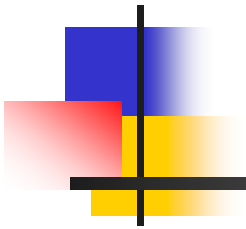
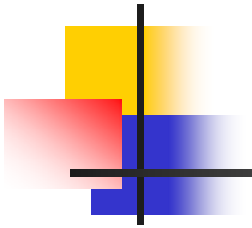


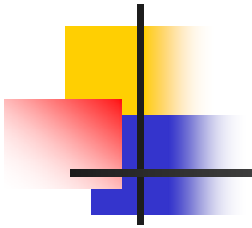
Operating System Scheduling



Operating System Scheduling

Introduction

- Let us start off by considering for a moment, a situation where we have computer with a single processor, running just one single process.
- Given this situation, the concept of scheduling between should never need to arise.
- Here, the process is given all the available CPU time and resources it can possibly use, even if this may not be very productive or efficient.
- For example, a burglar alarm could be designed around a single process running on a small microcontroller
- Its whole life would be dedicated to monitoring all the sensors connected to the system to see if they have anything to report and sounding an alarm in response to any of them being triggered.
- It would not matter if the software employed inefficient polling techniques to monitor the sensors, or whether it employed interrupt driven IO.
- Similarly, it would not matter if the CPU spent it's whole life monitoring these sensors and never detected a single break-in, that is, the system was 100% inefficient in the sense that it never succeeded in carrying out its primary task (detecting break-ins).
- The bottom line is that this single CPU/Single Task computer has nothing else to do – no background tasks or batch programs to run etc so it doesn't matter how inefficient the designer is in its implementation.



Operating System Scheduling

- In a multi-tasking system with an operating system however, it would be inappropriate to waste such time executing a process that achieved so little and in the process, hogs so much of the CPU time that other processes are blocked from running.
- Therefore, with the introduction of multiple processes, an operating system has to come up with a scheduling strategy that will effectively share out the CPU time, not necessarily always equally amongst the processes, but in a way that would give time only to those processes that could make effective use of it.
- In order to achieve this goal, it may be that the programmer and hardware designer will have consider scheduling issues when designing their processes and perhaps give the OS some clues about when to reschedule the process.
- This is the basis of these two lectures on Operating system scheduling.

Process Classification

- Before we can come up with an effective OS scheduling strategy, we need to classify the different kinds of processes that could potentially run on our system, since as we shall see later, each process type may dictate its own optimised scheduling approach which may conflict with that of other types of processes which may be optimised differently.
- In other words it may be necessary to come up with a compromised scheduling scheme in order for different process types to co-exist under one operating system.

Process Classification (cont…)

- Considered simply, we can classify processes on the basis of
  - Those that perform their task and then naturally stop. (e.g. Calculating a tax return)
  - Those that run forever and have no natural end. (e.g. Burglar alarm, video game)
- Regardless of the above classification, we could classify processes on the basis of how they perform their task w.r.t. to the outside world and their processing requirements. Briefly we can classify processes as being
  - Numerically Intensive or (Example: Compiler)
  - Interactive (Example: Word Processor)
- A Numerically Intensive process is one which consumes vast amounts of CPU time in order to carry out its task. Processes such as these would utilise all the CPU time that came their way and would be unwilling to voluntarily relinquish the CPU to another process as this would only serve to slow down its own progress. Furthermore, numerically intensive processes have very little interest in interacting with the outside world.
- An Interactive process is one which spends most of its time idle waiting for an operator or some device to supply it with the data it need to continue processing. Processes such as this are characterised by their very "bursty" CPU processing requirement. That is, most of the time that lay dormant and occasionally wake up, (perhaps in response to a key press in the case of a word processor) to briefly consume processor time before returning again to their idle state.

Scheduling Requirements for Numerically Intensive Processes

- From the point of view of a Numerically Intensive process, the single most important thing in its life is obtaining the processing time it needs to complete a task or at least make respectable progress towards it.
- If the OS is reckless with its scheduling strategy and hands out CPU time (i.e. processor resources) to programs that waste that resource (such as a word processor or burglar alarm that utilises a polling approach to sensor detection), then the net effect will be to slow down numerically intensive processes that have to compete with those processes.
- In summary if a system is composed only of Numerically Intensive processes, then the OS should aim to optimise the Utilisation of the System.

Scheduling Requirements for Interactive Processes

- For an Interactive process, such as a word processor, or a real-time application, the primary concern is one of response time.
- In a multi-tasking environment, fast response times, by necessity require rapid and frequent process swaps, so that the process designed to deal with the response is given the opportunity or 'window' in which to generate the response to it.

