Introduction to Embedded System

os

Embedded OS

- The OS is a set of software libraries
- An operating system (OS) is an optional part of an embedded device's system software stack
- An OS either sits over the hardware, over the device driver layer or over a BSP (Board Support Package)

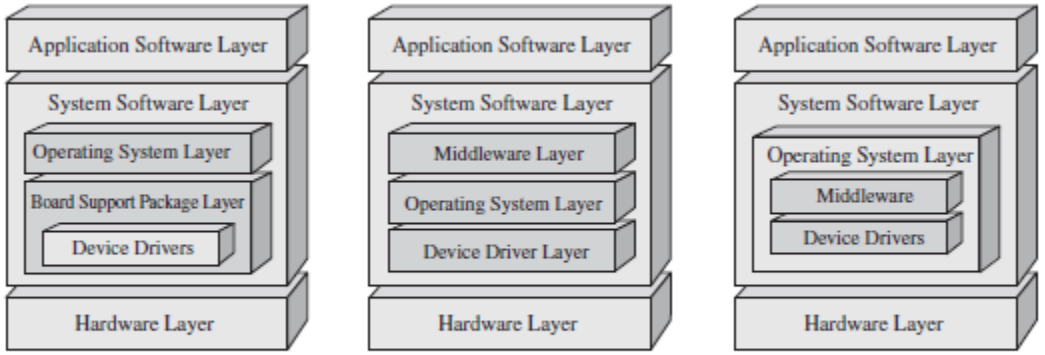


Figure 9-1: OSes and the Embedded Systems Model

Purpose of OS in Embedded system

- OS serves two main purposes in an embedded system:
  I. **Providing an abstraction layer** for other software to be less dependent on hardware, making the development of middleware and applications easier,
  II. **Managing the various system hardware and software resources** to ensure the entire system operates efficiently and reliably.

Functions of OS

- Process Management.
  - Process is the way in which an OS manages and views other software in the embedded system
  - Interrupt and error detection management
    - A subfunction found within process management
    - Multiple interrupts generated by the various processes need to be managed efficiently so that they are handled correctly and the processes that triggered them are properly tracked.
- Memory management
  - The embedded system's memory space is shared by all the different processes,
  - So access and allocation of portions of the memory space need to be managed

Cntd..

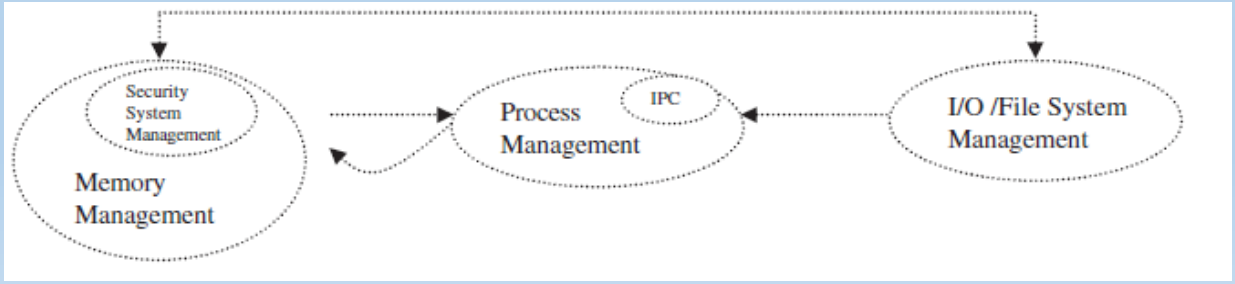
- I/O System Management
  - I/O devices also need to be shared among the various processes
  - So, just as with memory, access and allocation of an I/O device need to be managed.
  - Through I/O system management, file system management can also be provided as a method of storing and managing data in the form of files.

Kernel

- Embedded OSes vary in what components they possess,
- All OSes must have 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,

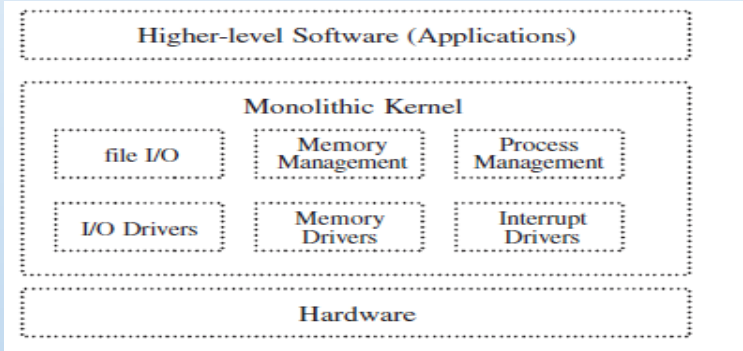
Kernel subsystem dependencies

- Process management component(PMC) is the central subsystem in an OS
- All other OS subsystems depend on the process management unit
- The PMC is equally dependent on the memory management subsystem
  - Because all code must be loaded into main memory (RAM or cache) for the master CPU to execute, with boot code and data located in non-volatile memory (ROM, Flash, etc.).
- I/O management, for example, could include networking I/O to interface with the memory manager in the case of a network file system (NFS).

