History

Back to another problem: a badly written (or malicious) program can bring the whole system down

History

Back to another problem: a badly written (or malicious) program can bring the whole system down

If a 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

Back to another problem: a badly written (or malicious) program can bring the whole system down

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

History

Back to another problem: a badly written (or malicious) program can bring the whole system down

If a 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 a 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 (e.g., CPU time)

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 (e.g., CPU time)
- 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 (e.g., CPU time)
- switch to running some other program

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

History

This kind of approach is called *preemptive* scheduling and enables *timesharing*

History

This kind of approach is called *preemptive* scheduling and enables *timesharing*

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

History

This kind of approach is called *preemptive* scheduling and enables *timesharing*

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

And usually in a fairly transparent manner to the programs

History

This kind of approach is called *preemptive* scheduling and enables *timesharing*

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

And usually in a fairly transparent manner to the programs

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”, it 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”, it 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

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

Of course, while we say “the program is waiting”, it 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

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

History

The OS probably won't choose a different program to run on *every* interrupt

History

The OS probably won't choose a different program to run on *every* interrupt

It will make decisions (see later) based on what the various programs need

History

The OS probably won't choose a different program to run on *every* interrupt

It will make decisions (see later) based on what the various programs need

A compute-intensive program might get a large slice of time

History

The OS probably won't choose a different program to run on *every* interrupt

It will make decisions (see later) based on what the various programs need

A compute-intensive program might get a large slice of time

This means the OS will continue to schedule the same program over many timer interrupts

History

An interactive program — one that spends most of its life waiting for a user to do something — doesn't need much CPU, so the OS would only give it a small slice of time

History

An interactive program — one that spends most of its life waiting for a user to do something — doesn't need much CPU, so the OS would only give it a small slice of time

Meaning it will deschedule the program after a few (possibly just one) timer interrupt

History

An interactive program — one that spends most of its life waiting for a user to do something — doesn't need much CPU, so the OS would only give it a small slice of time

Meaning it will deschedule the program after a few (possibly just one) timer interrupt

We shall return to scheduling later

History

Next: the programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

History

Next: the programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

This has to be done by hardware support as it needs to be fast and unobtrusive: every memory access needs to be checked

History

Next: the programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

This has to be done by hardware support as it needs to be fast and unobtrusive: every memory access needs to be checked

So it has to be hardware supported and not just software

History

Next: the programs and OS all live in the same computer memory: we need some way of protecting programs and the OS from each other

This has to be done by hardware support as it needs to be fast and unobtrusive: every memory access needs to be checked

So it has to be hardware supported and not just software

We shall start by looking at general hardware protection mechanisms

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like arithmetic operations, loads, stores, jumps and so on. Any program can execute these

History

Certain operations, like accessing tape or a printer, must be reserved for use by the OS and not be accessible by a random user program

So in the hardware (CPU) machine instructions are divided into two (or more) classes

- Unprivileged operations. Like arithmetic operations, loads, stores, jumps and so on. Any program can execute these
- Privileged operations. Like access peripherals, reboot the machine. Only certain privileged programs can run these

And the processor hardware can run in two (or more) modes

And the processor hardware can run in two (or more) modes

- Unprivileged. Normal computation, called *user mode*

And the processor hardware can run in two (or more) modes

- Unprivileged. Normal computation, called *user mode*
- Privileged. For systems operation, called *kernel mode*

History

Modern processor architectures can have more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

History

Modern processor architectures can have more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

For example, the Intel x86 architecture has four *rings*. Ring 0 can execute any instruction, while Ring 3 is for user mode. Rings 1 and 2 are rarely used these days

History

Modern processor architectures can have more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

For example, the Intel x86 architecture has four *rings*. Ring 0 can execute any instruction, while Ring 3 is for user mode. Rings 1 and 2 are rarely used these days

OS/2 used Ring 2

History

Modern processor architectures can have more levels of privilege, but for the most part it is rare that more than two levels are used in commodity computers

For example, the Intel x86 architecture has four *rings*. Ring 0 can execute any instruction, while Ring 3 is for user mode. Rings 1 and 2 are rarely used these days