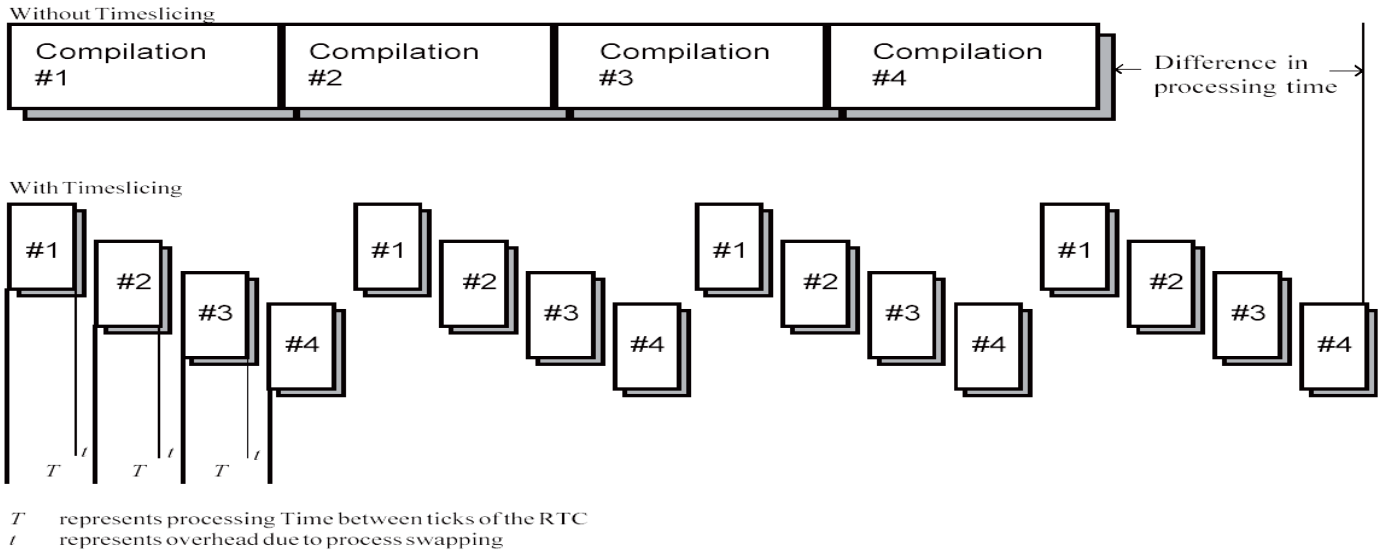
Conflict of Interest in Scheduling Numerically Intensive vs. Interactive Processes

- We shall see later that the Scheduling approach dictated by the needs of Numerically Intensive processes are in direct conflict with those required by Interactive processes and thus any operating system that could potentially find itself handling both types of process (the majority) must balance these conflicting needs. To understand where this conflict comes from, let us consider how we might optimise each kind of process.

Operating System Scheduling

Understanding the Scheduling of Numerically Intensive Processes

- Suppose as an example, we have a system running four Numerically Intensive processes that, in this case, perform a well defined task to completion and then cease.
- A good example of just such a process would be a C++ compiler which requires no interaction with the outside world leaving its operator to walk away and take a coffee break while their program compiles.
- The most efficient way to deal with this classification of process would be to suspend time-slicing altogether and deal with each compilation sequentially, one after the other until each completes. This would maximise the utilisation of the CPU since there would be virtually no overheads involved in swapping between processes (except at the completion and start of each one).
- This is illustrated graphically below, where 'T' represents the time between ticks (interrupts) from the RTC (typically 10mS) and 't' is a fixed operating system/CPU dependent overhead involved in swapping from one process to another, and varies from system to system.
- As you can see, running processes sequentially to completion means that the overall execution time for all 4 is less than would be the case if we time-sliced between them and tried to run all 4 concurrently. This is due to the accumulation of all the 't's brought about as a result of time-slicing.



Deriving a Figure for the Effective CPU Utilisation

- Effective CPU utilisation is defined as the ratio of the time spent doing useful work to the time spent on overheads and from the previous illustration, the utilisation for a time-sliced system is given by the equation

$$\text{Utilisation (\%)} = 100 * (T - t) / T$$

- Now provided T >> 't', (i.e. slow or non-existent process swapping) then utilisation approaches 100% which is the ideal optimum solution for scheduling numerically intensive processes.

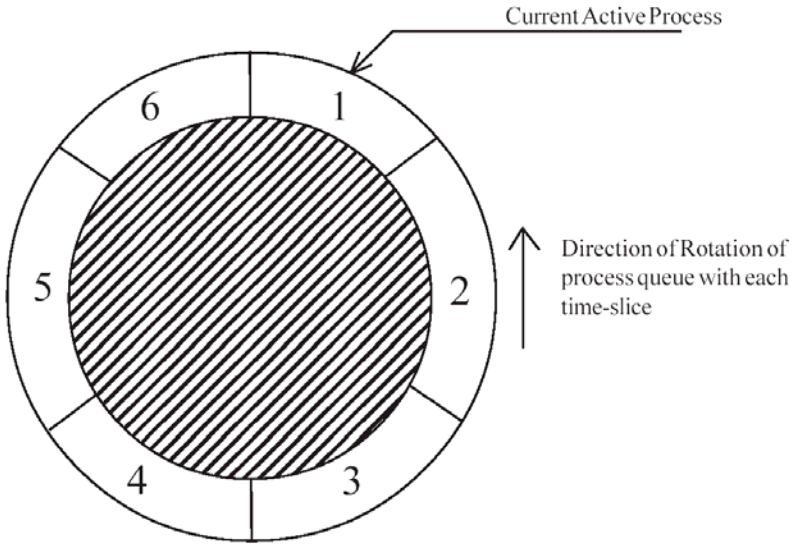
The Operators View of this approach

- Now even though disabling time-slicing for the duration of each of these processes would ultimately yield the highest CPU utilisation figure and result in all compilations being performed in the fastest, most efficient way, there are other factors to consider.
- How would the programmers sat at their terminal view this situation ? If one operator decided to perform a compilation and there were several already waiting in line for the opportunity to run, then our operator may well have to wait several minutes before for their compilation to even begin. The use may have to wait several minutes to find a compilation error on line 1!!!
- In other words, with little or no time-slicing, the system has become deeply unresponsive to its users/environment. This is unacceptable even in a simple data processing environment never mind a real-time system, where a response to an external event may make the difference between life and death. Such an approach is unusable if the processes had no natural end.
- With time-slicing enabled all compilations would be seen by their operators to progress simultaneously, albeit with each running at approximately a quarter of the speed (for four processes) they would run at with time-slicing disabled, but at least the system is more responsive to the operator and can be seen to be doing something.