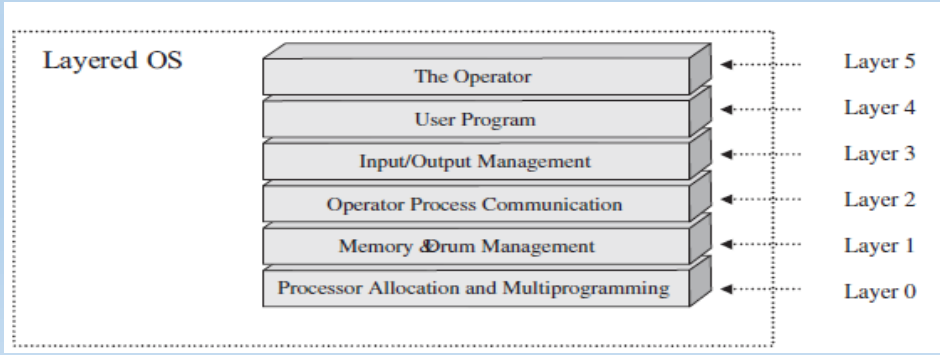


Different models of Embedded OS

- Most embedded OSes are typically based upon one of three models,
  1) Monolithic OS
     - ➢ It is a single executable file containing all of these components
     - ➢ middleware and device driver functionality is typically integrated into the OS along with the kernel
     - ➢ Difficult to scale down, modify, or debug
     - ➢ Example- Jbed RTOS, MicroC/OS-II, and PDOS

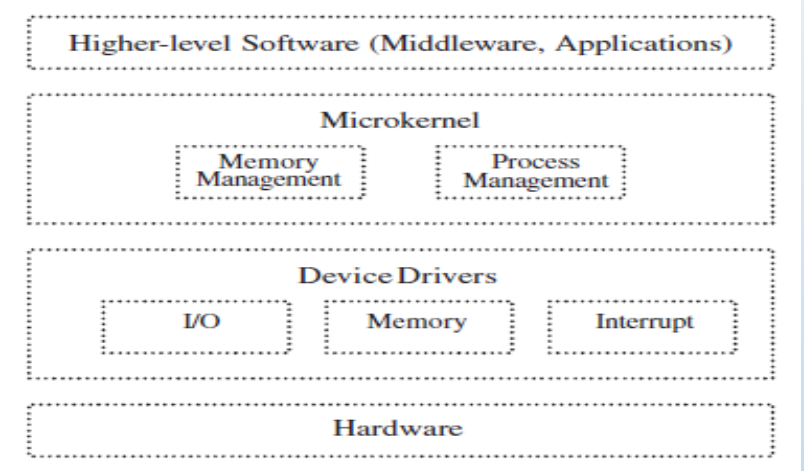


  2) Layered OS
     - ➢ OS is divided into hierarchical layers (0...N), where upper layers are dependent on the functionality provided by the lower layers.
     - ➢ Simpler to develop and maintain
     - ➢ APIs provided at each layer create additional overhead
       - Can impact size and performance
     - ➢ Example- DOS-C(FreeDOS), DOS/eRTOS, and VRTX



Different models of Embedded OS(cntd..)

3) Microkernel (client-server) design



- Has only process and memory management subunits
- More scalable (modular) and debuggable design,
  - Since additional components can be dynamically added
  - Easier to port to new architectures. However, this model
- More secure
  - since much of the functionality is now independent of the OS,
- There is a separate memory space for client and server functionality.
- May be slower
  - Because of the communication paradigm between the microkernel components and other “kernel-like” components.
- Overhead is also added when switching between the kernel and the other OS components and non-OS components
- Example- OS-9, C Executive, vxWorks, CMX-RTX, Nucleus Plus, and QNX.

What is a process?

- An OS differentiates between a **program** and the **Executing of a program**.
- **Program**
  - A passive, static sequence of instructions
  - Represent a system's hardware and software resources
- **Executing of a program**.
  - An active, dynamic event in which various properties change relative to time and the instruction being executed.
- A process is a program running in execution
- Process is commonly referred to as a ***task*** in many embedded OSes

Types of OS based on task

- A process is created by an OS to encapsulate all the information that is involved in the executing of a program

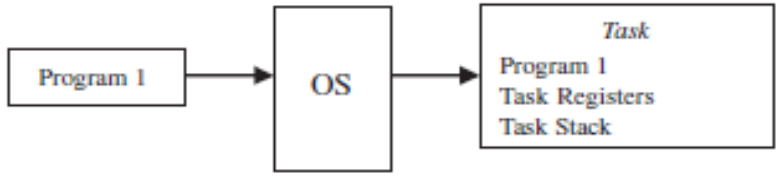


Figure 9-7: OS task

- Embedded OSes manage all embedded software using tasks, and can either be **unitasking** or **multitasking**
- **Unitasking OS**
  - Only one task can exist at any given time
  - Don’t require complex task management facility
  - No need to provide security to each process
  - Example: DOS-C

Cntd..

