
CSPC61, EMBEDDED SYSTEMS AND ARCHITECTURE
CHAPTER-8: DEVICE DRIVERS

---

1. What is a device driver?

Device drivers are the software libraries that initialize the hardware and manage access to the hardware by higher layers of software. It directly interfaces with and controls the hardware. Device drivers are the liaison between the hardware and the operating system, middleware, and application layers.

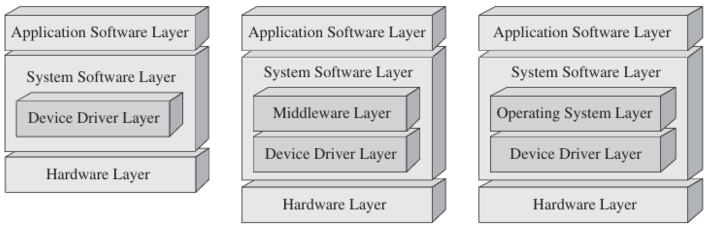


Figure 8-1
Embedded Systems Model and Device Drivers.

---

2. Which of Figures 8-46a, b, c, and d is incorrect in terms of mapping device driver software into the Embedded Systems Model? **Figure 8-46b**

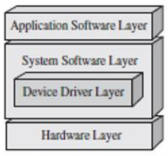


Figure 8-46a: Example 1

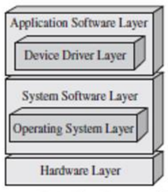


Figure 8-46b: Example 2

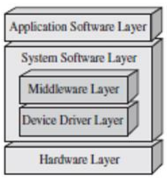


Figure 8-46c: Example 3

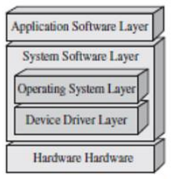


Figure 8-46d: Example 4

---

3. What is the difference between an architecture-specific device driver and a generic device driver? Give two examples of each.

Parameter	*Architecture-Specific*	*Generic*
Functionality	Manages hardware integrated into the master processor (the architecture).	Manages hardware located on the board and not integrated onto the master processor.
Code Structure	Primarily involves low-level hardware access functions optimized for the specific processor architecture.	Typically includes architecture-specific portions of source code, as the master processor serves as the central control unit to access anything on the board.
Versatility	Limited to the architecture it's designed for.	Can be configured to run on a variety of architectures that contain the related board hardware for which the driver is written.
Examples	• On-Chip memory • Integrated memory managers (MMUs) • Floating-point hardware	• Board buses (I2C, PCI, PCMCIA) • Off-chip memory (controllers, cache, Flash, etc) • Off-chip I/O (Ethernet, RS-232, display, mouse, etc)

---

5. List and describe five types of device driver functions.

   i. *Hardware Startup*: Initialization of the hardware upon Power-ON or reset.
   ii. *Hardware Shutdown*: Configuring hardware into its Power-OFF state.
   iii. *Hardware Disable*: Allowing other software to disable hardware on-the-fly.
   iv. *Hardware Enable*: Allowing other software to enable hardware on-the-fly.
   v. *Hardware Acquire*: Allowing other software to gain singular (locking) access to hardware.

---

6. Finish the sentence: The software's implicit perception of hardware is that it exists in one of three states at any given time:

inactive, finished, or busy / inactive, finished, or broken / fixed, finished, or busy / fixed, inactive, or broken / None of the above.

---

7. [T/F] On master processors that offer different modes in which different types of software can execute, device drivers usually do not run in supervisory mode. **False**.

Depending on the master processor, different types of software can execute in different modes, the most common being supervisory and user modes. These modes essentially differ in terms of what system components the software is allowed access to, with software running in supervisory mode having more access (privileges) than software running in user mode. **Device driver code typically runs in supervisory mode.**

---

8. What is an interrupt? How can interrupts be initiated?

Interrupts are signals triggered by some event during the execution of an instruction stream by the master processor.

Interrupts can be initiated asynchronously, for external hardware devices, resets, power failures, etc., or synchronously, for instruction-related activities such as system calls or illegal instructions. These signals cause the master processor to stop executing the current instruction stream and start the process of handling (processing) the interrupt.