OS/2 used Ring 2

The latest Intel and AMD architectures added a Ring -1 (for OS virtualisation)

History

Exercise And it doesn't stop there. Read about rings -2 and -3

Exercise The ARM architecture has 3 levels. Read about this

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

If an unprivileged program (i.e., a program running in an unprivileged mode) tries to execute a privileged operation the hardware causes an interrupt (also called a system *trap*) and sets the processor to privileged mode. The interrupt service routine then jumps to the OS

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

If an unprivileged program (i.e., a program running in an unprivileged mode) tries to execute a privileged operation the hardware causes an interrupt (also called a system *trap*) and sets the processor to privileged mode. The interrupt service routine then jumps to the OS

The OS is now running (in privileged mode) and can then decide what to do

History

Note that privilege is a state of the *processor*, not the program, but we tend to say “a privileged program” rather than “a program running with the CPU in privileged mode”

If an unprivileged program (i.e., a program running in an unprivileged mode) tries to execute a privileged operation the hardware causes an interrupt (also called a system *trap*) and sets the processor to privileged mode. The interrupt service routine then jumps to the OS

The OS is now running (in privileged mode) and can then decide what to do

For example, the OS may decide to disallow the operation, and kill the program (i.e., not run it any more)

History

The system starts in kernel (privileged) mode

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special “call OS” (or *syscall*) instruction that jumps to the OS

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special "call OS" (or *syscall*) instruction that jumps to the OS
6. This enables privileged mode, so the OS regains control, with privilege

History

The system starts in kernel (privileged) mode

1. The OS decides which process to schedule
2. It uses a special jump-and-drop-privilege instruction to start running the program
3. The program runs user mode (unprivileged)
4. The program finishes or decides it needs a system resource
5. The program executes a special "call OS" (or *syscall*) instruction that jumps to the OS
6. This enables privileged mode, so the OS regains control, with privilege
7. The OS decides what to do next

History

Of course, even if the program does not do a syscall, a timer interrupt will come along at some point, anyway

History

Of course, even if the program does not do a syscall, a timer interrupt will come along at some point, anyway

The syscall instruction always jumps to the same place in the OS. So the program cannot use it to gain privilege for itself and run its own code privileged

History

Of course, even if the program does not do a syscall, a timer interrupt will come along at some point, anyway

The syscall instruction always jumps to the same place in the OS. So the program cannot use it to gain privilege for itself and run its own code privileged

This to-ing and fro-ing between modes ensures that the OS is running in privileged mode and the user program is running in unprivileged mode

History

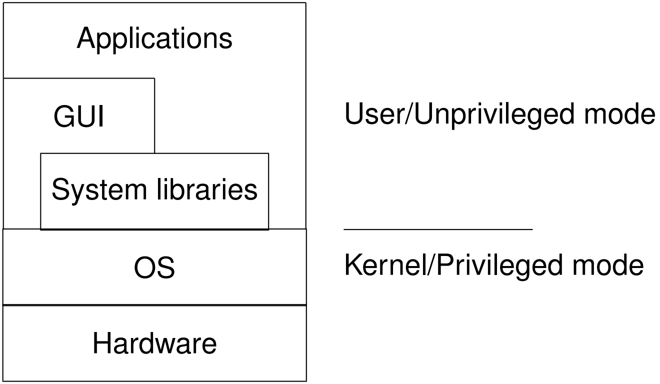
Of course, even if the program does not do a syscall, a timer interrupt will come along at some point, anyway

The syscall instruction always jumps to the same place in the OS. So the program cannot use it to gain privilege for itself and run its own code privileged

This to-ing and fro-ing between modes ensures that the OS is running in privileged mode and the user program is running in unprivileged mode

And the user program can never manage to get into privileged mode as every transition to privileged mode is tied by the hardware to a jump to the OS

History



There is a strict divide between kernel (OS) code and user code, controlled by the hardware

History

Unless there are bugs in the kernel code

History

Unless there are bugs in the kernel code

Or you don't maintain the proper separation between OS and everything else: revisit the diagrams from earlier