- **Multitasking**
  - ➢multiple tasks are allowed to exist simultaneously
  - ➢Require complex task management facility
    - Added complexity of allowing multiple existing tasks requires that each process remain independent of the others and not affect any other without the specific programming to do so.
  - ➢Provides each process with more security
  - ➢Provide a more organized way for a complex embedded system to function
- Examples:
  - ❑vxWorks (Wind River),
  - ❑Embedded Linux (Timesys), and
  - ❑Jbed (Esmertec)

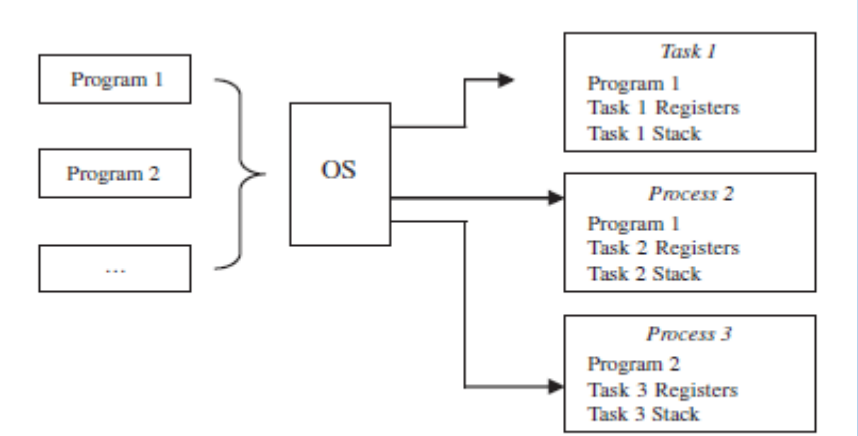


Figure 9-8: Multitasking OS

Threads

- A thread is a sequential execution stream within its task
- Threads are created within the context of a task, and,
- Depending on the OS, the task can own one or more threads
- Tasks have their own independent memory spaces that are inaccessible to other tasks,
- Threads of a task share the same resources (working directories, files, I/O devices, global data, address space, program code, etc.)
- Have their own Program Counter, stack, and scheduling information (PC, SP, stack, registers, etc.) To allow for the instructions they are executing to be scheduled independently

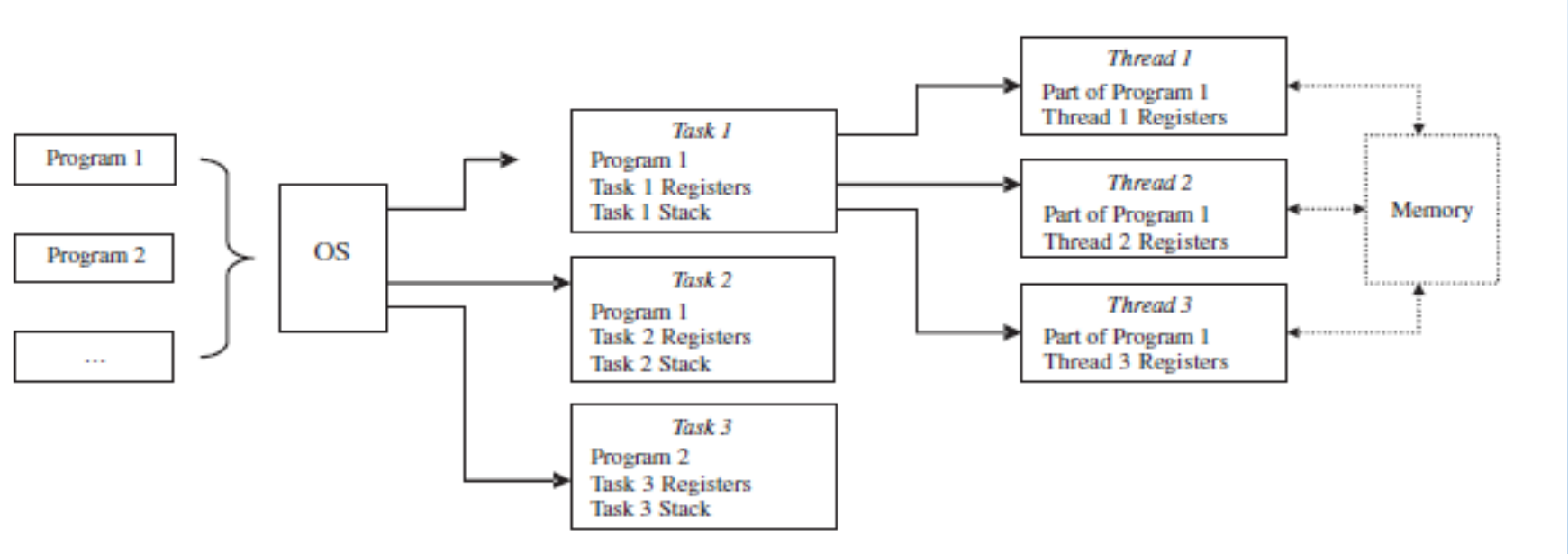
Why threads?