Operating System Scheduling

Scheduling Highly Interactive Processes

- Of course suspending time-slicing becomes intolerable and unusable if the process is interactive where it spends much of it's time waiting for some user interaction.
- For example, what would happen if a programs operator disappeared for a coffee break when the program required some input data? The whole system would grind to a halt as far as other processes were concerned
- So what factors influence how responsive a system is and how can we make use of this knowledge to optimise our scheduler for running interactive processes?
- The illustration opposite shows how an operating system might schedule 6 processes.
- Let's assume that process 6 is an editor and the other 5 are say compilers.
- Let's also assume that process 1 is currently at the start of its time slice.
- This means that process 1 is executing and receiving CPU time (meaning processes 2-6 are suspended and not receiving CPU time).
- With each 'tick' from the RTC, the current executing process is 'frozen' and the queue is rotated (in this case anti-clockwise) so that process 2 can pick up where it left off last time it received CPU time.



Operating System Scheduling

Scheduling Highly Interactive Processes (cont…)

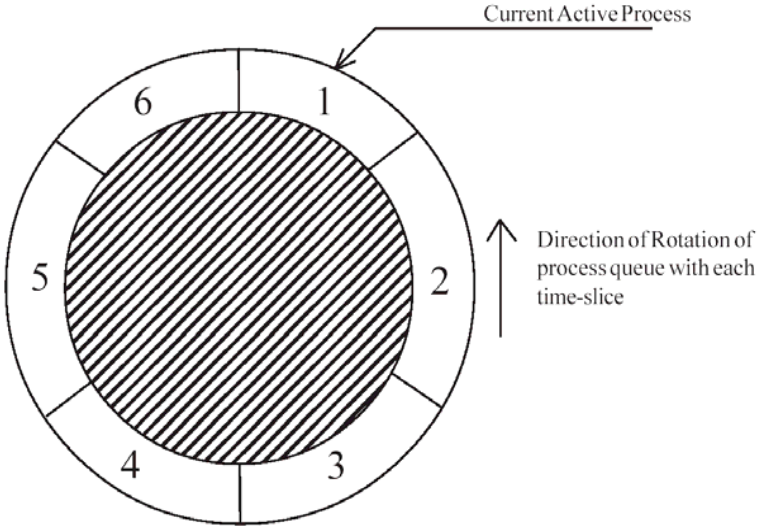
- Now imagine an operator sat in front of a terminal connected to process 6 presses a key for the editor program. How long does the system take to respond to it?.
- Before process 6 can deal with the operators key press, it must become the *active* process.
- For this to occur, 5 ticks of the RTC and 5 rotations of the queue must take place before process 6 finds itself at the head of the queue receiving CPU time.
- In general then, there is a worst case response time for process 6 in responding to the event given by the equation

  Worst Case Response time = T * (N - 1)

  where 'N' is the Number of processes in the queue awaiting CPU time, and T is the time between ticks of the RTC.
- Typically, the delay is not always worst case (in fact a key could be pressed while process 6 was actually active, leading to virtually no delay in the responsive time) and thus the '*average*' or typical delay in responding to the key press given by the equation.

  Average Response time = T * (N - 1) /2

- From this it is obvious that in order to improve response time to events for interactive processes, the system would be better off with a *small time for T*, that is, when the RTC is 'ticking' faster, leading to more frequent process swaps.



The Conflict between Interactive and Numerically Intensive Processes

- Reducing 'T' however has a detrimental effect on numerically intensive programs such as our 'C++' compiler which, in order to execute efficiently requires a *large T*.
- Remember that utilisation was given by the equation

Utilisation (%) = 100 * (T- t) / T

- Now given that 't' is a fixed overhead based on the clock speed of the CPU coupled with the size and complexity of the algorithm within the OS required to deal with the RTC interrupts and perform the process swap, this leaves 'T' as the only variable. Thus a small T, reduces utilisation (*all other things being equal*).