History

Unless there are bugs in the kernel code

Or you don't maintain the proper separation between OS and everything else: revisit the diagrams from earlier

Incidentally, the system libraries usually include a bunch of "nice" interfaces to the syscalls: wrapping them to make using them easier

History

Unless there are bugs in the kernel code

Or you don’t maintain the proper separation between OS and everything else: revisit the diagrams from earlier

Incidentally, the system libraries usually include a bunch of “nice” interfaces to the syscalls: wrapping them to make using them easier

E.g., the “open file” syscall might need certain values (file name, etc.) to be placed in certain CPU registers; and the “open file” code to be placed in a register before the syscall

History

Unless there are bugs in the kernel code

Or you don't maintain the proper separation between OS and everything else: revisit the diagrams from earlier

Incidentally, the system libraries usually include a bunch of "nice" interfaces to the syscalls: wrapping them to make using them easier

E.g., the "open file" syscall might need certain values (file name, etc.) to be placed in certain CPU registers; and the "open file" code to be placed in a register before the syscall

The `open` system library function simply hides these details from the programmer

History

The result of all this messing with modes is certain operations like loading programs, or accessing hardware like a printer, are only available to the OS

History

The result of all this messing with modes is certain operations like loading programs, or accessing hardware like a printer, are only available to the OS

If an unprivileged program tries to access the printer directly, that again trips an interrupt and the OS takes over anyway

History

The result of all this messing with modes is certain operations like loading programs, or accessing hardware like a printer, are only available to the OS

If an unprivileged program tries to access the printer directly, that again trips an interrupt and the OS takes over anyway

Forcing access to hardware via the OS also provides protection and management for other system resources, like access to files or the network

History

The result of all this messing with modes is certain operations like loading programs, or accessing hardware like a printer, are only available to the OS

If an unprivileged program tries to access the printer directly, that again trips an interrupt and the OS takes over anyway

Forcing access to hardware via the OS also provides protection and management for other system resources, like access to files or the network

In kernel mode, everything is possible

History

The result of all this messing with modes is certain operations like loading programs, or accessing hardware like a printer, are only available to the OS

If an unprivileged program tries to access the printer directly, that again trips an interrupt and the OS takes over anyway

Forcing access to hardware via the OS also provides protection and management for other system resources, like access to files or the network

In kernel mode, everything is possible

In user mode, only “safe” things are possible

History

Preemption and protection appeared in OSs for large mainframe computers and Unix for minicomputers in the late 1960s

History

Preemption and protection appeared in OSs for large mainframe computers and Unix for minicomputers in the late 1960s

When microcomputers (IBM PC) arrived in the early 1980s much of OS knowledge was thrown away and DOS (Disk Operating System) was non-preemptive, single process and no protection

History

Preemption and protection appeared in OSs for large mainframe computers and Unix for minicomputers in the late 1960s

When microcomputers (IBM PC) arrived in the early 1980s much of OS knowledge was thrown away and DOS (Disk Operating System) was non-preemptive, single process and no protection

This was because the earliest PC hardware did not support such things (no rings)

History

Support was rapidly added in later PC hardware, but DOS and, later, Windows 3.1 took no advantage of it: the lack of protection meaning a single bad program could mess up the OS and crash the entire computer

History

Support was rapidly added in later PC hardware, but DOS and, later, Windows 3.1 took no advantage of it: the lack of protection meaning a single bad program could mess up the OS and crash the entire computer

Windows NT was the first true OS from Microsoft (mid 1990s) for PCs, possibly as much as a decade after other OSs (such as Unix derivatives) were providing preemption and protection on the same hardware

History

Support was rapidly added in later PC hardware, but DOS and, later, Windows 3.1 took no advantage of it: the lack of protection meaning a single bad program could mess up the OS and crash the entire computer

Windows NT was the first true OS from Microsoft (mid 1990s) for PCs, possibly as much as a decade after other OSs (such as Unix derivatives) were providing preemption and protection on the same hardware

Incidentally, Microsoft's need for backwards compatability with these early systems is a major reason why they have so many problems with security

History

Back to memory protection: this must stop a program from writing and/or reading the memory used by another program or by the OS