- Since **threads** are created within the context of the same task and can share the same memory space,
- Multiple threads are typically less expensive than creating multiple tasks to do the same work
- Threads can allow for simpler communication and coordination relative to tasks.
  - Because a task can contain at least one thread executing one program in one address space,
  - or a task can contain many threads executing different portions of one program in one address space, needing no intertask communication mechanisms

The benefits of multithreaded programming can be broken down into four major categories:

Responsiveness	Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
Resource sharing	By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
Economy	Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
Utilization of multiprocessor architectures	The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

Tasks and threads

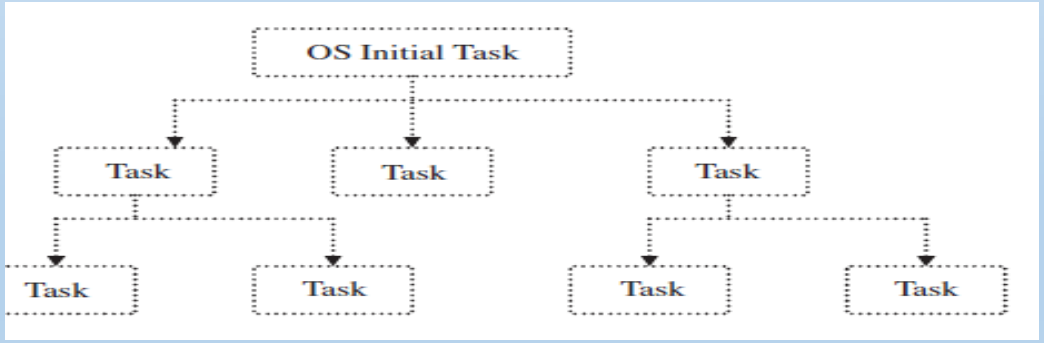


Multitasking and Process Management

- Even when an OS allows multiple tasks to coexist,
- One master processor on an embedded board can only execute one task or thread at any given time
- So multitasking embedded OSes must find some way of allocating each task a certain amount of time to use the master CPU
- The **various ways** via which an OS successfully gives the illusion of a single processor simultaneously running multiple tasks:
  1. Task/Process Implementation,
  2. Process Scheduling,
  3. Process Synchronization,
  4. Inter-task Communication Mechanisms

Process Implementation

- In multitasking embedded OSes, tasks are structured as a hierarchy of parent and child tasks,
- When an embedded kernel starts up only one task exists
  - first task is also created by the programmer in the system's initialization code,
- It is from this first task that all others are created