- However worst case response time is given by

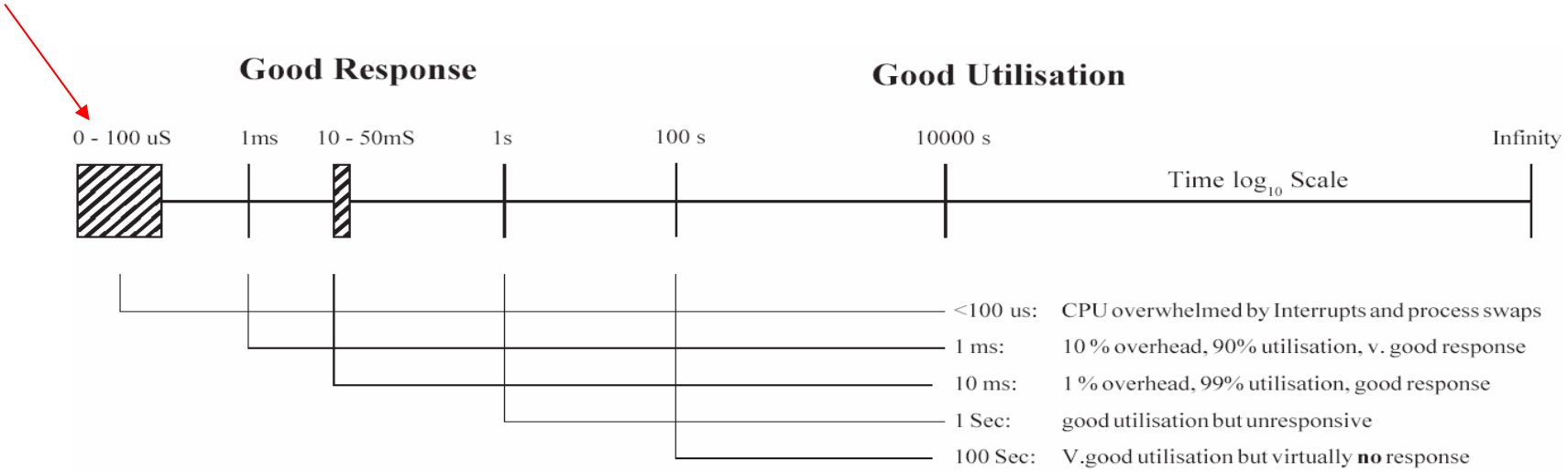
Worst Case Response time = T * (N - 1)

- In other words Numerically Intensive and Interactive processes have conflicting needs and this presents something of a dilemma for the systems designer who may have to design a system able to run both types of processes at the same time.
- How do they optimise the system and arrive at a value for 'T' which neither favours, nor disadvantages either kind of process ?. That is what value of T serves as a good compromise for both types of processes?

Operating System Scheduling

Balancing the needs of Numerically Intensive and Interactive Processes

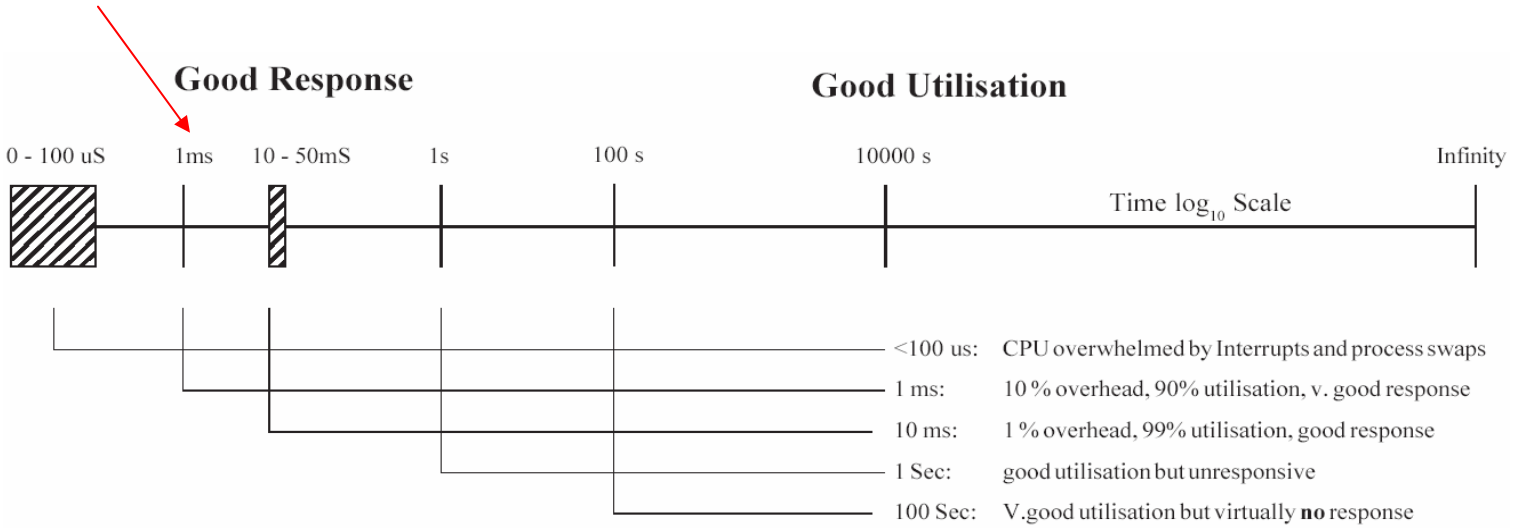
- Let us consider now how response time and CPU utilisation are both affected by adjustments of the time slice period and see if we can come up with a time for this where both processes can co-exist happily.
- Imagine that on the front panel of the computer system you are using, is located a control knob which can be used to alter the Time between ticks of the real-time clock (RTC) in the system from 0mS fully anti-clockwise (i.e. the RTC is producing continuous ticks), to infinity fully clockwise (i.e. there is no time-slicing taking place). What would a user of the system observe from the system?
- Let's start off with the knob turned fully anti-clockwise, so that interrupts from the RTC are being generated as fast as possible. If we assume for the moment that given a particular microprocessor and operating system combination, the overhead 't' involved in swapping from one process to another is 100us, then if the RTC were producing interrupts every 100uS or less (T <= 100 uS) , the CPU would be overwhelmed by interrupt requests. That is requests by the RTC to perform a process swap would happen so frequently that the OS/CPU would not even be able to perform the required process swap and resume the execution of the new process, before another request would have arrived from RTC.
- In effect the system would lock up, unable to deal with any of the desired processes, achieving nothing. The CPU utilisation figure would be zero (T – t < 0). In effect RTC interrupts of less than 't' lead to no productivity, since the Operating system is total overwhelmed in swapping processes.