History

Back to memory protection: this must stop a program from writing and/or reading the memory used by another program or by the OS

The OS must be allowed to read and write any part of memory

History

Back to memory protection: this must stop a program from writing and/or reading the memory used by another program or by the OS

The OS must be allowed to read and write any part of memory

Again, there must be hardware support to do this to make it fast

History

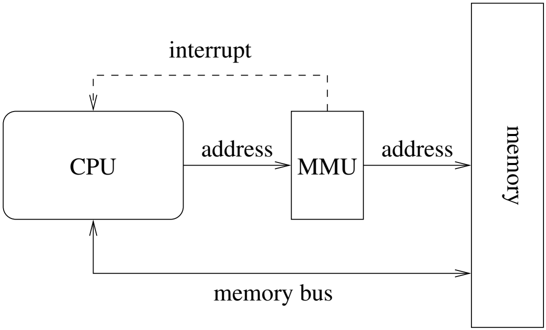
Back to memory protection: this must stop a program from writing and/or reading the memory used by another program or by the OS

The OS must be allowed to read and write any part of memory

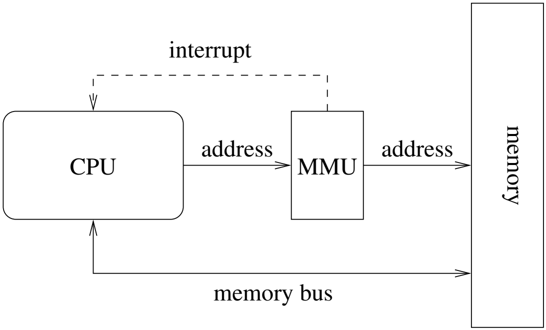
Again, there must be hardware support to do this to make it fast

There is a table of flags in a special piece of hardware: the *memory management unit* (MMU). These flags say whether the *currently running* (user mode) program can read or write a given area of memory

History

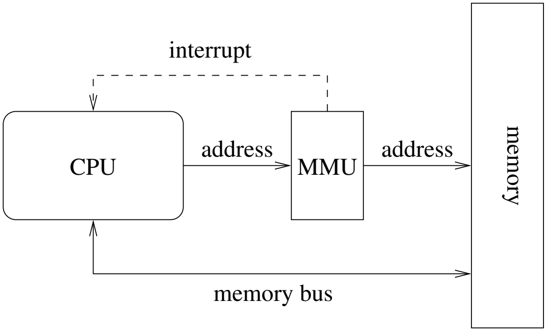


History



One bit to say if an area is readable; another to say if it is writable by the current program

History



One bit to say if an area is readable; another to say if it is writable by the current program

It is often useful to separate ability to read from ability to write

History

Setting these flags in the MMU is a privileged operation, of course

History

Setting these flags in the MMU is a privileged operation, of course

And if an unprivileged program tries to read or write to an area of memory for which it does not have the required permission (say some other program's or the OS's memory) the MMU raises an interrupt and the OS takes control again

History

Setting these flags in the MMU is a privileged operation, of course

And if an unprivileged program tries to read or write to an area of memory for which it does not have the required permission (say some other program's or the OS's memory) the MMU raises an interrupt and the OS takes control again

It would not be feasible to have control like this on a byte-by-byte level, so memory is divided into blocks called *pages*

History

Setting these flags in the MMU is a privileged operation, of course

And if an unprivileged program tries to read or write to an area of memory for which it does not have the required permission (say some other program's or the OS's memory) the MMU raises an interrupt and the OS takes control again

It would not be feasible to have control like this on a byte-by-byte level, so memory is divided into blocks called *pages*

A page is just a contiguous area of memory: 4096 bytes is popular on modern machines, though current hardware can support 4MB pages

History

A page is marked as read/writable as a whole: this makes this technique practical

History

A page is marked as read/writable as a whole: this makes this technique practical

Exercise How many flags (bits) are needed to cover 2GB? How many bytes of flags does that correspond to?

History

A page is marked as read/writable as a whole: this makes this technique practical

Exercise How many flags (bits) are needed to cover 2GB? How many bytes of flags does that correspond to?