---

9. Name and describe four examples of device driver functions that can be implemented for interrupt handling.

   i. *Interrupt Handling Startup*: Initialization of the interrupt hardware (interrupt controller, activating interrupts, etc.) upon Power-ON or reset.
   ii. *Interrupt Handling Shutdown*: Configuring interrupt hardware (interrupt controller, deactivating interrupts, etc.) into its Power-OFF state.
   iii. *Interrupt Handling Disable*: Allowing other software to disable active interrupts on-the fly (not allowed for non-maskable interrupts (NMIs), which are interrupts that cannot be disabled).
   iv. *Interrupt Handling Enable*: Allowing other software to enable inactive interrupts on-the-fly.
   v. *Interrupt Handler Servicing*: The interrupt handling code itself, which is executed after the interruption of the main execution stream (this can range in complexity from a simple non-nested routine to nested and/or re-entrant routines).

---

10. What are the three main types of interrupts? List examples in which each type is triggered.

The three main types of interrupts are software, internal hardware, and external hardware.

   i. *Software Interrupts*: These are explicitly triggered internally by some instruction within the current instruction stream being executed by the master processor.
   ii. *Internal Hardware Interrupts*: These, on the other hand, are initiated by an event that is a result of a problem with the current instruction stream that is being executed by the master processor because of the features (or limitations) of the hardware, such as illegal math

---

---

operations (overflow, divide-by-zero), debugging (single-stepping, breakpoints), and invalid instructions (opcodes).

iii. *External Hardware Interrupts*: These are interrupts initiated by hardware other than the master CPU (board buses, I/O, etc.).

---

11. What is the difference between a level-triggered interrupt and an edge-triggered interrupt? What are some strengths and drawbacks of each?

Parameter	*Level-Triggered Interrupt*	*Edge-Triggered Interrupt*
Initiation	Initiated when IRQ signal remains at a certain level (HIGH or LOW).	Triggered by a change in the IRQ signal (LOW to HIGH or vice versa).
Processing	Processed when CPU samples IRQ line for a level-triggered request.	Latches into the CPU until processed once triggered by signal change.
Strengths	Suitable for interrupts with shared IRQ lines.	Suitable for short or long interrupt signals.
Drawbacks	May cause CPU to repeatedly service the same interrupt.	Risk of missing interrupts if triggered simultaneously on shared IRQ lines.

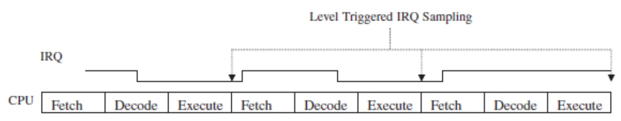


Figure 4-64a: Level-triggered interrupts [4-24]

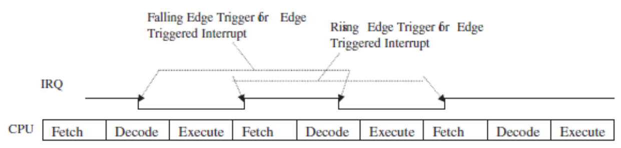

Falling Edge Trigger for Edge Triggered Interrupt
Rising Edge Trigger for Edge Triggered Interrupt
IRQ
CPU
Fetch
Decode
Execute
Fetch
Decode
Execute
Fetch
Decode
Execute

Figure 4-64b: Edge-triggered interrupts [4-24]

---

12. An IACK is: An interrupt controller / An IRQ port / **An interrupt acknowledgement /** None.

---

13. [T/F] An ISR is executed before an interrupt is triggered.

False. An ISR (Interrupt Service Routine) is executed in response to an interrupt being triggered, not before.

---

14. What is the difference between an auto-vectored and an interrupt-vectored scheme?

Parameter	*Interrupt-Vectored Scheme*	*Auto-Vectored Scheme*
Interrupt Vector	Depends on external device providing an interrupt vector.	Not applicable (Used when devices cannot provide interrupt vectors).
ISR Handling	Each device has its own interrupt vector pointing to its specific ISR.	A single ISR is shared among all non-vectored interrupts.
Responsibility	CPU directly jumps to the appropriate ISR using the interrupt vector.	ISR handles determining which specific interrupt occurred and other related tasks internally.
Implementation	Used when devices can provide interrupt vectors.	Used when devices cannot provide interrupt vectors (non-vectored interrupts).