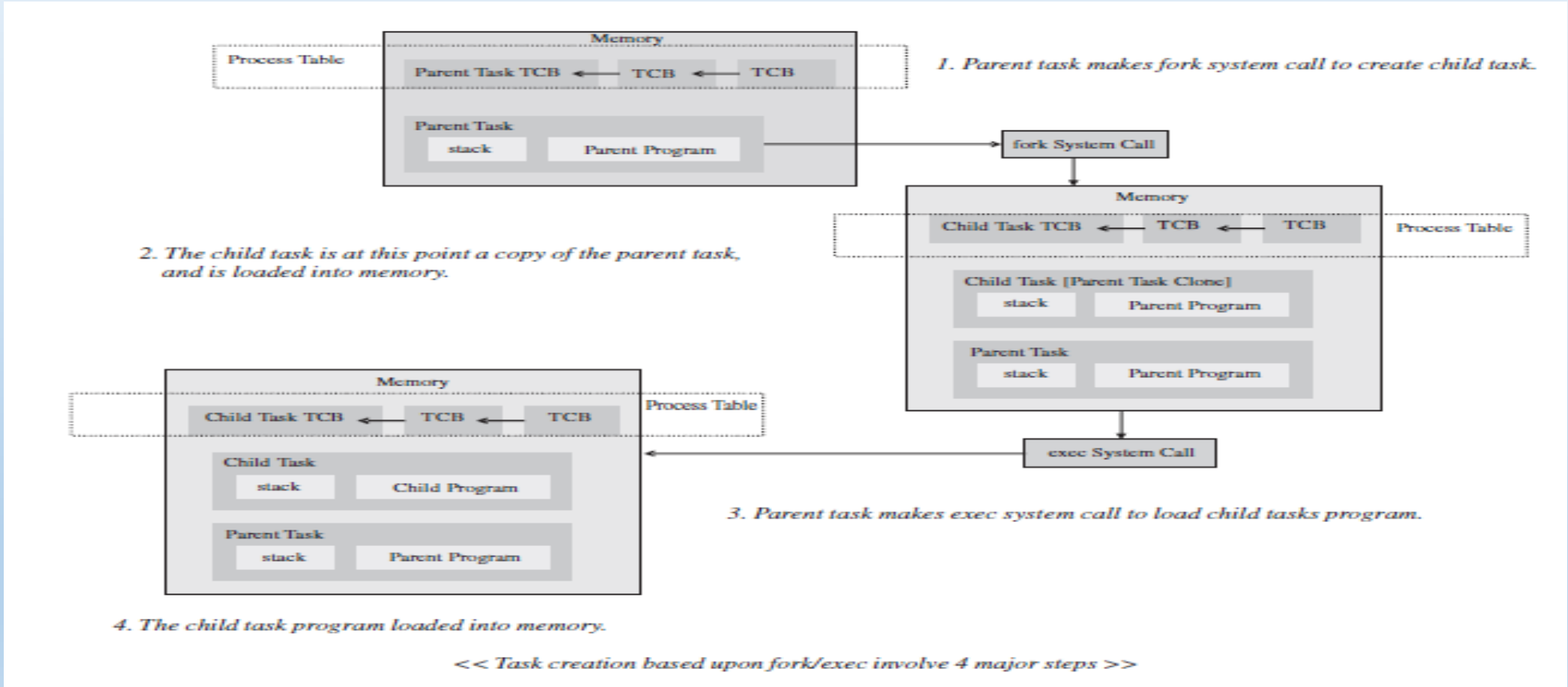
Task creation

- Task creation in embedded OSes is primarily based upon two models, **fork/exec** and **spawn**
- All tasks create their child tasks through fork/exec or spawn system calls.
- After the system call, the OS gains control and creates the **Task Control Block (TCB)/Process Control Block** (PCB) for that particular task
  - Contains OS control information, such as task ID, task state, task priority, and error status, and CPU context information, such as registers
- At this point, memory is allocated for the new child task, including for its TCB, any parameters passed with the system call, and the code to be executed by the child task.
- After the task is set up to run, the system call returns and the OS releases control back to the main program.

Fork/exec process creation

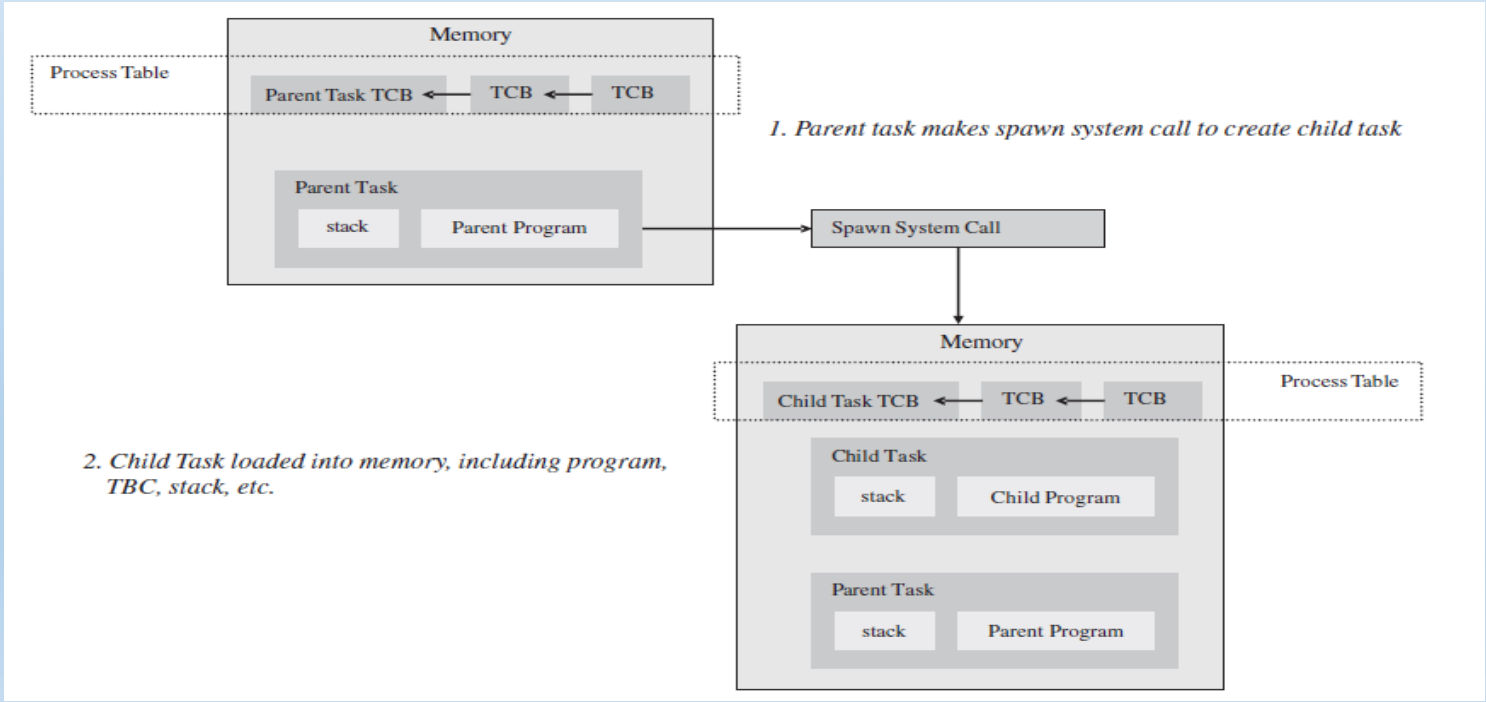
- Under the fork/exec model, the “fork” call creates a copy of the parent task’s memory space in what is allocated for the child task,
- Thus allowing the child task to inherit various properties, such as program code and variables, from the parent task.
- Because the parent task’s entire memory space is duplicated for the child task, two copies of the parent task’s program code are in memory,
  - One for the parent, and one belonging to the child.
- The “exec” call is used to explicitly remove from the child task’s memory space any references to the parent’s program and sets the new program code belonging to the child task to run.

