Real-Time Operating Systems (0_KRI)
Operating System Architecture

Ivan Cibrario Bertolotti

IEIIT-CNR / Politecnico di Torino

Academic Year 2006-2007

Outline

What is a Real-Time System?

Definition

A real-time system is an information processing system which has to respond to external events both correctly and within a finite, specified period of time.

- The correctness and usefulness of the system depends not only on the logical results of its computations, but also on the time at which they are produced.
- Giving the right result too late may be as bad as giving the wrong result or giving no result at all.

What is a Real-Time System?

Definition

A real-time system is an information processing system which has to respond to external events both correctly and within a finite, specified period of time.

- The correctness and usefulness of the system depends not only on the logical results of its computations, but also on the time at which they are produced.
- Giving the right result too late may be as bad as giving the wrong result or giving no result at all.

What is a Real-Time System?

Definition

A real-time system is an information processing system which has to respond to external events both correctly and within a finite, specified period of time.

- The correctness and usefulness of the system depends not only on the logical results of its computations, but also on the time at which they are produced.
- Giving the right result too late may be as bad as giving the wrong result or giving no result at all.

Embedded Systems

Embedded Systems

In a real-time system, the computer is often embedded into, and interfaced directly to, some physical equipment, and controls or monitors its operation.

- The computer receives stimuli coming from the environment, and acts on the environment by means of dedicated I/O devices: sensors and actuators.
- The reaction must take place within a time frame dictated by the characteristics of the environment itself.
- The consequences of a late or missing reaction also depend on the environment.
- Over 90% of the processors sold each year are for the embedded systems market.

Embedded Systems

Embedded Systems

In a real-time system, the computer is often embedded into, and interfaced directly to, some physical equipment, and controls or monitors its operation.

- The computer receives stimuli coming from the environment, and acts on the environment by means of dedicated I/O devices: sensors and actuators.
- The reaction must take place within a time frame dictated by the characteristics of the environment itself.
- The consequences of a late or missing reaction also depend on the environment.
- Over 90% of the processors sold each year are for the embedded systems market.

Embedded Systems

Embedded Systems

In a real-time system, the computer is often embedded into, and interfaced directly to, some physical equipment, and controls or monitors its operation.

- The computer receives stimuli coming from the environment, and acts on the environment by means of dedicated I/O devices: sensors and actuators.
- The reaction must take place within a time frame dictated by the characteristics of the environment itself.
- The consequences of a late or missing reaction also depend on the environment.
- Over 90% of the processors sold each year are for the embedded systems market.

Embedded Systems

Embedded Systems

In a real-time system, the computer is often embedded into, and interfaced directly to, some physical equipment, and controls or monitors its operation.

- The computer receives stimuli coming from the environment, and acts on the environment by means of dedicated I/O devices: sensors and actuators.
- The reaction must take place within a time frame dictated by the characteristics of the environment itself.
- The consequences of a late or missing reaction also depend on the environment.
- Over 90% of the processors sold each year are for the embedded systems market.

Embedded Systems

Embedded Systems

In a real-time system, the computer is often embedded into, and interfaced directly to, some physical equipment, and controls or monitors its operation.

- The computer receives stimuli coming from the environment, and acts on the environment by means of dedicated I/O devices: sensors and actuators.
- The reaction must take place within a time frame dictated by the characteristics of the environment itself.
- The consequences of a late or missing reaction also depend on the environment.
- Over 90% of the processors sold each year are for the embedded systems market.

Hard, Firm, Soft Real-Time

Hard Real-Time

It is imperative that reactions occur within the specified deadline, because they are useless when late and missing a deadline leads to a catastrophe. Example: pacemaker.

Soft Real-Time

Response times are still important, but it is acceptable to occasionally miss a deadline (with a low probability). The value of a result decreases as its lateness increases. Example: video streaming.

Firm Real-Time

It is acceptable to occasionally miss a deadline, but there is no benefit from late delivery of a result. Example: financial applications.

Hard, Firm, Soft Real-Time

Hard Real-Time

It is imperative that reactions occur within the specified deadline, because they are useless when late and missing a deadline leads to a catastrophe. Example: pacemaker.

Soft Real-Time

Response times are still important, but it is acceptable to occasionally miss a deadline (with a low probability). The value of a result decreases as its lateness increases. Example: video streaming.

Firm Real-Time

It is acceptable to occasionally miss a deadline, but there is no benefit from late delivery of a result. Example: financial applications.

Hard, Firm, Soft Real-Time

Hard Real-Time

It is imperative that reactions occur within the specified deadline, because they are useless when late and missing a deadline leads to a catastrophe. Example: pacemaker.

Soft Real-Time

Response times are still important, but it is acceptable to occasionally miss a deadline (with a low probability). The value of a result decreases as its lateness increases. Example: video streaming.

Firm Real-Time

It is acceptable to occasionally miss a deadline, but there is no benefit from late delivery of a result. Example: financial applications.

Real-World Systems

- Often, real-world systems have both soft and hard real-time requirements.
- For example, the reaction to some warning event may have both an optimal, soft deadline which should be met most of the times, and a longer, hard deadline which guarantees that no damage takes place.
- Moreover, the term "soft" deadline encompasses several different properties, for example:
  - the deadline can be missed occasionally, with an upper limit of misses within a defined interval
  - the result can occasionally be delivered late, with an upper limit on its lateness

Real-World Systems

- Often, real-world systems have both soft and hard real-time requirements.
- For example, the reaction to some warning event may have both an optimal, soft deadline which should be met most of the times, and a longer, hard deadline which guarantees that no damage takes place.
- Moreover, the term “soft” deadline encompasses several different properties, for example:
  - the deadline can be missed occasionally, with an upper limit of misses within a defined interval
  - the result can occasionally be delivered late, with an upper limit on its lateness

Real-World Systems

- Often, real-world systems have both soft and hard real-time requirements.
- For example, the reaction to some warning event may have both an optimal, soft deadline which should be met most of the times, and a longer, hard deadline which guarantees that no damage takes place.
- Moreover, the term “soft” deadline encompasses several different properties, for example:
  - the deadline can be missed occasionally, with an upper limit of misses within a defined interval
  - the result can occasionally be delivered late, with an upper limit on its lateness

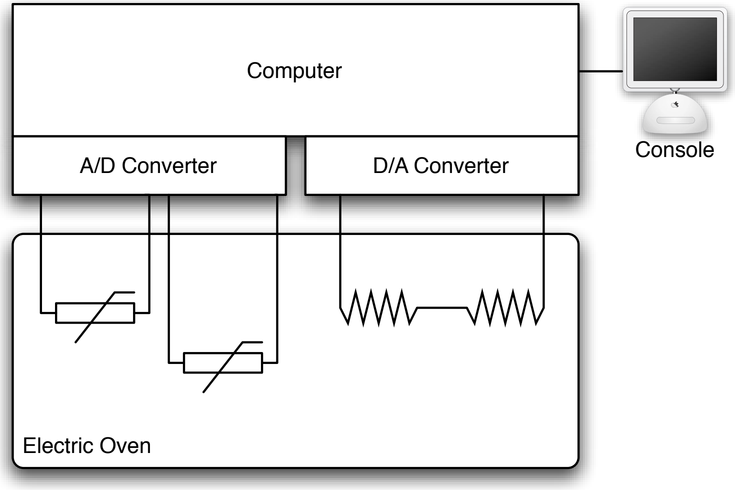
Real-World Systems

- Often, real-world systems have both soft and hard real-time requirements.
- For example, the reaction to some warning event may have both an optimal, soft deadline which should be met most of the times, and a longer, hard deadline which guarantees that no damage takes place.
- Moreover, the term “soft” deadline encompasses several different properties, for example:
  - the deadline can be missed occasionally, with an upper limit of misses within a defined interval
  - the result can occasionally be delivered late, with an upper limit on its lateness