----

15. Name and describe four examples of device driver functions that can be implemented for managing memory.

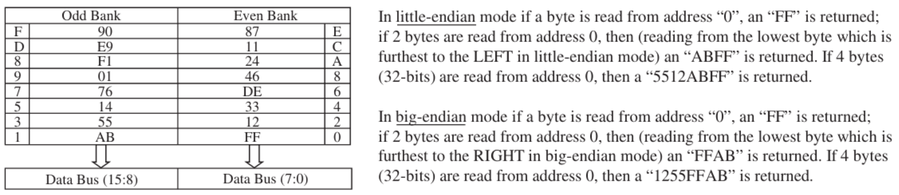
i. *Memory Subsystem Startup*: Initialization of the hardware upon Power-ON or reset (initialize translation lookaside buffers (TLBs) for MMU, initialize/configure MMU).
ii. *Memory Subsystem Shutdown*: Configuring hardware into its Power-OFF state. (Note: Under the MPC860, there is no necessary shutdown sequence for the memory subsystem, so pseudocode examples are not shown.)
iii. *Memory Subsystem Disable*: Allowing other software to disable hardware on-the-fly (disabling cache).
iv. *Memory Subsystem Enable*: Allowing other software to enable hardware on-the-fly (enable cache).
v. *Memory Subsystem Write*: Storing in memory a byte or set of bytes (i.e., in cache, ROM, and main memory).
vi. *Memory Subsystem Read*: Retrieving from memory a "copy" of the data in the form of a byte or set of bytes (i.e., in cache, ROM, and main memory).

----

16. What is byte ordering? Name and describe the possible byte ordering schemes.

The order in which bytes are retrieved or stored in memory is known as byte ordering. It depends on the byte ordering scheme of an architecture.

The two possible byte ordering schemes are little-endian and big endian. In little-endian mode, bytes (or "bits" with 1-byte (8-bit) schemes) are retrieved and stored in the order of the lowest byte first, meaning the lowest byte is furthest to the left. In big-endian mode bytes are accessed in the order of the highest byte first, meaning that the lowest byte is furthest to the right.



	Odd Bank	Even Bank	
F	90	87	E
D	E9	11	C
8	F1	24	A
9	01	46	8
7	76	DE	6
5	14	33	4
3	55	12	2
1	AB	FF	0

Data Bus (15:8)
Data Bus (7:0)

In little-endian mode if a byte is read from address "0", an "FF" is returned; if 2 bytes are read from address 0, then (reading from the lowest byte which is furthest to the LEFT in little-endian mode) an "ABFF" is returned. If 4 bytes (32-bits) are read from address 0, then a "5512ABFF" is returned.

In big-endian mode if a byte is read from address "0", an "FF" is returned; if 2 bytes are read from address 0, then (reading from the lowest byte which is furthest to the RIGHT in big-endian mode) an "FFAB" is returned. If 4 bytes (32-bits) are read from address 0, then a "1255FFAB" is returned.

----

17. Name and describe four examples of device driver functions that can be implemented for bus protocols.

i. *Bus Startup*: Initialization of the bus upon Power-ON or reset.
ii. *Bus Shutdown*: Configuring bus into its Power-OFF state.
iii. *Bus Disable*: Allowing other software to disable bus on-the-fly.
iv. *Bus Enable*: Allowing other software to enable bus on-the-fly.

----

18. Name and describe four examples of device driver functions that can be implemented for I/O.

i. *I/O Startup*: Initialization of the I/O upon Power-ON or reset.
ii. *I/O Shutdown*: Configuring I/O into its Power-OFF state.
iii. *I/O Disable*: Allowing other software to disable I/O on-the-fly.
iv. *I/O Enable*: Allowing other software to enable I/O on-the-fly.

----

19. Where in the OSI model are the Ethernet and serial device drivers mapped to?

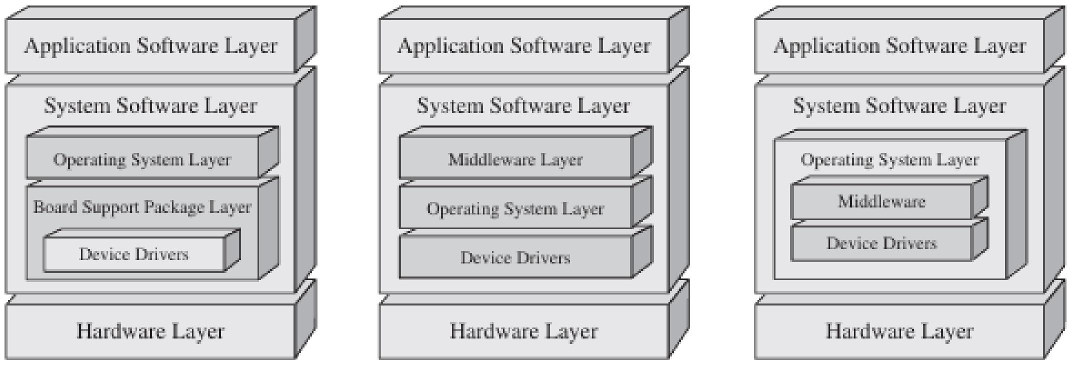
They are mapped to the lower section of the OSI (Open Systems Interconnection) data-link layer.