Fork/exec process creation(cntd..)



Spawn process creation

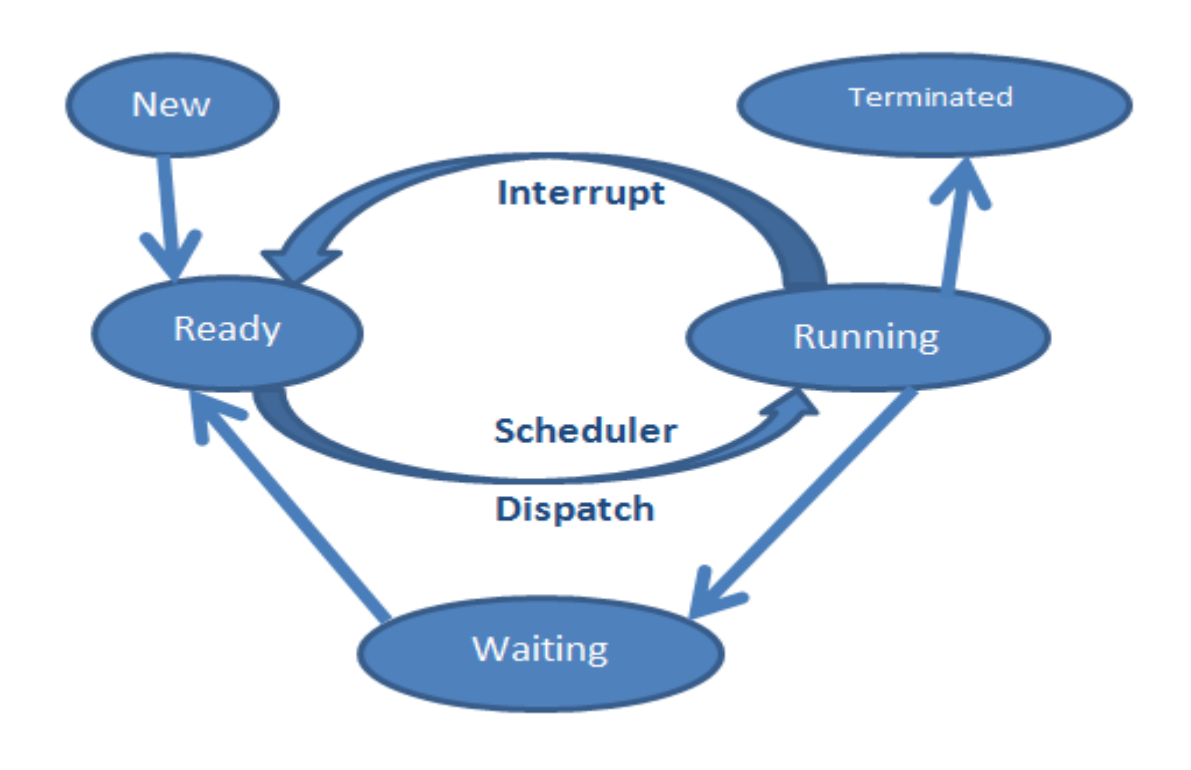
- Creates an entirely new address space for the child task.
- The spawn system call allows for the new program and arguments to be defined for the child task.
- This allows for the child task's program to be loaded and executed immediately at the time of its creation.



Advantage and Disadvantages

- Drawbacks of spawn approach :
  - Under the spawn approach, there are no duplicate memory spaces to be created and destroyed,
  - and then new space allocated, as is the case with the fork/exec model
- Advantages of the fork/exec model:
  - include the efficiency gained by the child task inheriting properties from the parent task,
  - Have the flexibility to change the child task's environment afterwards.