Real-World Systems

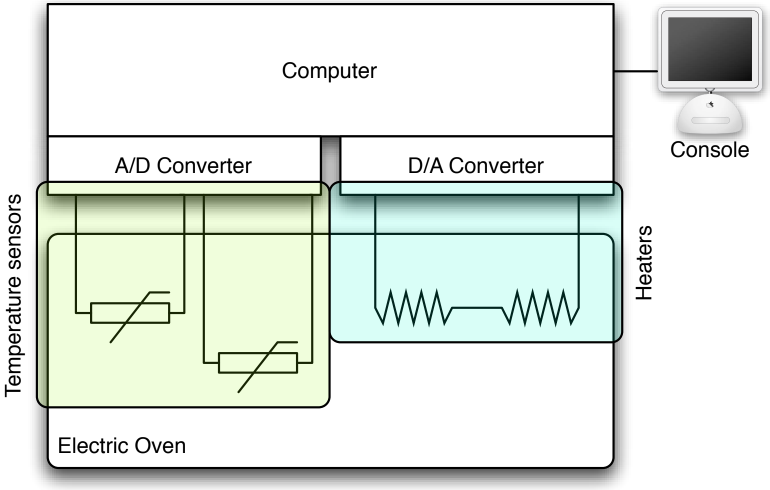
- Often, real-world systems have both soft and hard real-time requirements.
- For example, the reaction to some warning event may have both an optimal, soft deadline which should be met most of the times, and a longer, hard deadline which guarantees that no damage takes place.
- Moreover, the term “soft” deadline encompasses several different properties, for example:
  - the deadline can be missed occasionally, with an upper limit of misses within a defined interval
  - the result can occasionally be delivered late, with an upper limit on its lateness

An Example of Real-Time System



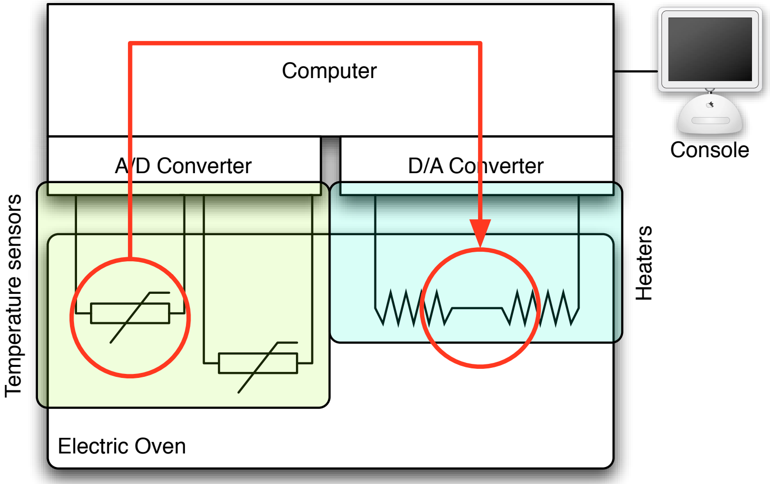
A computer can control the temperature of an electric oven.

An Example of Real-Time System



It interacts with the environment by means of sensors and actuators (heaters in this case).

An Example of Real-Time System



It must react to any change of temperature by properly regulating the heaters within a specified deadline.

Characteristics of a Real-Time System

- Real-time systems are often complex, because they must respond to the changing requirements of the real world and therefore undergo continuous maintenance and extension.
- Real-world elements naturally operate in parallel and the computer must interact with them.

The Concurrent Programming Role

For both reasons, a major (and important) problem is how to express concurrency in a program, and how to solve the resulting synchronization and communication problems. This is the goal of concurrent programming.

Characteristics of a Real-Time System

- Real-time systems are often complex, because they must respond to the changing requirements of the real world and therefore undergo continuous maintenance and extension.
- Real-world elements naturally operate in parallel and the computer must interact with them.

The Concurrent Programming Role

For both reasons, a major (and important) problem is how to express concurrency in a program, and how to solve the resulting synchronization and communication problems. This is the goal of concurrent programming.

Characteristics of a Real-Time System

- Real-time systems are often complex, because they must respond to the changing requirements of the real world and therefore undergo continuous maintenance and extension.
- Real-world elements naturally operate in parallel and the computer must interact with them.

The Concurrent Programming Role

For both reasons, a major (and important) problem is how to express concurrency in a program, and how to solve the resulting synchronization and communication problems. This is the goal of concurrent programming.

Additional Requirements

- Language and run-time support is required to enable the processes to synchronize with time itself, for example to specify times at which actions are to be performed.
- Since real-time systems are often time-critical, efficiency of implementation will be more important than in other systems. Programmers must be concerned with the cost of using a particular language or operating system feature.
- It is necessary to interact with special-purpose hardware and to be able to program devices in an easy and abstract way.
- Real-time systems must be extremely reliable and safe.

Additional Requirements

- Language and run-time support is required to enable the processes to synchronize with time itself, for example to specify times at which actions are to be performed.
- Since real-time systems are often time-critical, efficiency of implementation will be more important than in other systems. Programmers must be concerned with the cost of using a particular language or operating system feature.
- It is necessary to interact with special-purpose hardware and to be able to program devices in an easy and abstract way.
- Real-time systems must be extremely reliable and safe.

Additional Requirements

- Language and run-time support is required to enable the processes to synchronize with time itself, for example to specify times at which actions are to be performed.
- Since real-time systems are often time-critical, efficiency of implementation will be more important than in other systems. Programmers must be concerned with the cost of using a particular language or operating system feature.
- It is necessary to interact with special-purpose hardware and to be able to program devices in an easy and abstract way.
- Real-time systems must be extremely reliable and safe.

Additional Requirements

- Language and run-time support is required to enable the processes to synchronize with time itself, for example to specify times at which actions are to be performed.
- Since real-time systems are often time-critical, efficiency of implementation will be more important than in other systems. Programmers must be concerned with the cost of using a particular language or operating system feature.
- It is necessary to interact with special-purpose hardware and to be able to program devices in an easy and abstract way.
- Real-time systems must be extremely reliable and safe.

The Operating System Role

- A simple approach to expressing (and handling) concurrency is to leave it all up to the programmer, who must construct her system so that it involves the cyclic execution of a program sequence which handles the various, concurrent tasks.
- Another possibility is to rely on language and/or operating system support for concurrency. In this case, a set of concurrent programming primitives is available to make the programmer's task easier, and to avoid re-inventing the wheel each time.

Real-time operating systems are gaining popularity in modern embedded systems, especially when the application is large and complex.

The Operating System Role

- A simple approach to expressing (and handling) concurrency is to leave it all up to the programmer, who must construct her system so that it involves the cyclic execution of a program sequence which handles the various, concurrent tasks.
- Another possibility is to rely on language and/or operating system support for concurrency. In this case, a set of concurrent programming primitives is available to make the programmer's task easier, and to avoid re-inventing the wheel each time.

Real-time operating systems are gaining popularity in modern embedded systems, especially when the application is large and complex.