CSPC61, EMBEDDED SYSTEMS AND ARCHITECTURE
CHAPTER-9: EMBEDDED OPERATING SYSTEMS

---

1. What is an operating system (OS)? What does an operating system do? Draw a diagram showing where the operating system fits in the Embedded Systems Model.

The OS is a set of software libraries that serves two main purposes in an embedded system: providing an abstraction layer for software on top of the OS to be less dependent on hardware, making the development of middleware and applications that sit on top of the OS easier, and managing the various system hardware and software resources to ensure the entire system operates efficiently and reliably.



---

2. What is a kernel? Name and describe at least two functions of a kernel.

The kernel is a component that contains the main functionality of the OS, specifically all or some combination of features and their interdependencies including:

i. *Process Management*: How the OS manages & views other software in the embedded system.
ii. *Memory Management*: The embedded system's memory space is shared by all the different processes, so that access and allocation of portions of the memory space need to be managed.
iii. *I/O System Management*: I/O devices also need to be shared among the various processes and so, just as with memory, access and allocation of an I/O device need to be managed.

---

3. OSes typically fall under one of three models: **monolithic, layered, or microkernel** / monolithic, layered, or monolithic-modularized / layered, client/server, or microkernel / monolithic-modularized, client/server, or microkernel / None of the above.

---

4. Match the type of OS model to Figures 9-40a, b, and c. Name a real-world OS that falls under each model.

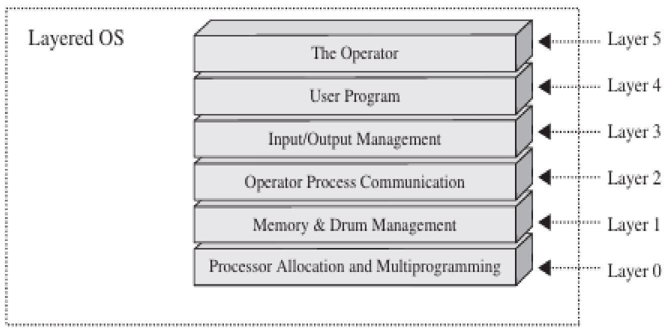


Figure 9-5
Layered OS block diagram.

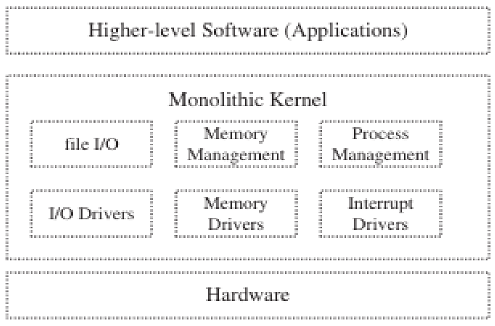


Figure 9-3
Monolithic OS block diagram.

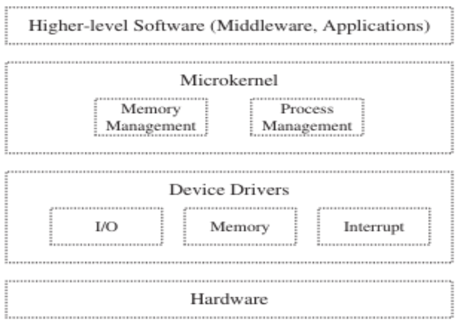


Figure 9-6
Microkernel-based OS block diagram.

i. The embedded Linux OS is an example of a monolithic-based OS. The Jbed RTOS, MicroC/OS-II, and PDOS are all examples of embedded monolithic OSs.
ii. DOS-C (FreeDOS), DOS/eRTOS, and VRTX are all examples of a layered OS.
iii. OS-9, C Executive, VxWorks, CMX-RTX, Nucleus Plus, and QNX fall under the microkernel category.