Operating System Scheduling

T = 1mSec

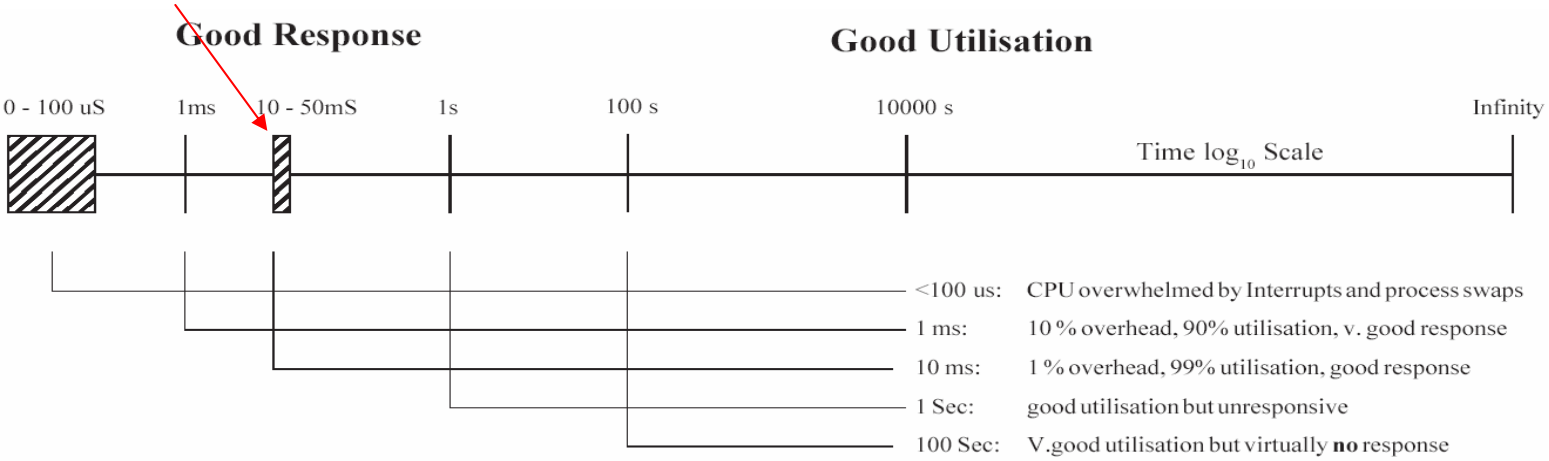
- Let us now turn the knob slightly clockwise, so that we slow down the rate of process swap request generated by the RTC (T = 1mS). Now the CPU has more time to deal with them when they do arrive and may be able to perform the process swap and allow the process some time to execute before the next swap request from the RTC.
- At this point, the CPU is still spending a disproportionate amount of time dealing with the interrupt itself and actually directing the CPU from executing one process to executing another. This obviously results in a very low level of CPU utilisation, but this does mean that processes are allowed to progress, even if such progress is slow.
- Obviously, the rate at which process swaps are occurring may still be very high (100us < RTC < 1mS), and in theory, the user would notice a very fast response time to any external activity, such as keyboard presses. In practice however, because the time slice period available to the process is still short, due to the overheads involved in the process swap itself, the process is still not very productive, though slowing down the RTC further still would improve this as we shall see.