Process states

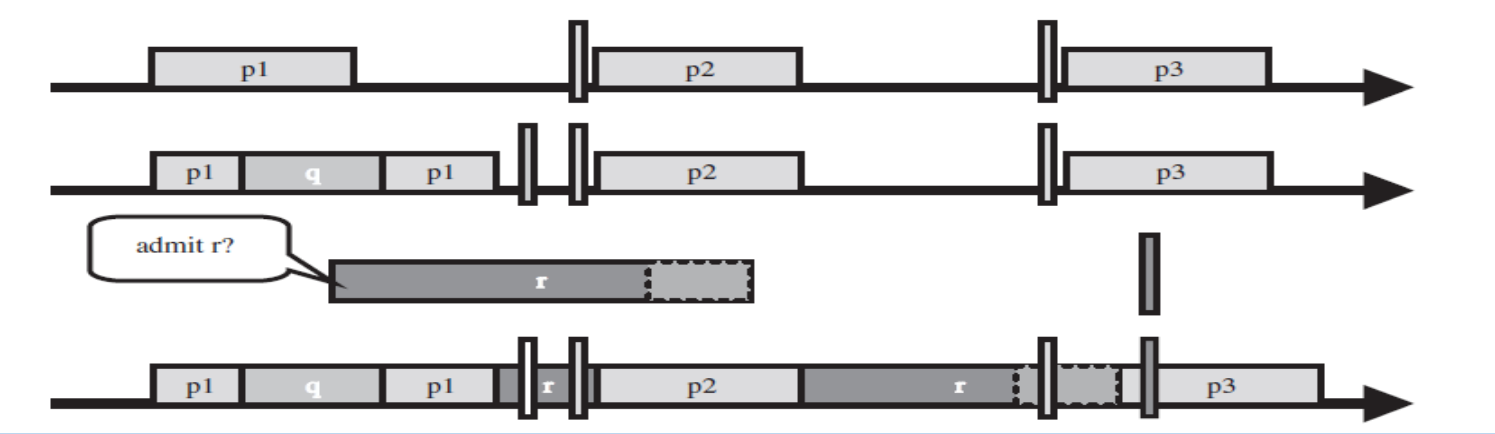


Process scheduling

- In the embedded OS, scheduling algorithms fall under two approaches:
  - Non-preemptive and
    - 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
    - Example – FCFS, Shortest Process Next (SPN),
    - **Co-operative,**
      - here the tasks themselves run until they tell the OS when they can be context switched (i.e., for I/O, etc.).
      - This algorithm can be implemented with the FCFS or SPN algorithms, rather than the run-to-completion scenario, but starvation could still occur with SPN if shorter processes were designed not to "cooperate,"
  - Preemptive scheduling
    - Example- **Round Robin/FIFO, Priority (Preemptive) Scheduling, EDF (Earliest Deadline First)/Clock Driven Scheduling**

EDF (Earliest Deadline First)/Clock Driven Scheduling

- Schedules priorities to processes according to three parameters: frequency (number of times the process is run), deadline (when processes execution needs to be completed), and duration (time it takes to execute the process).
- While the EDF algorithm allows for timing constraints to be verified and enforced (basically guaranteed deadlines for all tasks), the difficulty is defining an exact duration for various processes.



Process Synchronization

- Different tasks in an embedded system typically must share the same hardware and software resources,
- Or may rely on each other in order to function correctly
- Synchronization is needed to coordinate their functions, avoid problems, and allow tasks to run simultaneously in harmony
- **Semaphores**
  - Used to lock access to shared memory (mutual exclusion), and also 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.

Memory management

- Kernel manages program code within an embedded system via tasks
- kernel's memory management responsibilities include:
  - Managing the mapping between logical (physical) memory and task memory references.
  - Determining which processes to load into the available memory space.
  - Allocating and deallocating of memory for processes that make up the system.
  - Supporting memory allocation and deallocation of code requests (within a process) such as the C language "alloc" and "dealloc" functions, or specific buffer allocation and deallocation routines.
  - Tracking the memory usage of system components.
  - Ensuring cache coherency (for systems with cache).
  - Ensuring process memory protection.

Why Memory address conversion?

- Physical memory is composed of two-dimensional arrays made up of cells addressed by a unique row and column, in which each cell can store 1 bit.
- The OS treats memory as one large one-dimensional array, called a memory map.
- Either a hardware component integrated in the master CPU or on the board does the conversion between logical and physical addresses (such as an MMU), or it must be handled via the OS.

User Memory Space

- Because multiple processes are sharing the same physical memory when being loaded into RAM for processing,
- There also must be some protection mechanism so processes cannot inadvertently affect each other when being swapped in and out of a single physical memory space.
- These issues are typically resolved by the operating system through memory “swapping,” where partitions of memory are swapped in and out of memory at run-time.
- The most common partitions of memory used in swapping are **segments** (fragmentation of processes from within) and **pages** (fragmentation of logical memory as a whole).

