History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

The choices can be made according to many criteria

History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

The choices can be made according to many criteria

- how long a program has been running

History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

The choices can be made according to many criteria

- how long a program has been running
- a *priority* of a program

History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

The choices can be made according to many criteria

- how long a program has been running
- a *priority* of a program
- whether a program is likely to need CPU very soon, or can wait

History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

The choices can be made according to many criteria

- how long a program has been running
- a *priority* of a program
- whether a program is likely to need CPU very soon, or can wait
- how much the owner of the program has paid

History

So the monitor needs to make decisions on what to program to run next: it is *scheduling* the programs

The choices can be made according to many criteria

- how long a program has been running
- a *priority* of a program
- whether a program is likely to need CPU very soon, or can wait
- how much the owner of the program has paid
- And many more things

History

Early scheduling algorithms were very simple, e.g., keep running the same program until it’s done; later algorithms tried to be more clever

History

Early scheduling algorithms were very simple, e.g., keep running the same program until it's done; later algorithms tried to be more clever

Some programmers would write their programs to take advantage of deficiencies in the scheduling algorithm: in the worst case *starve* other programs of any CPU time at all!

History

Early scheduling algorithms were very simple, e.g., keep running the same program until it's done; later algorithms tried to be more clever

Some programmers would write their programs to take advantage of deficiencies in the scheduling algorithm: in the worst case *starve* other programs of any CPU time at all!

It is tempting to make the scheduling algorithm complicated: but remember more time spent in the monitor deciding what to schedule next is less time for the programs

History

Early scheduling algorithms were very simple, e.g., keep running the same program until it's done; later algorithms tried to be more clever

Some programmers would write their programs to take advantage of deficiencies in the scheduling algorithm: in the worst case *starve* other programs of any CPU time at all!

It is tempting to make the scheduling algorithm complicated: but remember more time spent in the monitor deciding what to schedule next is less time for the programs

So there is a trade-off of making scheduling *fast* but *fair*

History

Early scheduling algorithms were very simple, e.g., keep running the same program until it's done; later algorithms tried to be more clever

Some programmers would write their programs to take advantage of deficiencies in the scheduling algorithm: in the worst case *starve* other programs of any CPU time at all!

It is tempting to make the scheduling algorithm complicated: but remember more time spent in the monitor deciding what to schedule next is less time for the programs

So there is a trade-off of making scheduling *fast* but *fair*

This is still an issue today: we'll look a little into scheduling later

History

A badly written (or malicious) program can bring the whole system down

History

A badly written (or malicious) program can bring the whole system down

If the program never hands control back to the OS (we'll call the monitor the operating system from now on), the OS never gets to run and schedule another program

History

A badly written (or malicious) program can bring the whole system down

If the program never hands control back to the OS (we'll call the monitor the operating system from now on), the OS never gets to run and schedule another program

If the program goes into an infinite loop the whole computer is jammed

History

A badly written (or malicious) program can bring the whole system down

If the program never hands control back to the OS (we'll call the monitor the operating system from now on), the OS never gets to run and schedule another program

If the program goes into an infinite loop the whole computer is jammed

This *cooperative* approach needs something extra

History

Interrupts can be used to solve the problem of runaway programs

History

Interrupts can be used to solve the problem of runaway programs

A hardware clock or *timer* can be set to send interrupts regularly after an appropriate period of time has elapsed

History

Interrupts can be used to solve the problem of runaway programs

A hardware clock or *timer* can be set to send interrupts regularly after an appropriate period of time has elapsed

When the interrupt is taken, the interrupt service routine jumps to the OS and so it can decide what to do next, including:

History

Interrupts can be used to solve the problem of runaway programs

A hardware clock or *timer* can be set to send interrupts regularly after an appropriate period of time has elapsed

When the interrupt is taken, the interrupt service routine jumps to the OS and so it can decide what to do next, including:

- resume running the interrupted program

History

Interrupts can be used to solve the problem of runaway programs

A hardware clock or *timer* can be set to send interrupts regularly after an appropriate period of time has elapsed

When the interrupt is taken, the interrupt service routine jumps to the OS and so it can decide what to do next, including:

- resume running the interrupted program
- kill (no longer run and remove resources from) the program if it has used up its allotted resources

History

Interrupts can be used to solve the problem of runaway programs

A hardware clock or *timer* can be set to send interrupts regularly after an appropriate period of time has elapsed

When the interrupt is taken, the interrupt service routine jumps to the OS and so it can decide what to do next, including:

- resume running the interrupted program
- kill (no longer run and remove resources from) the program if it has used up its allotted resources
- switch to running some other program

History

Interrupts can be used to solve the problem of runaway programs

A hardware clock or *timer* can be set to send interrupts regularly after an appropriate period of time has elapsed

When the interrupt is taken, the interrupt service routine jumps to the OS and so it can decide what to do next, including:

- resume running the interrupted program
- kill (no longer run and remove resources from) the program if it has used up its allotted resources
- switch to running some other program

Similarly, interrupts from peripherals like terminals or disks pass control to the OS

History

This is called *preemptive* scheduling and enables *timesharing*

History

This is called *preemptive* scheduling and enables *timesharing*

Timesharing is where several programs share the available CPU time and so appear to be running simultaneously

History

This is called *preemptive* scheduling and enables *timesharing*

Timesharing is where several programs share the available CPU time and so appear to be running simultaneously