---

5. What is the difference between a process and a thread? What is the difference between a process and a task?

- *Process*: It is the instance of a program in execution, created by an OS to encapsulate all the information that is involved in the executing of a program (stack, PC, source code, data, etc.).
- *Threads*: These are lightweight processes which are an alternative means for encapsulating an instance of a program. Threads are created within the context of a task (meaning a thread is bound to a task) and, depending on the OS, the task can own one or more threads. A thread is a sequential execution stream within its task. Tasks have independent memory spaces that are inaccessible by other tasks, but threads of a task share resource but have separate PCs, stack, and scheduling information.
- *Task*: It is a set of program instructions that are loaded in memory. Tasks and processes are synonymous nowadays.

---

6. What are the most common schemes used to create tasks? Give one example of an OS that uses each of the schemes.

Task creation in embedded OSs primarily follows two models: fork/exec and spawn.

- *Fork/Exec Model:*
  - Derived from the IEEE/ISO POSIX 1003.1 standard.
  - Used in embedded Linux systems.
  - Tasks create their child tasks through fork/exec system calls.
  - "Fork" call creates a copy of the parent task's memory space for the child task, allowing inheritance of properties like program code and variables.
- *Spawn Model:*
  - Derived from fork/exec model.
  - Used in VxWorks.
  - Tasks create child tasks through spawn system calls.
  - Creates an entirely new address space for the child task.
- *Task Control Block (TCB):*
  - Created by the OS after the system call.
  - Contains control information such as task ID, state, priority, and error status.
  - Also includes CPU context information like registers for the task.

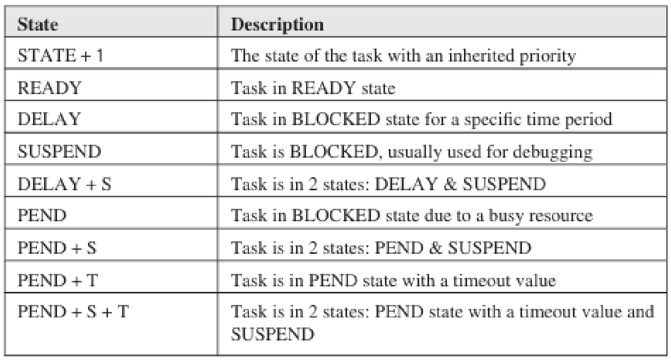
---

7. In general terms, what states can a task be in? Give one example of an OS and its available states, including the state diagrams.

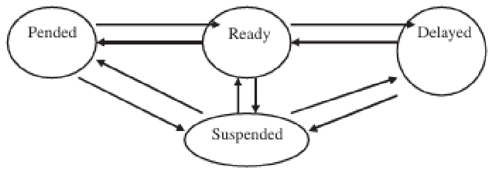
A task's state is the activity (if any) that is going on with that task once it has been created but has not been deleted. OSs usually define a task as being in one of three states:

i. *READY*: The process is ready to be executed at any time but is waiting for permission to use the CPU.
ii. *RUNNING*: The process has been given permission to use the CPU and can execute.
iii. *BLOCKED or WAITING*: The process is waiting for some external event to occur before it can be "ready" to "run."

Example: *VxWorks* - Other than the RUNNING state, VxWorks implements nine variations of the READY and BLOCKED/WAITING states, as shown in the following table and state diagram.



State	Description
STATE + 1	The state of the task with an inherited priority
READY	Task in READY state
DELAY	Task in BLOCKED state for a specific time period
SUSPEND	Task is BLOCKED, usually used for debugging
DELAY + S	Task is in 2 states: DELAY & SUSPEND
PEND	Task in BLOCKED state due to a busy resource
PEND + S	Task is in 2 states: PEND & SUSPEND
PEND + T	Task is in PEND state with a timeout value
PEND + S + T	Task is in 2 states: PEND state with a timeout value and SUSPEND



This state diagram shows how a vxWorks task can switch between all of the various states.

Figure 9-17a1
State diagram for VxWorks tasks.[5]

---

8. What is the difference between pre-emptive and non-pre-emptive scheduling? Give examples of OSes that implement pre-emptive and non-pre-emptive scheduling.

Scheduling algorithms implemented in embedded OSs typically fall under two approaches: non-pre-emptive and pre-emptive scheduling.

Under non-pre-emptive scheduling, tasks are given control of the master CPU until they have finished execution, regardless of the length of time or the importance of the other tasks that are waiting. OS can't force context switch in non-pre-emptive scheduling. Microsoft Windows 3.x and Classic Mac OS (prior to Mac OS X) are examples of OSes which implement non-pre-emptive scheduling.