The Operating System Role

- A simple approach to expressing (and handling) concurrency is to leave it all up to the programmer, who must construct her system so that it involves the cyclic execution of a program sequence which handles the various, concurrent tasks.
- Another possibility is to rely on language and/or operating system support for concurrency. In this case, a set of concurrent programming primitives is available to make the programmer's task easier, and to avoid re-inventing the wheel each time.

Real-time operating systems are gaining popularity in modern embedded systems, especially when the application is large and complex.

What is an Operating System?

Operating systems perform two, mostly unrelated, functions: extending the machine and managing resources.

1. The architecture of most computers is quite primitive and difficult to program, especially for I/O. On the other hand, programmers do not like to get involved with its details and want to deal with a simpler, higher-level and hardware-independent set of services instead.
2. When multiple processes run concurrently, the operating system must provide for an orderly management and allocation of the system resources (processors, memory, devices, . . .) among them. Moreover, it must ensure that different processes cannot interfere with one another (either by accident or by purpose).

What is an Operating System?

Operating systems perform two, mostly unrelated, functions: extending the machine and managing resources.

1. The architecture of most computers is quite primitive and difficult to program, especially for I/O. On the other hand, programmers do not like to get involved with its details and want to deal with a simpler, higher-level and hardware-independent set of services instead.
2. When multiple processes run concurrently, the operating system must provide for an orderly management and allocation of the system resources (processors, memory, devices, ...) among them. Moreover, it must ensure that different processes cannot interfere with one another (either by accident or by purpose).

What is an Operating System?

Operating systems perform two, mostly unrelated, functions: extending the machine and managing resources.

1. The architecture of most computers is quite primitive and difficult to program, especially for I/O. On the other hand, programmers do not like to get involved with its details and want to deal with a simpler, higher-level and hardware-independent set of services instead.
2. When multiple processes run concurrently, the operating system must provide for an orderly management and allocation of the system resources (processors, memory, devices, . . .) among them. Moreover, it must ensure that different processes cannot interfere with one another (either by accident or by purpose).

Operating System Structure

Monolithic systems: It was the most common "non-structure" of older operating systems, and it is still widespread. The operating system is written as an unstructured collection of procedures, which can freely call each other.

Layered systems: The operating system is organized as a hierarchy of layers, each one constructed upon the one below it.

Microkernel: The operating system is a set of processes, which are independent of each other and cooperate to realize the operating system functions. They communicate by means of a microkernel that, ideally, only transports messages between processes.

Virtual machines: Multiprogramming is provided by several virtual machines that are an exact copy of the bare hardware. Each of them runs its own operating system.

Operating System Structure

Monolithic systems: It was the most common “non-structure” of older operating systems, and it is still widespread. The operating system is written as an unstructured collection of procedures, which can freely call each other.

Layered systems: The operating system is organized as a hierarchy of layers, each one constructed upon the one below it.

Microkernel: The operating system is a set of processes, which are independent of each other and cooperate to realize the operating system functions. They communicate by means of a microkernel that, ideally, only transports messages between processes.

Virtual machines: Multiprogramming is provided by several virtual machines that are an exact copy of the bare hardware. Each of them runs its own operating system.

Operating System Structure

Monolithic systems: It was the most common “non-structure” of older operating systems, and it is still widespread. The operating system is written as an unstructured collection of procedures, which can freely call each other.

Layered systems: The operating system is organized as a hierarchy of layers, each one constructed upon the one below it.

Microkernel: The operating system is a set of processes, which are independent of each other and cooperate to realize the operating system functions. They communicate by means of a microkernel that, ideally, only transports messages between processes.

Virtual machines: Multiprogramming is provided by several virtual machines that are an exact copy of the bare hardware. Each of them runs its own operating system.

Operating System Structure

Monolithic systems: It was the most common “non-structure” of older operating systems, and it is still widespread. The operating system is written as an unstructured collection of procedures, which can freely call each other.

Layered systems: The operating system is organized as a hierarchy of layers, each one constructed upon the one below it.

Microkernel: The operating system is a set of processes, which are independent of each other and cooperate to realize the operating system functions. They communicate by means of a microkernel that, ideally, only transports messages between processes.

Virtual machines: Multiprogramming is provided by several virtual machines that are an exact copy of the bare hardware. Each of them runs its own operating system.

Monolithic Systems (OS/360)

- The operating system is written as a collection of procedures. Each procedure has a well-defined interface, in terms of parameters and results, but each one is free to call any other one.
- To build the operating system executable image, all the procedures are first compiled individually and then bound together by the linker.
- There is essentially no information hiding, because every procedure is visible to every other procedure.
- It is possible to have a little structure by informally dividing the procedures into three groups, or levels:
  - A main procedure, that intercepts the system calls and invokes the requested service procedure.
  - A set of service procedures, that carry out the system calls.
  - A set of utility procedures, that help the service procedures by performing commonly-needed functions.

Monolithic Systems (OS/360)

- The operating system is written as a collection of procedures. Each procedure has a well-defined interface, in terms of parameters and results, but each one is free to call any other one.
- To build the operating system executable image, all the procedures are first compiled individually and then bound together by the linker.
- There is essentially no information hiding, because every procedure is visible to every other procedure.
- It is possible to have a little structure by informally dividing the procedures into three groups, or levels:
  - A main procedure, that intercepts the system calls and invokes the requested service procedure.
  - A set of service procedures, that carry out the system calls.
  - A set of utility procedures, that help the service procedures by performing commonly-needed functions.

Monolithic Systems (OS/360)

- The operating system is written as a collection of procedures. Each procedure has a well-defined interface, in terms of parameters and results, but each one is free to call any other one.
- To build the operating system executable image, all the procedures are first compiled individually and then bound together by the linker.
- There is essentially no information hiding, because every procedure is visible to every other procedure.
- It is possible to have a little structure by informally dividing the procedures into three groups, or levels:
  - A main procedure, that intercepts the system calls and invokes the requested service procedure.
  - A set of service procedures, that carry out the system calls.
  - A set of utility procedures, that help the service procedures by performing commonly-needed functions.

Monolithic Systems (OS/360)

- The operating system is written as a collection of procedures. Each procedure has a well-defined interface, in terms of parameters and results, but each one is free to call any other one.
- To build the operating system executable image, all the procedures are first compiled individually and then bound together by the linker.
- There is essentially no information hiding, because every procedure is visible to every other procedure.
- It is possible to have a little structure by informally dividing the procedures into three groups, or levels:
  - A main procedure, that intercepts the system calls and invokes the requested service procedure.
  - A set of service procedures, that carry out the system calls.
  - A set of utility procedures, that help the service procedures by performing commonly-needed functions.

Monolithic Systems (OS/360)

- The operating system is written as a collection of procedures. Each procedure has a well-defined interface, in terms of parameters and results, but each one is free to call any other one.
- To build the operating system executable image, all the procedures are first compiled individually and then bound together by the linker.
- There is essentially no information hiding, because every procedure is visible to every other procedure.
- It is possible to have a little structure by informally dividing the procedures into three groups, or levels:
  - A main procedure, that intercepts the system calls and invokes the requested service procedure.
  - A set of service procedures, that carry out the system calls.
  - A set of utility procedures, that help the service procedures by performing commonly-needed functions.