Operating System Scheduling

T = 10mSec

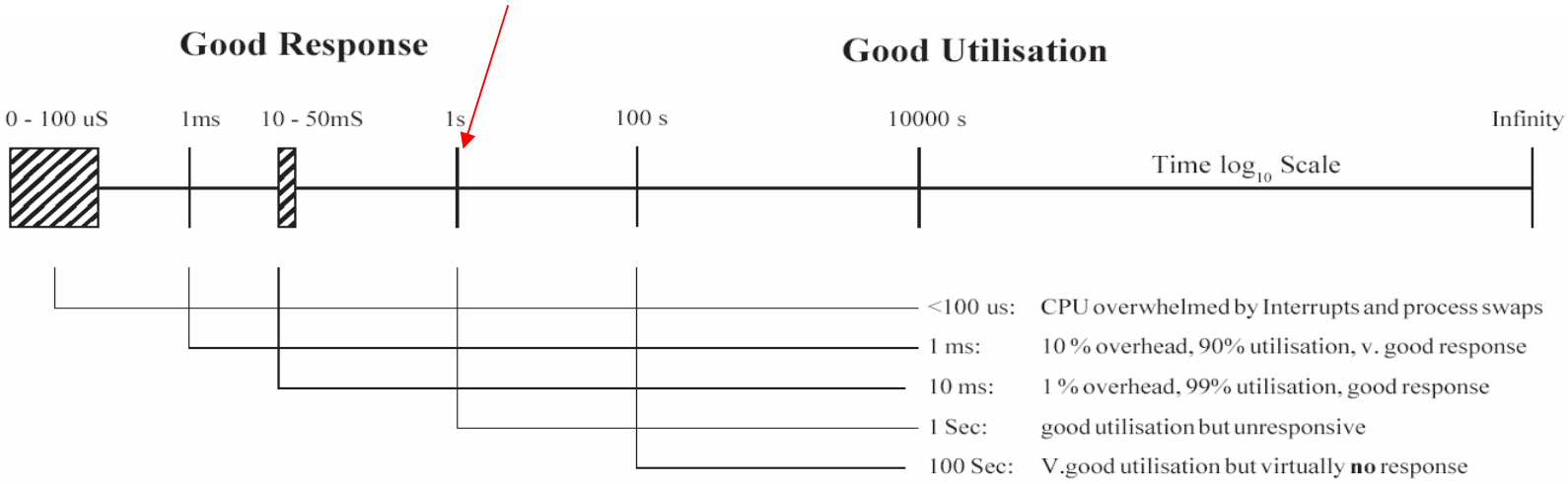
- As the knob is turned further clockwise, (T = 10 mS) time slice requests become less frequent from the RTC and a process is given proportionally more time to execute within each time-slice. CPU utilisation increases correspondingly as a greater proportion of the CPU time within a given time slice period is spent executing the process and less is spent performing the swap.
- The system is still responsive to its environment (100 swaps per second) and is able to adequately deal with each event within the allotted time slice. In effect an optimum point of compromise will be reached, whereby, the process will be given sufficient time during any one time slice period to perform a useful operation (i.e. we get 99% utilisation) and yet a process does not have to wait unduly for its next time slice (because we are swapping between processes at the rate of 100 per second)
- Time-slicing, at this point, should not be noticeable to the user, unless of course there are so many processes running in the system that the CPU is unable to service them all within an acceptable time period. For example if the time-slice period were 10ms and 100 processes were active, then the CPU might take 1 second to deal with them all before starting again. This might be noticed by the user depending upon the interactive nature of the process.



Operating System Scheduling

T > 1 Sec

- As the knob is turned further clockwise (T > 1 Sec), CPU utilisation is further increased, approaching the ideal (though never actually attained) figure of 100% but it's the law of diminishing returns once again, i.e. we sacrifice a lot of response time for very little perceptible gain in utilisation. At this point time slicing is approaching the state where it occurs so infrequently that it has all but ceased to operate.
- The user would notice that processes ran rapidly once started, but may take a long time to actually begin, since the process would have to wait much longer to obtain a time slice or window in which to execute.
- Response time to external activities though adequate for the process that is running at the time, becomes unacceptable for other processes and the time slicing action would become highly visible to the user, particularly in an interactive environment such as editing, whereby bursts of activity would be followed by long periods of inactivity. Ultimately or course as the knob is turned fully clockwise, time slicing is suspended. New processes would then have to wait for existing ones to complete before being allowed to execute).



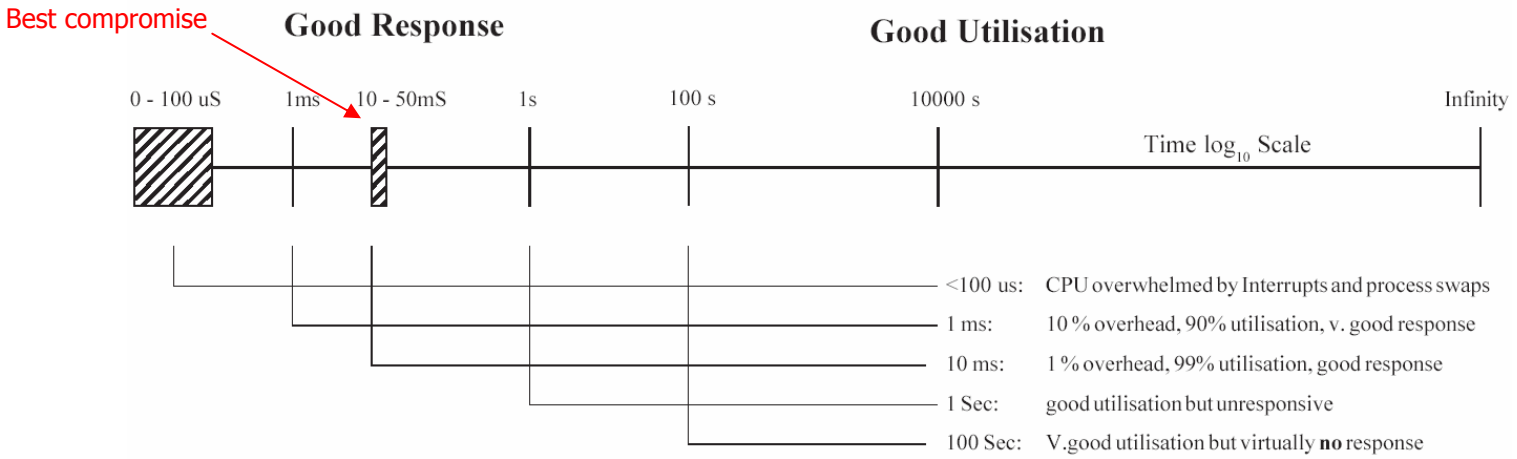
Operating System Scheduling

A Compromise Value for T

- From the diagram shown below, it is obvious that any time sliced computer system will have to be a compromise between achieving high CPU utilisation figure, and obtaining a good response time, and that both are a result of choosing a suitable RTC period.
- The optimum time-slice period will often depend on the processing capabilities of the CPU itself, a faster CPU is able to sustain the same CPU utilisation figure with more frequent process swaps (and thus be more responsive), than a slower CPU with a longer time-slice period.
- This is why many commercial operating systems running on a moderately fast CPU use a time slice period of between 10 and 50mS as the best compromise between utilisation and response time.