Usually in a fairly transparent (to the programs) manner

History

This is called *preemptive* scheduling and enables *timesharing*

Timesharing is where several programs share the available CPU time and so appear to be running simultaneously

Usually in a fairly transparent (to the programs) manner

Always mediated by the OS, of course

History

The same interrupt mechanism allowed the use of *terminals*, where users could now interact directly with the computer, not just via job submission

History

The same interrupt mechanism allowed the use of *terminals*, where users could now interact directly with the computer, not just via job submission

A program can sit and wait (i.e., not be scheduled to run by the OS) until the user hits a key on the terminal

History

The same interrupt mechanism allowed the use of *terminals*, where users could now interact directly with the computer, not just via job submission

A program can sit and wait (i.e., not be scheduled to run by the OS) until the user hits a key on the terminal

When a key is hit, an interrupt happens, the OS takes over, schedules and runs the appropriate program to deal with the keystroke

History

Thus the waiting program uses no CPU resources until they are needed

History

Thus the waiting program uses no CPU resources until they are needed

Of course, while we say “the program is waiting”, is important to realised that it’s not “waiting”: the program is not even running

History

Thus the waiting program uses no CPU resources until they are needed

Of course, while we say “the program is waiting”, is important to realised that it’s not “waiting”: the program is not even running

So interrupts like this are another way of bridging the gap between slow humans and fast computers

History

Typically, timer interrupts are set to go off fairly often

History

Typically, timer interrupts are set to go off fairly often

- Frequent interrupts mean several programs can get a slice of the CPU quite often

History

Typically, timer interrupts are set to go off fairly often

- Frequent interrupts mean several programs can get a slice of the CPU quite often
- With sufficiently frequent interrupts it appears to a human observer that several programs are running simultaneously

History

Typically, timer interrupts are set to go off fairly often

- Frequent interrupts mean several programs can get a slice of the CPU quite often
- With sufficiently frequent interrupts it appears to a human observer that several programs are running simultaneously
- An *interactive* program, one where a human is involved, will appear to be dedicated to that user: in reality humans are so slow we can't appreciate how little time the computer gives us

History

Typically, timer interrupts are set to go off fairly often

- Frequent interrupts mean several programs can get a slice of the CPU quite often
- With sufficiently frequent interrupts it appears to a human observer that several programs are running simultaneously
- An *interactive* program, one where a human is involved, will appear to be dedicated to that user: in reality humans are so slow we can't appreciate how little time the computer gives us
- It is important to remember that a single processor can only do one thing at a time: it is only the *appearance* of multiple programs running simultaneously

History

On the other hand, too frequent interrupts mean the OS is forever being called and using CPU time, so less time is available for the programs

History

On the other hand, too frequent interrupts mean the OS is forever being called and using CPU time, so less time is available for the programs

This is another tradeoff: frequent interrupts for good interactive behaviour, rare interrupts for good compute behaviour

History

On the other hand, too frequent interrupts mean the OS is forever being called and using CPU time, so less time is available for the programs

This is another tradeoff: frequent interrupts for good interactive behaviour, rare interrupts for good compute behaviour

Clever scheduling algorithms in the OS try to give high priority but small slices of time to interactive programs; and lower priority but larger slices to compute-intensive programs

History

On the other hand, too frequent interrupts mean the OS is forever being called and using CPU time, so less time is available for the programs

This is another tradeoff: frequent interrupts for good interactive behaviour, rare interrupts for good compute behaviour

Clever scheduling algorithms in the OS try to give high priority but small slices of time to interactive programs; and lower priority but larger slices to compute-intensive programs

A “large slice of time” means the OS will allow a program to continue running for a relatively long amount of time before scheduling a different program

History

A “small slice of time” means the OS will deschedule the program after only a brief amount of running time

History

A “small slice of time” means the OS will deschedule the program after only a brief amount of running time

Thus, the OS can deal out CPU time to the programs in appropriately sized chunks

History

A “small slice of time” means the OS will deschedule the program after only a brief amount of running time

Thus, the OS can deal out CPU time to the programs in appropriately sized chunks

This is all part of the scheduling decision computations that happen potentially every time the OS runs

History

A “small slice of time” means the OS will deschedule the program after only a brief amount of running time

Thus, the OS can deal out CPU time to the programs in appropriately sized chunks

This is all part of the scheduling decision computations that happen potentially every time the OS runs

My PC is running at about 150 interrupts per second (timers and other stuff)

History

Low power gadgets like to keep the number of interrupts down, too, as it increases the amount of time the CPU can be idling in low power *sleep states*

History

Low power gadgets like to keep the number of interrupts down, too, as it increases the amount of time the CPU can be idling in low power *sleep states*

Tuning an OS is very difficult and depends critically on the application

History

Low power gadgets like to keep the number of interrupts down, too, as it increases the amount of time the CPU can be idling in low power *sleep states*

Tuning an OS is very difficult and depends critically on the application

When an OS spends more time deciding what to do than doing useful work, it is called *thrashing*

History

Low power gadgets like to keep the number of interrupts down, too, as it increases the amount of time the CPU can be idling in low power *sleep states*

Tuning an OS is very difficult and depends critically on the application

When an OS spends more time deciding what to do than doing useful work, it is called *thrashing*

Many early OSs had a big problem with thrashing

Question

Exercise. To think on: should the OS be subject to timer interrupts and preemption?

□