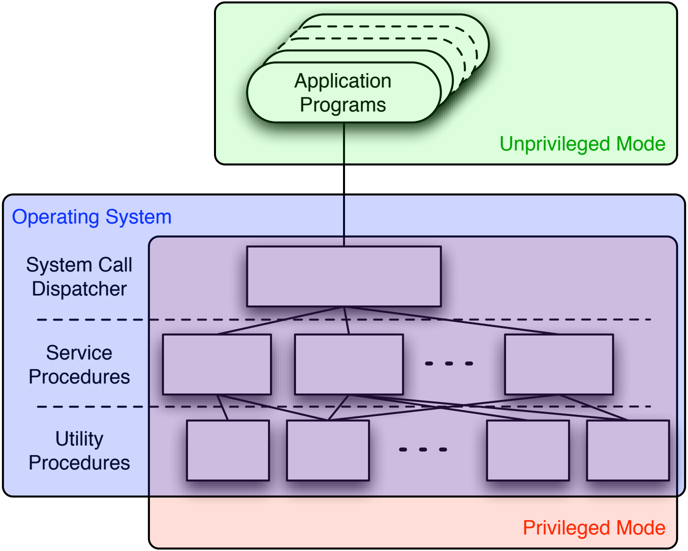
Monolithic Systems (OS/360)

- The operating system is written as a collection of procedures. Each procedure has a well-defined interface, in terms of parameters and results, but each one is free to call any other one.
- To build the operating system executable image, all the procedures are first compiled individually and then bound together by the linker.
- There is essentially no information hiding, because every procedure is visible to every other procedure.
- It is possible to have a little structure by informally dividing the procedures into three groups, or levels:
  - A main procedure, that intercepts the system calls and invokes the requested service procedure.
  - A set of service procedures, that carry out the system calls.
  - A set of utility procedures, that help the service procedures by performing commonly-needed functions.

Monolithic Systems (OS/360)

- The operating system is written as a collection of procedures. Each procedure has a well-defined interface, in terms of parameters and results, but each one is free to call any other one.
- To build the operating system executable image, all the procedures are first compiled individually and then bound together by the linker.
- There is essentially no information hiding, because every procedure is visible to every other procedure.
- It is possible to have a little structure by informally dividing the procedures into three groups, or levels:
  - A main procedure, that intercepts the system calls and invokes the requested service procedure.
  - A set of service procedures, that carry out the system calls.
  - A set of utility procedures, that help the service procedures by performing commonly-needed functions.

Structuring Model for a Monolithic System



Layered Systems (Unix)

The operating system is organized as a hierarchy of layers at the design level.

- The code is more modular and each layer can be designed and implemented independently from the other ones, possibly by different groups of programmers.
- The implementation of each layer may be changed without modifying the other layers and without even knowing their internal structure in detail.

Layered Systems (Unix)

The operating system is organized as a hierarchy of layers at the design level.

- The code is more modular and each layer can be designed and implemented independently from the other ones, possibly by different groups of programmers.
- The implementation of each layer may be changed without modifying the other layers and without even knowing their internal structure in detail.

Layered Systems (Unix)

The operating system is organized as a hierarchy of layers at the design level.

- The code is more modular and each layer can be designed and implemented independently from the other ones, possibly by different groups of programmers.
- The implementation of each layer may be changed without modifying the other layers and without even knowing their internal structure in detail.

Shortcomings of Monolithic and Layered Systems

- All operating system components are executed in privileged mode, ...
- ...even if this is not strictly necessary.
- For example, the operating system module that implements the file system could do its job in unprivileged mode as well.
- Hence, there is not any mutual protection among operating system modules and an error in one of them can easily corrupt the other ones.

Shortcomings of Monolithic and Layered Systems

- All operating system components are executed in privileged mode, ...
- ...even if this is not strictly necessary.
- For example, the operating system module that implements the file system could do its job in unprivileged mode as well.
- Hence, there is not any mutual protection among operating system modules and an error in one of them can easily corrupt the other ones.

Shortcomings of Monolithic and Layered Systems

- All operating system components are executed in privileged mode, ...
- ...even if this is not strictly necessary.
- For example, the operating system module that implements the file system could do its job in unprivileged mode as well.
- Hence, there is not any mutual protection among operating system modules and an error in one of them can easily corrupt the other ones.

Shortcomings of Monolithic and Layered Systems

- All operating system components are executed in privileged mode, . . .
- . . . even if this is not strictly necessary.
- For example, the operating system module that implements the file system could do its job in unprivileged mode as well.
- Hence, there is not any mutual protection among operating system modules and an error in one of them can easily corrupt the other ones.

Further Generalization of Layering (Multics)

In MULTICS, the layering concept was not only a design aid, but the hardware enforced it at runtime, too.

- Instead of layers, the MULTICS operating system had a series of concentric rings. The inner rings were more privileged than outer ones.
- When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to perform the equivalent of a system call, whose parameters were carefully checked for validity before proceeding.
- Although all rings were mapped into the address space of each MULTICS process, the hardware made it possible to mark each memory segment as protected against read, write or execution.

Further Generalization of Layering (Multics)

In MULTICS, the layering concept was not only a design aid, but the hardware enforced it at runtime, too.

- Instead of layers, the MULTICS operating system had a series of concentric rings. The inner rings were more privileged than outer ones.
- When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to perform the equivalent of a system call, whose parameters were carefully checked for validity before proceeding.
- Although all rings were mapped into the address space of each MULTICS process, the hardware made it possible to mark each memory segment as protected against read, write or execution.

Further Generalization of Layering (Multics)

In MULTICS, the layering concept was not only a design aid, but the hardware enforced it at runtime, too.

- Instead of layers, the MULTICS operating system had a series of concentric rings. The inner rings were more privileged than outer ones.
- When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to perform the equivalent of a system call, whose parameters were carefully checked for validity before proceeding.
- Although all rings were mapped into the address space of each MULTICS process, the hardware made it possible to mark each memory segment as protected against read, write or execution.

Further Generalization of Layering (Multics)

In MULTICS, the layering concept was not only a design aid, but the hardware enforced it at runtime, too.

- Instead of layers, the MULTICS operating system had a series of concentric rings. The inner rings were more privileged than outer ones.
- When a procedure in an outer ring wanted to call a procedure in an inner ring, it had to perform the equivalent of a system call, whose parameters were carefully checked for validity before proceeding.
- Although all rings were mapped into the address space of each MULTICS process, the hardware made it possible to mark each memory segment as protected against read, write or execution.

Microkernel Systems (MacOS)

Remove as much operating system code as possible from privileged mode and move it into a set of unprivileged system processes, leaving a minimal microkernel.

- The system processes are independent of each other and cooperate to realize the operating system functions.
- The only purpose of the microkernel is to handle the communications among user and system processes.
- By splitting the operating system up into separate processes, each part becomes smaller and more manageable.
- Being unprivileged, the system processes can be insulated and protected from each other. Hence, a bug in one of them will not bring the whole machine down.

Microkernel Systems (MacOS)

Remove as much operating system code as possible from privileged mode and move it into a set of unprivileged system processes, leaving a minimal microkernel.

- The system processes are independent of each other and cooperate to realize the operating system functions.
- The only purpose of the microkernel is to handle the communications among user and system processes.
- By splitting the operating system up into separate processes, each part becomes smaller and more manageable.
- Being unprivileged, the system processes can be insulated and protected from each other. Hence, a bug in one of them will not bring the whole machine down.

Microkernel Systems (MacOS)

Remove as much operating system code as possible from privileged mode and move it into a set of unprivileged system processes, leaving a minimal microkernel.