Effects of Improving CPU Speed

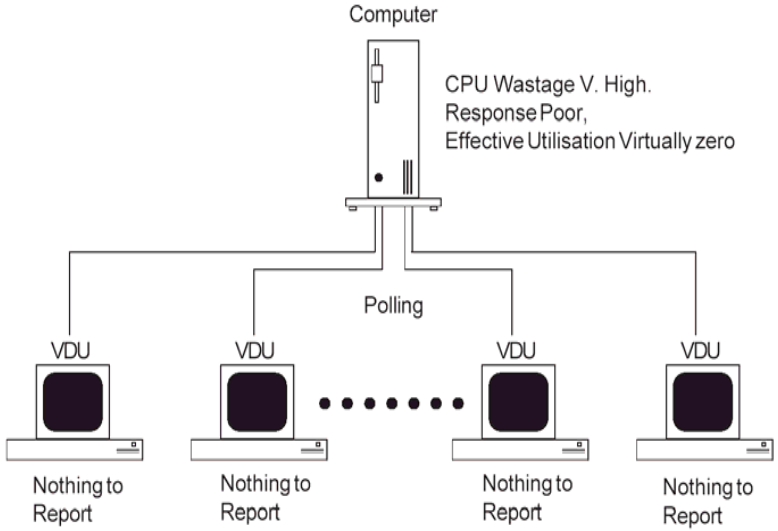
- If the speed of the CPU were to say double, then the execution time of the OS required to perform the process swap 't' would halve to say 50uS.
- We could thus double the RTC period 'T' to say 5mS and achieve the same % Utilisation while halving the worst case and average response times for a process
- This explains why with progressively faster CPU, new operating systems reduce the time 'T'. Windows for example uses 1mS RTC period. Ideally we would like an infinitely fast CPU !!!



Operating System Scheduling

Software Polling vs. Interrupt Driven I/O

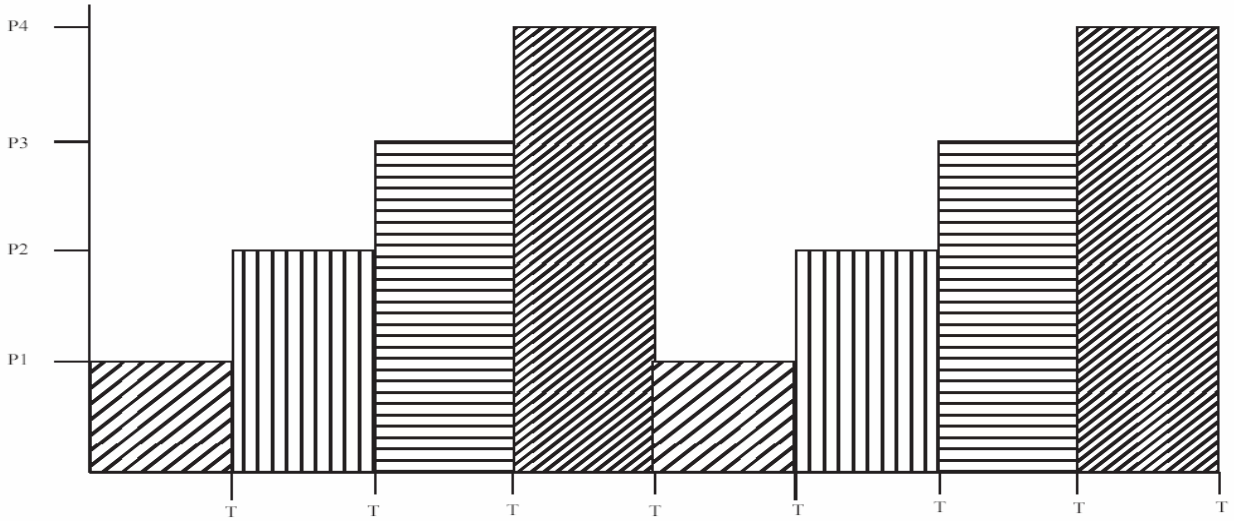
- In coming up with a compromise figure for the period of the RTC, we assumed in the previous discussion that all interactive processes were involved in polling their respective IO devices to determine if something important had happened and thus whether they needed to be serviced.
- Polling as we have seem with pipelines and mutex's is very wasteful of CPU time as this activity involves the process spending virtually all of its life testing for an event/condition that happens only infrequently.
- As an example of this, imagine a system with say 4 terminals/users connected to it (see illustration opposite).
- If each process connected to that terminal/user (such as an editor) were to adopt a polled approach, then each process, when it received its time slice would waste that whole 10mS period looking for a character to arrive from the users terminal.
- Given that operators struggle to sustain typing speeds of even 3 characters per second, then it would mean that 9 times out of 10, the process is not going to detect a key press within each time slice period, however the process still has to sit in a polling loop looking for that key press