In pre-emptive scheduling, on the other hand, the OS forces a context-switch on a task, whether a running task has completed executing or is cooperating with the context switch. Linux and Windows are examples of OSes which implement pre-emptive scheduling.

---

9. What is a real time operating system (RTOS)? Give two examples of RTOSes.

If in an OS, tasks always meet their execution time deadlines and related execution times are predictable (deterministic), the OS is referred to as an RTOS.

Examples include VxWorks (Wind River), Linux (Timesys)

---

10. [T/F] A RTOS does not contain a pre-emptive scheduler. **False**

---

11. Name and describe the most common OS inter-task communication and synchronization mechanisms.

Embedded OSs with multiple intercommunicating processes commonly implement inter-process communication (IPC) and synchronization algorithms based upon one or some combination of *memory sharing, message passing, and signalling mechanisms.*

With the shared data model shown in Figure 9-28, processes communicate via access to *shared areas of memory* in which variables modified by one process are accessible to all processes.

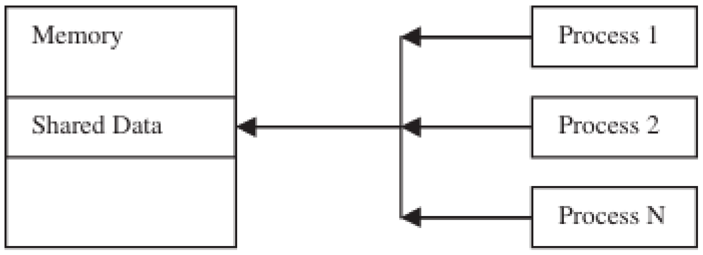


Figure 9-28
Memory sharing.

Inter-task communication via message passing is an algorithm in which messages (made up of data bits) are sent via message queues between processes. The OS defines the protocols for process addressing and authentication to ensure that messages are delivered to processes reliably, as well as the number of messages that can go into a queue and the message sizes.

Signals are indicators to a task that an asynchronous event has been generated by some external event (other processes, hardware on the board, timers, etc.) or some internal event (problems with the instructions being executed, etc.). When a task receives a signal, it suspends executing the current instruction stream and context switches to a signal handler (another set of instructions). However, signals can be used for general inter-task communication, but are implemented so that the possibility of a signal handler blocking or a deadlock occurring is avoided. The other inter-task communication mechanisms (shared memory, message queues, etc.), along with signals, can be used for ISR-to-Task level communication, as well.

---

12. What are race conditions? What are some techniques for resolving race conditions?

A race condition occurs when a process that is accessing shared variables is pre-empted before completing a modification access, thus affecting the integrity of shared variables. To counter this issue, portions of processes that access shared data, called critical sections, can be earmarked for mutual exclusion (or Mutex for short). Mutex mechanisms allow shared memory to be locked up by the process accessing it, giving that process exclusive access to shared data. Mutual exclusion techniques for synchronizing tasks that wish to concurrently access shared data can include:

a) *Processor-assisted locks* for tasks accessing shared data that are scheduled such that no other tasks can pre-empt them; the only other mechanisms that could force a context switch are interrupts. Disabling interrupts while executing code in the critical section would avoid a race condition scenario if the interrupt handlers accessed the same data. Under this mechanism, the setting and testing of a register flag (condition) is an atomic function, a process that cannot be interrupted, and this flag is tested by any process that wants to access a critical section.
b) *Semaphores*, which can be used to lock access to shared memory (mutual exclusion) and can be used to coordinate running processes with outside events (synchronization). The semaphore functions are atomic functions and are usually invoked through system calls by the process.

---

13. The OS inter-task communication mechanism typically used for interrupt handling is: a message queue / a signal / a semaphore / **All the above /** None of the above.

---

14. What is the difference between processes running in kernel mode and those running in user mode? Give an example of the type of code that would run in each mode.

Most OS processes typically run in one of two modes: *kernel mode* and *user mode*, depending on the routines being executed. Kernel routines run in kernel mode (also referred to as supervisor mode), in a different memory space and level than higher layers of software such as middleware or applications. Typically, these higher layers of software run in user mode, and can only access anything running in kernel mode via system calls, the higher-level interfaces to the kernel's subroutines. The kernel manages memory for both itself and user processes.