- The system processes are independent of each other and cooperate to realize the operating system functions.
- The only purpose of the microkernel is to handle the communications among user and system processes.
- By splitting the operating system up into separate processes, each part becomes smaller and more manageable.
- Being unprivileged, the system processes can be insulated and protected from each other. Hence, a bug in one of them will not bring the whole machine down.

Microkernel Systems (MacOS)

Remove as much operating system code as possible from privileged mode and move it into a set of unprivileged system processes, leaving a minimal microkernel.

- The system processes are independent of each other and cooperate to realize the operating system functions.
- The only purpose of the microkernel is to handle the communications among user and system processes.
- By splitting the operating system up into separate processes, each part becomes smaller and more manageable.
- Being unprivileged, the system processes can be insulated and protected from each other. Hence, a bug in one of them will not bring the whole machine down.

Microkernel Systems (MacOS)

Remove as much operating system code as possible from privileged mode and move it into a set of unprivileged system processes, leaving a minimal microkernel.

- The system processes are independent of each other and cooperate to realize the operating system functions.
- The only purpose of the microkernel is to handle the communications among user and system processes.
- By splitting the operating system up into separate processes, each part becomes smaller and more manageable.
- Being unprivileged, the system processes can be insulated and protected from each other. Hence, a bug in one of them will not bring the whole machine down.

Microkernels and I/O

Some operating system functions, for example interacting with physical I/O devices, are difficult or impossible to do from an unprivileged process, because they inherently require the execution of privileged instructions.

There are two ways to deal with this problem:

1. Make some system processes run in privileged mode, so that they have complete access to the hardware, but still communicate with other processes as usual, in order to keep the system structure intact.
2. Add to the microkernel a minimal mechanism for this. For example, the microkernel might recognize that a message sent to a "special" destination actually is a request to load its contents into the I/O registers of some device. Policy decisions are left to the requesting process, hence the kernel does not check the message to see if it is meaningful.

Microkernels and I/O

Some operating system functions, for example interacting with physical I/O devices, are difficult or impossible to do from an unprivileged process, because they inherently require the execution of privileged instructions.

There are two ways to deal with this problem:

1. Make some system processes run in privileged mode, so that they have complete access to the hardware, but still communicate with other processes as usual, in order to keep the system structure intact.
2. Add to the microkernel a minimal mechanism for this. For example, the microkernel might recognize that a message sent to a "special" destination actually is a request to load its contents into the I/O registers of some device. Policy decisions are left to the requesting process, hence the kernel does not check the message to see if it is meaningful.

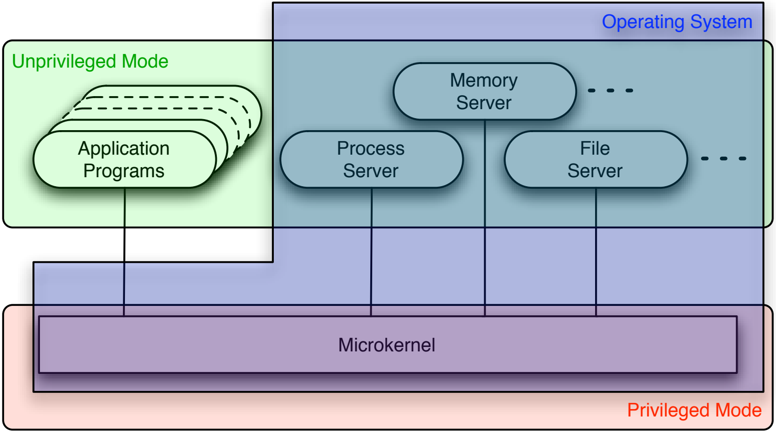
Microkernels and I/O

Some operating system functions, for example interacting with physical I/O devices, are difficult or impossible to do from an unprivileged process, because they inherently require the execution of privileged instructions.

There are two ways to deal with this problem:

1. Make some system processes run in privileged mode, so that they have complete access to the hardware, but still communicate with other processes as usual, in order to keep the system structure intact.
2. Add to the microkernel a minimal mechanism for this. For example, the microkernel might recognize that a message sent to a “special” destination actually is a request to load its contents into the I/O registers of some device. Policy decisions are left to the requesting process, hence the kernel does not check the message to see if it is meaningful.

Structure of a Microkernel-Based System



Microkernel Pros and Cons

+ Better modularity: the operating system is split up into simpler modules that communicate in a well-defined way.
+ Greater reliability: if a system process crashes or must be replaced, it can be stopped and restarted without rebooting the whole machine.
+ The model can be used in distributed systems: the communicating processes may very well be unaware that their messages are being transported across a network.
– It is difficult to implement it efficiently, even if some progress has been made recently (Liedtke).

Microkernel Pros and Cons

+ Better modularity: the operating system is split up into simpler modules that communicate in a well-defined way.
+ Greater reliability: if a system process crashes or must be replaced, it can be stopped and restarted without rebooting the whole machine.
+ The model can be used in distributed systems: the communicating processes may very well be unaware that their messages are being transported across a network.
– It is difficult to implement it efficiently, even if some progress has been made recently (Liedtke).

Microkernel Pros and Cons

+ Better modularity: the operating system is split up into simpler modules that communicate in a well-defined way.
+ Greater reliability: if a system process crashes or must be replaced, it can be stopped and restarted without rebooting the whole machine.
+ The model can be used in distributed systems: the communicating processes may very well be unaware that their messages are being transported across a network.
– It is difficult to implement it efficiently, even if some progress has been made recently (Liedtke).

Microkernel Pros and Cons

+ Better modularity: the operating system is split up into simpler modules that communicate in a well-defined way.
+ Greater reliability: if a system process crashes or must be replaced, it can be stopped and restarted without rebooting the whole machine.
+ The model can be used in distributed systems: the communicating processes may very well be unaware that their messages are being transported across a network.
– It is difficult to implement it efficiently, even if some progress has been made recently (Liedtke).

Virtual Machines (VM/370)

Observation

A timesharing system provides:

1. multiprogramming;
2. an extended machine.

VM/370 completely separates these two functions:

1. A virtual machine monitor runs on the bare hardware and provides (through multiprogramming) several virtual machines, that are exact copies of the bare hardware.
2. Each virtual machine can run any operating system that will run directly on the bare hardware. Different virtual machines can run different guest operating systems.

Virtual Machines (VM/370)

Observation

A timesharing system provides:

1. multiprogramming;
2. an extended machine.

VM/370 completely separates these two functions:

1. A virtual machine monitor runs on the bare hardware and provides (through multiprogramming) several virtual machines, that are exact copies of the bare hardware.
2. Each virtual machine can run any operating system that will run directly on the bare hardware. Different virtual machines can run different guest operating systems.

Virtual Machines (VM/370)

Observation

A timesharing system provides:

1. multiprogramming;
2. an extended machine.

VM/370 completely separates these two functions:

1. A virtual machine monitor runs on the bare hardware and provides (through multiprogramming) several virtual machines, that are exact copies of the bare hardware.
2. Each virtual machine can run any operating system that will run directly on the bare hardware. Different virtual machines can run different guest operating systems.

How does it work?

- The virtual machine monitor is completely transparent to the guest operating systems, except for timing.
- The privilege level of the physical CPU is kept completely separated and independent from the privilege level of each virtual machine.
- The guest operating systems run in virtually privileged mode on a CPU that runs in a physically unprivileged mode.
- Part of the job of the virtual machine monitor is to simulate the real hardware for what concerns the execution of privileged instructions.

