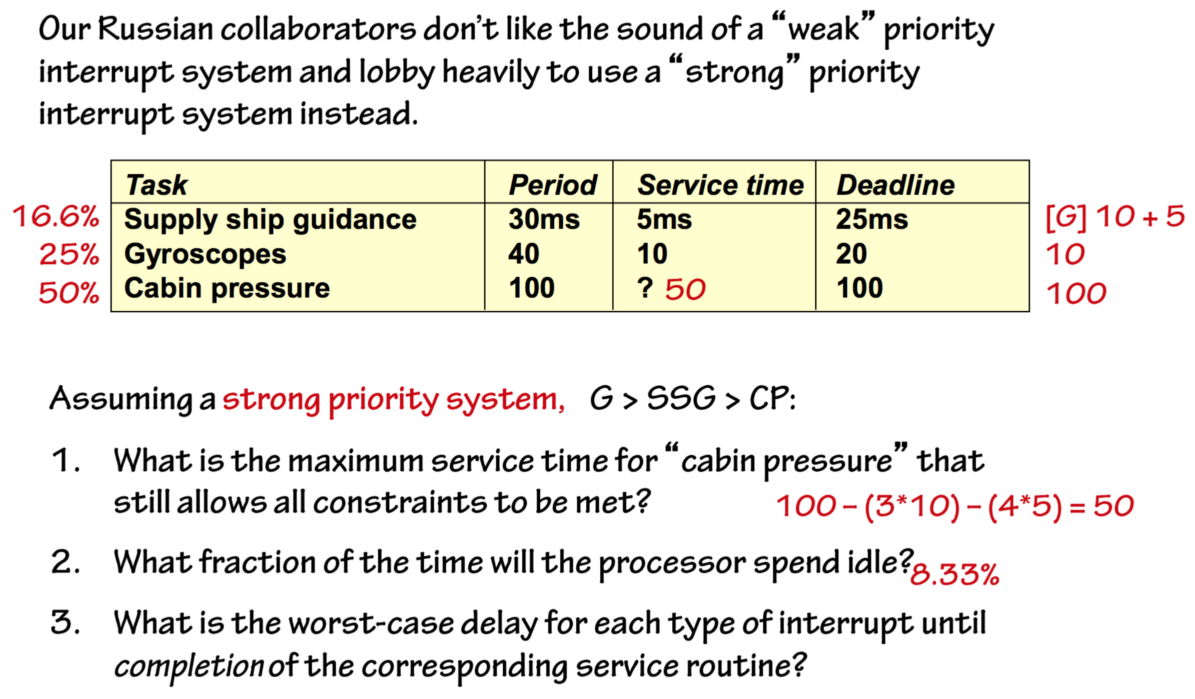
Example: Ben visits ISS

International Space Station's on-board computer performs 3 tasks:
- *guiding incoming supply ships to a safe docking*
- *monitoring gyros to keep solar panels properly oriented*
- *controlling air pressure in the crew cabin*

	Task	*Period*	*Service time*	*Deadline*	
16.6 %	Supply ship guidance	30ms	5ms	25ms	C,G = 10 + 10 + (5) = 25
25%	Gyroscopes	40	10	20	C = 10 + (10) = 20
10%	Cabin pressure	100	? 10	100	S,G = 5 + 10 + (10) = 25

Assuming a weak priority system:
1. What is the maximum service time for "cabin pressure" that still allows all constraints to be met? < 10 mS
2. Give a weak priority ordering that meets the constraints G > SSG > CP
3. What fraction of the time will the processor spend idle? 48.33%
4. What is the worst-case delay for each type of interrupt until *completion* of the corresponding service routine?

Figure 9



Our Russian collaborators don't like the sound of a "weak" priority interrupt system and lobby heavily to use a "strong" priority interrupt system instead.

	Task	Period	Service time	Deadline	
16.6%	Supply ship guidance	30ms	5ms	25ms	[G] 10 + 5
25%	Gyroscopes	40	10	20	10
50%	Cabin pressure	100	? 50	100	100

Assuming a strong priority system, G > SSG > CP:

1. What is the maximum service time for "cabin pressure" that still allows all constraints to be met? 100 – (3*10) – (4*5) = 50
2. What fraction of the time will the processor spend idle? 8.33%
3. What is the worst-case delay for each type of interrupt until *completion* of the corresponding service routine?

Figure 10

Explanation on max service time for CP with strong priority system:

1. During 100 ms period of cabin pressure,
2. SSG can happen 4 times: at t=0, t=30, t=60, t=90
3. Gyroscopes 3 times: at t=0, t=40, and t=80
4. With 100 ms deadline, the max service time is spread out over that 100 ms from the first moment CP request is given,
5. So the max service time of CP is 100 - 3*10 (gyroscope) - 4*5 (SSG)