---

15. What is segmentation? What are segment addresses made up of? What type of information can be found in a segment?

1. *Process and its Information:*
   - A process encapsulates all the information required for executing a program, including source code, stack, and data.
   - Different types of information within a process are divided into "logical" memory units called segments.
2. *Segment Structure:*
   - Segments are logical memory units of variable sizes, each containing a set of logical addresses with the same type of information.
   - Segment addresses start at 0 and consist of a segment number (base address of the segment) and a segment offset (actual physical memory address).
3. *Segment Protection:*

Segments are independently protected with assigned accessibility characteristics such as shared, read-only, or read/write.

4. *Types of Information within Segments:*

Most operating systems allow processes to have some combination of five types of information within segments:

- *Text Segment*: Contains the source code.
- *Data Segment*: Holds the source code's initialized variables (data).
- *BSS Segment*: Statically allocated memory space for the source code's uninitialized variables (data).
- *Stack Segment*: Used for function call stack and local variables.
- *Heap Segment*: Dynamically allocated memory space for program runtime.

---

16. [T/F] A stack is a segment of memory that is structured as a FIFO queue. **False** – LIFO queue.

---

17. What is paging? Name and describe four OS algorithms that can be implemented to swap pages in and out of memory.

Either with or without segmentation, some OSs divide logical memory into some number of fixed-size partitions, called blocks, frames, pages, or some combination of a few or all of these. For example, with OSs that divide memory into frames, the logical address is a compromise of a frame number and offset. The user memory space can then, also, be divided into pages, where page sizes are typically equal to frame sizes. When a process is loaded in its entirety into memory (in the form of pages), its pages may not be located within a contiguous set of frames. Every process has an associated process table that tracks its pages, and each page's corresponding frames in memory. The logical address spaces generated are unique for each process, even though multiple processes share the same physical memory space. Logical address spaces are typically made up of a page-frame number, which indicates the start of that page, and an offset of an actual memory location within that page. In essence, the logical address is the sum of the page number and the offset.

An OS may start by pre-paging, or loading the pages needed to get started, and then implementing the scheme of demand paging where processes have no pages in memory and pages are

---

only loaded into RAM when a page fault (an error occurring when attempting to access a page not in RAM) occurs.

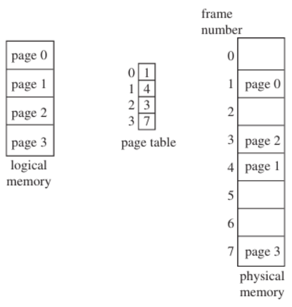


Figure 9-37
Paging.[3]

Page replacement policies:

i. *Optimal*: Using future reference time swapping out pages that won't be used in the near future.
ii. *Least Recently Used (LRU)*: Swaps out pages that have been used the least recently.
iii. *First In First Out (FIFO)*: As its name implies, swaps out the pages that are the oldest (regardless of how often it is accessed) in the system. While a simpler algorithm then LRU, FIFO is much less efficient.
iv. *Not Recently Used (NRU)*: Swaps out pages that were not used within a certain time period.

---

18. What is virtual memory? Why use virtual memory?

Virtual memory is typically implemented via demand segmentation (fragmentation of processes from within, as discussed in a previous section) and/or demand paging (fragmentation of logical user memory as a whole) memory fragmentation techniques. When virtual memory is implemented via these "demand" techniques, it means that only the pages and/or segments that are currently in use are loaded into RAM.

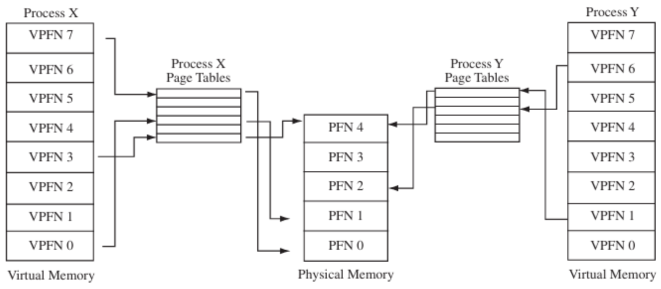


Figure 9-38
Virtual memory.[3]

In a virtual memory system, the OS generates virtual addresses based on the logical addresses and maintains tables for the sets of logical addresses into virtual addresses conversions (on some processors table entries are cached into translation lookaside buffers (TLBs). The OS (along with the hardware) then can end up managing more than one different address space for each process (the physical, logical, and virtual).