How does it work?

- The virtual machine monitor is completely transparent to the guest operating systems, except for timing.
- The privilege level of the physical CPU is kept completely separated and independent from the privilege level of each virtual machine.
- The guest operating systems run in virtually privileged mode on a CPU that runs in a physically unprivileged mode.
- Part of the job of the virtual machine monitor is to simulate the real hardware for what concerns the execution of privileged instructions.

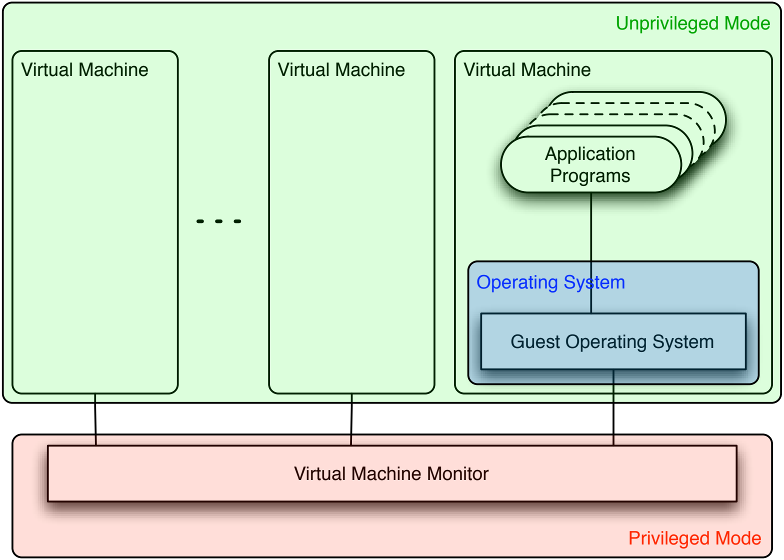
How does it work?

- The virtual machine monitor is completely transparent to the guest operating systems, except for timing.
- The privilege level of the physical CPU is kept completely separated and independent from the privilege level of each virtual machine.
- The guest operating systems run in virtually privileged mode on a CPU that runs in a physically unprivileged mode.
- Part of the job of the virtual machine monitor is to simulate the real hardware for what concerns the execution of privileged instructions.

How does it work?

- The virtual machine monitor is completely transparent to the guest operating systems, except for timing.
- The privilege level of the physical CPU is kept completely separated and independent from the privilege level of each virtual machine.
- The guest operating systems run in virtually privileged mode on a CPU that runs in a physically unprivileged mode.
- Part of the job of the virtual machine monitor is to simulate the real hardware for what concerns the execution of privileged instructions.

Structure of a VM-based system



VM Pros and Cons

- \+ The code being executed inside a VM is completely insulated from the other VMs, and is always executed in unprivileged mode, including the guest operating system.
- \+ Different operating systems can run unmodified inside their own VM, and be executed concurrently on the same physical machine.
- \+ All privileged actions taken by the guest operating systems are intercepted by the virtual machine monitor so that it can check them for validity beforehand.
- – Simulating a number of VMs is complex and incurs a high overhead, unless appropriate hardware assistance is available.

VM Pros and Cons

+ The code being executed inside a VM is completely insulated from the other VMs, and is always executed in unprivileged mode, including the guest operating system.
+ Different operating systems can run unmodified inside their own VM, and be executed concurrently on the same physical machine.
+ All privileged actions taken by the guest operating systems are intercepted by the virtual machine monitor so that it can check them for validity beforehand.
– Simulating a number of VMs is complex and incurs a high overhead, unless appropriate hardware assistance is available.

VM Pros and Cons

+ The code being executed inside a VM is completely insulated from the other VMs, and is always executed in unprivileged mode, including the guest operating system.
+ Different operating systems can run unmodified inside their own VM, and be executed concurrently on the same physical machine.
+ All privileged actions taken by the guest operating systems are intercepted by the virtual machine monitor so that it can check them for validity beforehand.

– Simulating a number of VMs is complex and incurs a high overhead, unless appropriate hardware assistance is available.

VM Pros and Cons

- \+ The code being executed inside a VM is completely insulated from the other VMs, and is always executed in unprivileged mode, including the guest operating system.
- \+ Different operating systems can run unmodified inside their own VM, and be executed concurrently on the same physical machine.
- \+ All privileged actions taken by the guest operating systems are intercepted by the virtual machine monitor so that it can check them for validity beforehand.
- – Simulating a number of VMs is complex and incurs a high overhead, unless appropriate hardware assistance is available.

What is a System Call?

The set of system calls that the operating system provides defines the interface between an operating system and the user programs.

- With a system call, a process transfers control to the operating system to request a service.
- System calls are usually realized by means of a trap or a system call instruction.
- In some real-time operating systems, the mechanism is simpler and uses a normal call instead:
  - + greater efficiency;
  - − less protection.

What is a System Call?

The set of system calls that the operating system provides defines the interface between an operating system and the user programs.

- With a system call, a process transfers control to the operating system to request a service.
- System calls are usually realized by means of a trap or a system call instruction.
- In some real-time operating systems, the mechanism is simpler and uses a normal call instead:
  - + greater efficiency;
  - − less protection.

What is a System Call?

The set of system calls that the operating system provides defines the interface between an operating system and the user programs.

- With a system call, a process transfers control to the operating system to request a service.
- System calls are usually realized by means of a trap or a system call instruction.
- In some real-time operating systems, the mechanism is simpler and uses a normal call instead:
  - + greater efficiency;
  - − less protection.

What is a System Call?

The set of system calls that the operating system provides defines the interface between an operating system and the user programs.

- With a system call, a process transfers control to the operating system to request a service.
- System calls are usually realized by means of a trap or a system call instruction.
- In some real-time operating systems, the mechanism is simpler and uses a normal call instead:
  - \+ greater efficiency;
  - – less protection.

What is a System Call?

The set of system calls that the operating system provides defines the interface between an operating system and the user programs.

- With a system call, a process transfers control to the operating system to request a service.
- System calls are usually realized by means of a trap or a system call instruction.
- In some real-time operating systems, the mechanism is simpler and uses a normal call instead:
  + greater efficiency;
  – less protection.

What is a System Call?

The set of system calls that the operating system provides defines the interface between an operating system and the user programs.

- With a system call, a process transfers control to the operating system to request a service.
- System calls are usually realized by means of a trap or a system call instruction.
- In some real-time operating systems, the mechanism is simpler and uses a normal call instead:
  - + greater efficiency;
  - – less protection.

Groups of System Calls (I)

Process Management

- Create a new process.
- Terminate the invoking process.
- Wait for another process to terminate.
- Low-level, dynamic memory allocation.

Signal Management

- Get and set the action associated with an asynchronous signal.
- Send a signal to oneself or to another process.

Groups of System Calls (I)

Process Management

- Create a new process.
- Terminate the invoking process.
- Wait for another process to terminate.
- Low-level, dynamic memory allocation.

Signal Management

- Get and set the action associated with an asynchronous signal.
- Send a signal to oneself or to another process.

Groups of System Calls (II)

File Management

- Create, open, read, write, close, . . . a file.
- Get and set file characteristics and attributes.

Directory and File System Management

- Create and remove a directory.
- Mount and unmount a file system.

Miscellaneous

- Change the current working directory.
- Get and set the system time and date.