Operating System Scheduling

- We can see that adopting a polled approach to Interaction means that even though very little actual work is taking place, 100% of the available CPU resources are committed to processes polling their IO, leaving very little time for any other tasks such as batch processes.
- A graph of CPU utilisation might look something like that shown below. As you can see, a polling process will fully utilise its 10mS time slice period, even though there is often little that it can achieve within that time period.
- Imagine then that one of those four processes was actually a C++ compiler, i.e. a numerically intensive process. The graph would look exactly the same, meaning that our compiler is now running at 25% of the speed it would run if it were the only process being scheduled.
- If there were 99 users editing and they went for a coffee break, our C++ compiler would only run at 1% of the speed it would run if it were the only process running in the system.
- In effect polling turns an interactive process into a numerically intensive one and the net effect is that the whole system become sluggish and unresponsive.

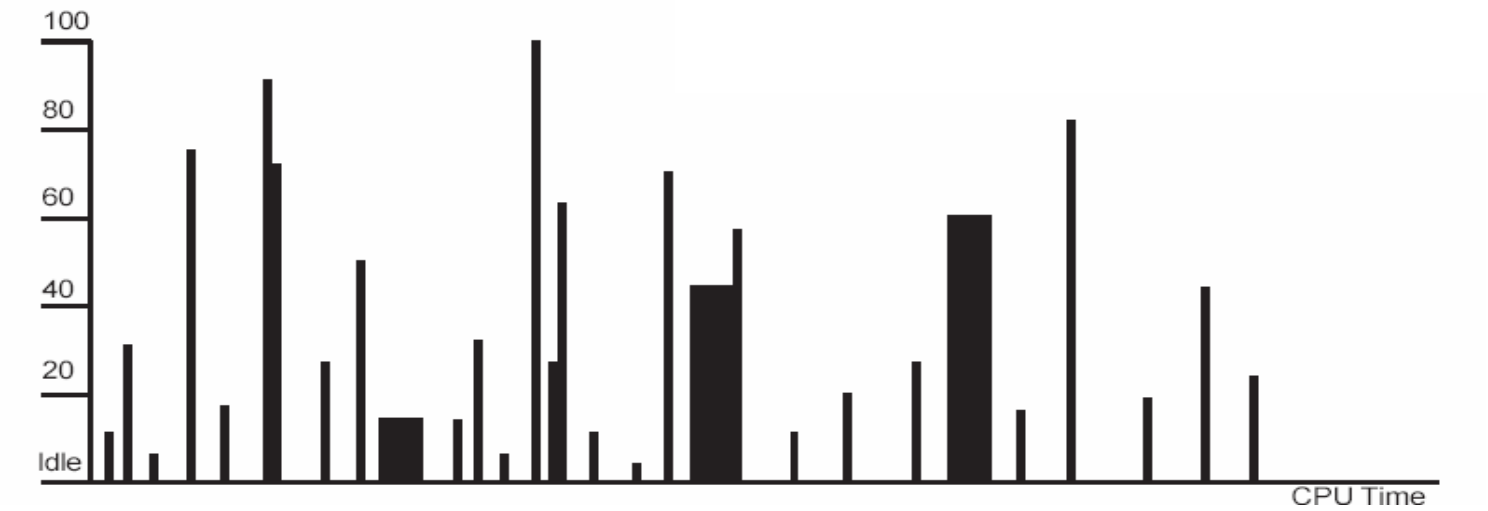
Effect of Scheduling Four Polled IO Processes



Operating System Scheduling

Using an Interrupt Driven Approach

- If we could design the hardware of our system to utilise an interrupt driven approach to IO, then each device could signal the operating system whenever it had something important to say and was requesting service.
- This would have a dramatic effect on the utilisation of our CPU since the fact that the IO device can now request service rather than having to be polled means that the operating system could now suspend a process whenever it attempted to read or write to a device that was not ready.
- This means that any process attempting to read from a keyboard with no data or write to a terminal/printer/network etc that was busy/off-line would not consume CPU time leaving the operating system the time and freedom to schedule the CPU to run those tasks that are not blocked by IO operations, such as a C++ compiler.
- The effect of introducing this interrupt driven scheme on our system with 99 editors would mean that the CPU utilisation graph would look that below.
- Each vertical line corresponds to a process becoming active for the time shown horizontally, this would occur only in response to say a key press at the keyboard. As you can see there is a lot of spare processing capacity in the white space.



Operating System Scheduling

Mixing Numerical Intensive and Interrupt Driven Interactive Processes

- Imagine now we introduce both interactive and numerically intensive processes into our system equipped with Interrupt driven IO. For example, suppose we had a system running

  1. A numerically intensive C++ compilation requiring several minutes of CPU time
  2. An interactive editing operation.

- The C++ compiler can now be given all the CPU time that is available whenever the editor is suspended by the operating system due to it being blocked by an IO request. This arrangement is shown in the graph below. It highlights the fact that the editor gains from this scheme due to the improved response time of an interrupt driven IO scheme, while the compiler gains by virtue of the fact that it does not have to compete for CPU time with a polling process.