Note that there is a set of flags for *each* program, and are part of the program's state that must be saved and restored when that program is re-scheduled

History

A page is marked as read/writable as a whole: this makes this technique practical

Exercise How many flags (bits) are needed to cover 2GB? How many bytes of flags does that correspond to?

Note that there is a set of flags for *each* program, and are part of the program's state that must be saved and restored when that program is re-scheduled

There is usually also an *executable* flag: can you execute code from this memory address?

History

Every read or write to memory is checked by the MMU before it is allowed: this means the hardware that does this check has to be very fast

History

Every read or write to memory is checked by the MMU before it is allowed: this means the hardware that does this check has to be very fast

We shall not be going into this in depth here, because in modern machines this is enhanced by the notion of *virtual memory*

History

Every read or write to memory is checked by the MMU before it is allowed: this means the hardware that does this check has to be very fast

We shall not be going into this in depth here, because in modern machines this is enhanced by the notion of *virtual memory*

This we shall cover later, but it builds on the ideas above and provides a much more flexible method of protection

History

But for now: the OS sets the MMU flags to say which pages of memory are accessible for the current program

History

But for now: the OS sets the MMU flags to say which pages of memory are accessible for the current program

And every memory access is checked

History

But for now: the OS sets the MMU flags to say which pages of memory are accessible for the current program

And every memory access is checked

An interrupt is raised if the program tries to read or write memory that is not allocated to it

History

So what is the current state of OSs with regard to preemption and memory protection?

History

So what is the current state of OSs with regard to preemption and memory protection?

In current large OSs we have:

History

So what is the current state of OSs with regard to preemption and memory protection?

In current large OSs we have:

- Windows. Preemptive multitasking from Windows NT (1996) onwards. Previously (Windows 95 etc.) was little more than a monitor with a pretty interface on top

History

So what is the current state of OSs with regard to preemption and memory protection?

In current large OSs we have:

- Windows. Preemptive multitasking from Windows NT (1996) onwards. Previously (Windows 95 etc.) was little more than a monitor with a pretty interface on top
- Linux. A Unix re-implementation. Preemptive multitasking from inception (1991). (Recall that Unix had preemption from early 1970s)

History

So what is the current state of OSs with regard to preemption and memory protection?

In current large OSs we have:

- Windows. Preemptive multitasking from Windows NT (1996) onwards. Previously (Windows 95 etc.) was little more than a monitor with a pretty interface on top
- Linux. A Unix re-implementation. Preemptive multitasking from inception (1991). (Recall that Unix had preemption from early 1970s)
- MacOS. MacOS X is a Unix derivative (BSD), from 1999 onwards. Earlier systems (MacOS 9 and earlier) were completely different, with no preemption, only cooperative

History

- Solaris. A Unix derivative (System V). Preemptive multitasking from inception (1992), an extensive rewrite of the earlier SunOS (1983), another Unix variant (BSD)

History

- Solaris. A Unix derivative (System V). Preemptive multitasking from inception (1992), an extensive rewrite of the earlier SunOS (1983), another Unix variant (BSD)
- OS/2. Initially from Microsoft and IBM (1997), then just IBM as Microsoft went off to do its own thing. Intended to be the followup to DOS. Multitasking when the hardware could support it: OS/2 2.0 (1992) could run multiple copies of DOS/Windows simultaneously. Previously used a lot in bank ATMs (until IBM ended support in 2006). OS/2 3.0 became Windows NT

History

And thousands of others: but the major players in the PC market are either derived from Windows NT, or from Unix

History

And thousands of others: but the major players in the PC market are either derived from Windows NT, or from Unix

In contrast, in the embedded market are things are much more mixed, with both preemptive and cooperative OSs, as required by the application

History

And thousands of others: but the major players in the PC market are either derived from Windows NT, or from Unix

In contrast, in the embedded market are things are much more mixed, with both preemptive and cooperative OSs, as required by the application

All PC-level OSs have MMU protection (and more); while embedded systems have it if required, otherwise not (so not to have the cost of the MMU hardware)