Groups of System Calls (II)

File Management

- Create, open, read, write, close, ... a file.
- Get and set file characteristics and attributes.

Directory and File System Management

- Create and remove a directory.
- Mount and unmount a file system.

Miscellaneous

- Change the current working directory.
- Get and set the system time and date.

Groups of System Calls (II)

File Management

- Create, open, read, write, close, . . . a file.
- Get and set file characteristics and attributes.

Directory and File System Management

- Create and remove a directory.
- Mount and unmount a file system.

Miscellaneous

- Change the current working directory.
- Get and set the system time and date.

System Calls and Programming Languages

- Most programming languages, by themselves, know nothing about system calls.
- System calls are performed by invoking an appropriate library procedure that, in turn, makes the system call.
- The library procedure, often written in assembly language, takes the system call parameters from the location into which the compiler placed them for the function call (usually the stack), puts them into the place where the operating system expects them for the system call, and executes the system call instruction.
- When the system call ends, it does the reverse for the system call return value(s), to make the system call indistinguishable from a regular function call.

System Calls and Programming Languages

- Most programming languages, by themselves, know nothing about system calls.
- System calls are performed by invoking an appropriate library procedure that, in turn, makes the system call.
- The library procedure, often written in assembly language, takes the system call parameters from the location into which the compiler placed them for the function call (usually the stack), puts them into the place where the operating system expects them for the system call, and executes the system call instruction.
- When the system call ends, it does the reverse for the system call return value(s), to make the system call indistinguishable from a regular function call.

System Calls and Programming Languages

- Most programming languages, by themselves, know nothing about system calls.
- System calls are performed by invoking an appropriate library procedure that, in turn, makes the system call.
- The library procedure, often written in assembly language, takes the system call parameters from the location into which the compiler placed them for the function call (usually the stack), puts them into the place where the operating system expects them for the system call, and executes the system call instruction.
- When the system call ends, it does the reverse for the system call return value(s), to make the system call indistinguishable from a regular function call.

System Calls and Programming Languages

- Most programming languages, by themselves, know nothing about system calls.
- System calls are performed by invoking an appropriate library procedure that, in turn, makes the system call.
- The library procedure, often written in assembly language, takes the system call parameters from the location into which the compiler placed them for the function call (usually the stack), puts them into the place where the operating system expects them for the system call, and executes the system call instruction.
- When the system call ends, it does the reverse for the system call return value(s), to make the system call indistinguishable from a regular function call.

System Calls on Layered Systems

- The user process puts the system call identifier and parameters into a well-defined location (e.g. a set of registers or the stack) and executes a trap.
- The trap switches the processor from unprivileged to privileged mode and starts executing within then kernel at the trap handler address, where there the system call dispatcher resides.
- The system call dispatcher saves the unprivileged execution context, determines which system call has been requested, and dispatches to the right system call handler.
- Once the system call handler has completed its work, control may be returned to the invoking process at the instruction that follows the trap instruction, by restoring the unprivileged execution context previously saved.

System Calls on Layered Systems

- The user process puts the system call identifier and parameters into a well-defined location (e.g. a set of registers or the stack) and executes a trap.
- The trap switches the processor from unprivileged to privileged mode and starts executing within then kernel at the trap handler address, where there the system call dispatcher resides.
- The system call dispatcher saves the unprivileged execution context, determines which system call has been requested, and dispatches to the right system call handler.
- Once the system call handler has completed its work, control may be returned to the invoking process at the instruction that follows the trap instruction, by restoring the unprivileged execution context previously saved.

System Calls on Layered Systems

- The user process puts the system call identifier and parameters into a well-defined location (e.g. a set of registers or the stack) and executes a trap.
- The trap switches the processor from unprivileged to privileged mode and starts executing within then kernel at the trap handler address, where there the system call dispatcher resides.
- The system call dispatcher saves the unprivileged execution context, determines which system call has been requested, and dispatches to the right system call handler.
- Once the system call handler has completed its work, control may be returned to the invoking process at the instruction that follows the trap instruction, by restoring the unprivileged execution context previously saved.

System Calls on Layered Systems

- The user process puts the system call identifier and parameters into a well-defined location (e.g. a set of registers or the stack) and executes a trap.
- The trap switches the processor from unprivileged to privileged mode and starts executing within then kernel at the trap handler address, where there the system call dispatcher resides.
- The system call dispatcher saves the unprivileged execution context, determines which system call has been requested, and dispatches to the right system call handler.
- Once the system call handler has completed its work, control may be returned to the invoking process at the instruction that follows the trap instruction, by restoring the unprivileged execution context previously saved.

Interrupts on Layered Systems (I)

- The interrupt request, when accepted, switches the processor from unprivileged to privileged mode and starts executing the corresponding interrupt handler within the kernel.
- The interrupt handler saves the current execution context (either unprivileged or privileged), identifies the interrupt source and handles the interrupt.
- Then, the operating system resumes the execution of a process, either in privileged or unprivileged mode, by restoring an execution context.
- If the system supports nested interrupts, the interrupted execution contexts are saved into the interrupt stack.

Interrupts on Layered Systems (I)

- The interrupt request, when accepted, switches the processor from unprivileged to privileged mode and starts executing the corresponding interrupt handler within the kernel.
- The interrupt handler saves the current execution context (either unprivileged or privileged), identifies the interrupt source and handles the interrupt.
- Then, the operating system resumes the execution of a process, either in privileged or unprivileged mode, by restoring an execution context.
- If the system supports nested interrupts, the interrupted execution contexts are saved into the interrupt stack.

Interrupts on Layered Systems (I)

- The interrupt request, when accepted, switches the processor from unprivileged to privileged mode and starts executing the corresponding interrupt handler within the kernel.
- The interrupt handler saves the current execution context (either unprivileged or privileged), identifies the interrupt source and handles the interrupt.
- Then, the operating system resumes the execution of a process, either in privileged or unprivileged mode, by restoring an execution context.
- If the system supports nested interrupts, the interrupted execution contexts are saved into the interrupt stack.

Interrupts on Layered Systems (I)

- The interrupt request, when accepted, switches the processor from unprivileged to privileged mode and starts executing the corresponding interrupt handler within the kernel.
- The interrupt handler saves the current execution context (either unprivileged or privileged), identifies the interrupt source and handles the interrupt.
- Then, the operating system resumes the execution of a process, either in privileged or unprivileged mode, by restoring an execution context.
- If the system supports nested interrupts, the interrupted execution contexts are saved into the interrupt stack.

Interrupts on Layered Systems (II)

- Interrupt handlers implicitly have a priority that is higher than the priority of any other activity in the system.
- To ensure the atomicity of a sequence of instructions, interrupts can be disabled temporarily, but doing this increases the interrupt response latency.
- Unlike processes, interrupt handlers do not have their own fixed location to hold their execution context, so they are more limited than processes in the interaction with the system.

Interrupts on Layered Systems (II)

- Interrupt handlers implicitly have a priority that is higher than the priority of any other activity in the system.
- To ensure the atomicity of a sequence of instructions, interrupts can be disabled temporarily, but doing this increases the interrupt response latency.
- Unlike processes, interrupt handlers do not have their own fixed location to hold their execution context, so they are more limited than processes in the interaction with the system.

Interrupts on Layered Systems (II)

- Interrupt handlers implicitly have a priority that is higher than the priority of any other activity in the system.
- To ensure the atomicity of a sequence of instructions, interrupts can be disabled temporarily, but doing this increases the interrupt response latency.
- Unlike processes, interrupt handlers do not have their own fixed location to hold their execution context, so they are more limited than processes in the interaction with the system.

System Calls on Microkernels

- The user process builds a message containing the system call parameters. Then, it asks the microkernel to transfer the message to the appropriate system process by means of a trap and waits for an answer.
- The microkernel sends the message to the system process, awakening it if necessary.
- When the system process runs, it executes the function it has been asked for, builds a message containing the results, and asks the microkernel to send it back to the user process.
- The user process receives the answer and continues its execution.
- While doing its job, the system process may need to communicate with other system processes; the mechanism is the same.

System Calls on Microkernels

- The user process builds a message containing the system call parameters. Then, it asks the microkernel to transfer the message to the appropriate system process by means of a trap and waits for an answer.
- The microkernel sends the message to the system process, awakening it if necessary.
- When the system process runs, it executes the function it has been asked for, builds a message containing the results, and asks the microkernel to send it back to the user process.
- The user process receives the answer and continues its execution.
- While doing its job, the system process may need to communicate with other system processes; the mechanism is the same.

System Calls on Microkernels

- The user process builds a message containing the system call parameters. Then, it asks the microkernel to transfer the message to the appropriate system process by means of a trap and waits for an answer.
- The microkernel sends the message to the system process, awakening it if necessary.
- When the system process runs, it executes the function it has been asked for, builds a message containing the results, and asks the microkernel to send it back to the user process.
- The user process receives the answer and continues its execution.
- While doing its job, the system process may need to communicate with other system processes; the mechanism is the same.

System Calls on Microkernels

- The user process builds a message containing the system call parameters. Then, it asks the microkernel to transfer the message to the appropriate system process by means of a trap and waits for an answer.
- The microkernel sends the message to the system process, awakening it if necessary.
- When the system process runs, it executes the function it has been asked for, builds a message containing the results, and asks the microkernel to send it back to the user process.
- The user process receives the answer and continues its execution.
- While doing its job, the system process may need to communicate with other system processes; the mechanism is the same.

System Calls on Microkernels

- The user process builds a message containing the system call parameters. Then, it asks the microkernel to transfer the message to the appropriate system process by means of a trap and waits for an answer.
- The microkernel sends the message to the system process, awakening it if necessary.
- When the system process runs, it executes the function it has been asked for, builds a message containing the results, and asks the microkernel to send it back to the user process.
- The user process receives the answer and continues its execution.
- While doing its job, the system process may need to communicate with other system processes; the mechanism is the same.

Interrupt Handling on Microkernels

- The interrupt handler performs the minimum amount of work needed to keep the hardware running (e.g. resetting the interrupt controller), then converts the interrupt into a message directed to the system process responsible for handling it.
- When that system process runs, it performs the actual interrupt handling activities (e.g. interacting with the interrupting device). This may require several interactions with other system or user processes.
- The system processes devoted to interrupt handling are full-fledged processes.

Interrupt Handling on Microkernels

- The interrupt handler performs the minimum amount of work needed to keep the hardware running (e.g. resetting the interrupt controller), then converts the interrupt into a message directed to the system process responsible for handling it.
- When that system process runs, it performs the actual interrupt handling activities (e.g. interacting with the interrupting device). This may require several interactions with other system or user processes.
- The system processes devoted to interrupt handling are full-fledged processes.

Interrupt Handling on Microkernels

- The interrupt handler performs the minimum amount of work needed to keep the hardware running (e.g. resetting the interrupt controller), then converts the interrupt into a message directed to the system process responsible for handling it.
- When that system process runs, it performs the actual interrupt handling activities (e.g. interacting with the interrupting device). This may require several interactions with other system or user processes.
- The system processes devoted to interrupt handling are full-fledged processes.

System Calls on VMs

- When a system call is executed inside a VM, the trap activates the virtual machine monitor instead of the guest operating system.
- The virtual machine monitor redirects it to the guest operating system after switching the processor back to unprivileged mode.
- All privileged instructions issued by the guest operating system will then raise a privileged instruction trap.
- Again, they will activate the virtual machine monitor that will perform them, after checking their legality, as part of its simulation of the real hardware.
- This mechanism supports the implementation of virtual I/O devices and the controlled communication among virtual machines.

System Calls on VMs

- When a system call is executed inside a VM, the trap activates the virtual machine monitor instead of the guest operating system.
- The virtual machine monitor redirects it to the guest operating system after switching the processor back to unprivileged mode.
- All privileged instructions issued by the guest operating system will then raise a privileged instruction trap.
- Again, they will activate the virtual machine monitor that will perform them, after checking their legality, as part of its simulation of the real hardware.
- This mechanism supports the implementation of virtual I/O devices and the controlled communication among virtual machines.

System Calls on VMs

- When a system call is executed inside a VM, the trap activates the virtual machine monitor instead of the guest operating system.
- The virtual machine monitor redirects it to the guest operating system after switching the processor back to unprivileged mode.
- All privileged instructions issued by the guest operating system will then raise a privileged instruction trap.
- Again, they will activate the virtual machine monitor that will perform them, after checking their legality, as part of its simulation of the real hardware.
- This mechanism supports the implementation of virtual I/O devices and the controlled communication among virtual machines.

System Calls on VMs

- When a system call is executed inside a VM, the trap activates the virtual machine monitor instead of the guest operating system.
- The virtual machine monitor redirects it to the guest operating system after switching the processor back to unprivileged mode.
- All privileged instructions issued by the guest operating system will then raise a privileged instruction trap.
- Again, they will activate the virtual machine monitor that will perform them, after checking their legality, as part of its simulation of the real hardware.
- This mechanism supports the implementation of virtual I/O devices and the controlled communication among virtual machines.

System Calls on VMs

- When a system call is executed inside a VM, the trap activates the virtual machine monitor instead of the guest operating system.
- The virtual machine monitor redirects it to the guest operating system after switching the processor back to unprivileged mode.
- All privileged instructions issued by the guest operating system will then raise a privileged instruction trap.
- Again, they will activate the virtual machine monitor that will perform them, after checking their legality, as part of its simulation of the real hardware.
- This mechanism supports the implementation of virtual I/O devices and the controlled communication among virtual machines.

Interrupts on VMs

- All interrupts trigger the execution of the corresponding interrupt handler within the virtual machine monitor.
- Depending on the source of the interrupt, the virtual machine monitor may redirect the interrupt towards the virtual machine that owns that device.
- Also in this case, the processor is switched back to unprivileged mode, so that the interactions between the guest interrupt handler and the device can be inspected by the virtual machine monitor beforehand.

Interrupts on VMs

- All interrupts trigger the execution of the corresponding interrupt handler within the virtual machine monitor.
- Depending on the source of the interrupt, the virtual machine monitor may redirect the interrupt towards the virtual machine that owns that device.
- Also in this case, the processor is switched back to unprivileged mode, so that the interactions between the guest interrupt handler and the device can be inspected by the virtual machine monitor beforehand.

Interrupts on VMs

- All interrupts trigger the execution of the corresponding interrupt handler within the virtual machine monitor.
- Depending on the source of the interrupt, the virtual machine monitor may redirect the interrupt towards the virtual machine that owns that device.
- Also in this case, the processor is switched back to unprivileged mode, so that the interactions between the guest interrupt handler and the device can be inspected by the virtual machine monitor beforehand.