Virtual memory

- A mechanism managed by the OS to allow a device's limited memory space to be shared by multiple competing "user" tasks,
- Enlarging the device's actual physical memory space into a larger "virtual" memory space.
- **Segment:**
  - A set of logical addresses containing the same type of information
  - A process encapsulates all the information that is involved in executing a program, including source code, stack, data, and so on.
  - All of the different types of information within a process are divided into "logical" memory units of variable sizes, **called segments**.

Segmentation

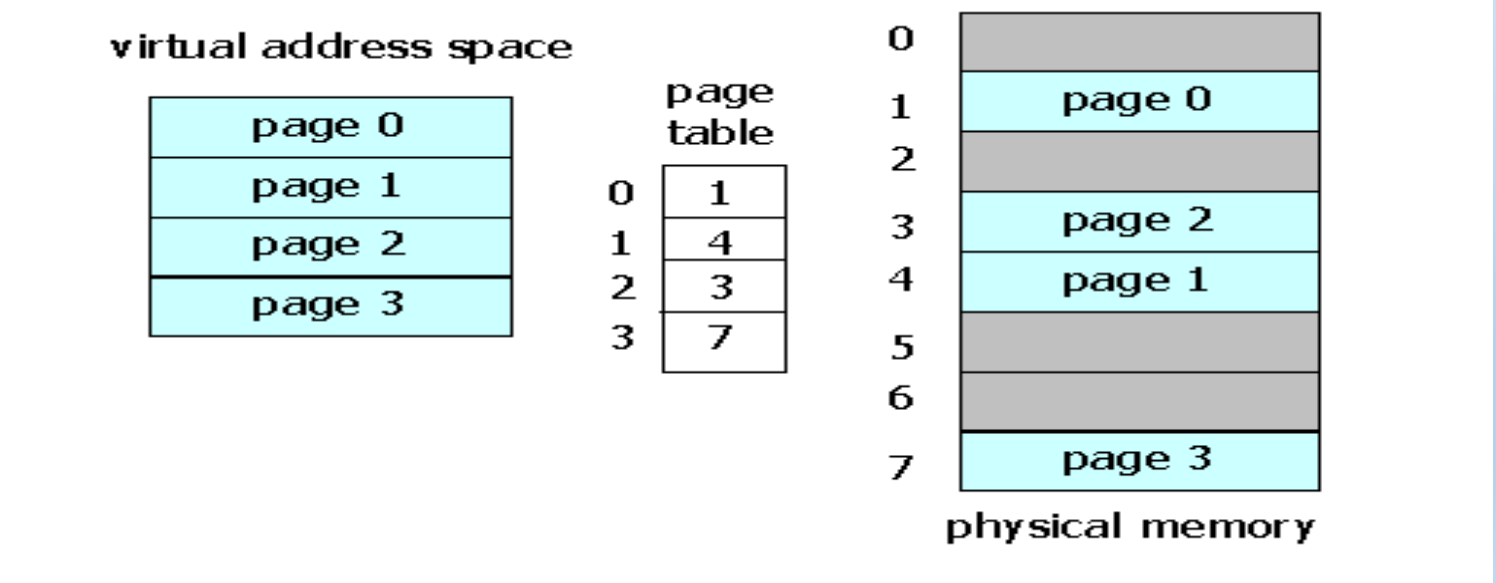
- Segment is made up of
  - a segment number,
    - Indicates the base address of the segment, and
  - a segment offset,
  - Processes can have all or some combination of five types of information within segments:
    - **Text (or code) segment**,
    - **Data segment**; contains initialized variable
    - **BSS (block started by symbol)** segment; contains un-initialized variable
    - **Stack** segment,
    - **Heap** segment

Memory allocation schemes

- FF (first fit) algorithm,
  - Here the list is scanned from the beginning for the first “hole” that is large enough.
- NF (next fit)
  - Here the list is scanned from where the last search ended for the next “hole” that is large enough.
- BF (best fit)
  - Here the entire list is searched for the hole that best fits the new data.
- WF (worst fit)
  - Placing data in the largest available “hole”.
- QF (quick fit)
  - Here a list is kept of memory sizes and allocation is done from this information.
- The buddy system
  - here blocks are allocated in sizes of powers of 2.
  - When a block is deallocated, it is then merged with contiguous blocks.

Paging and Virtual Memory

- Either with or without segmentation, some OSes divide logical memory into some number of fixed-size partitions, called pages.
- The Main Memory is divided into ‘Frames’ so that a part of the process (a page) can be contained in a frame (part of the main memory).



Page selection and replacement schemes

- Common page selection and replacement schemes to determine which pages are swapped include:
- **Optimal**, using future reference time, swapping out pages that won't be used in the near future.
- **Least Recently Used (LRU)**, which swaps out pages that have been used the least recently.
- **FIFO (First-In-First-Out)**,
  - 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.

Cntd..

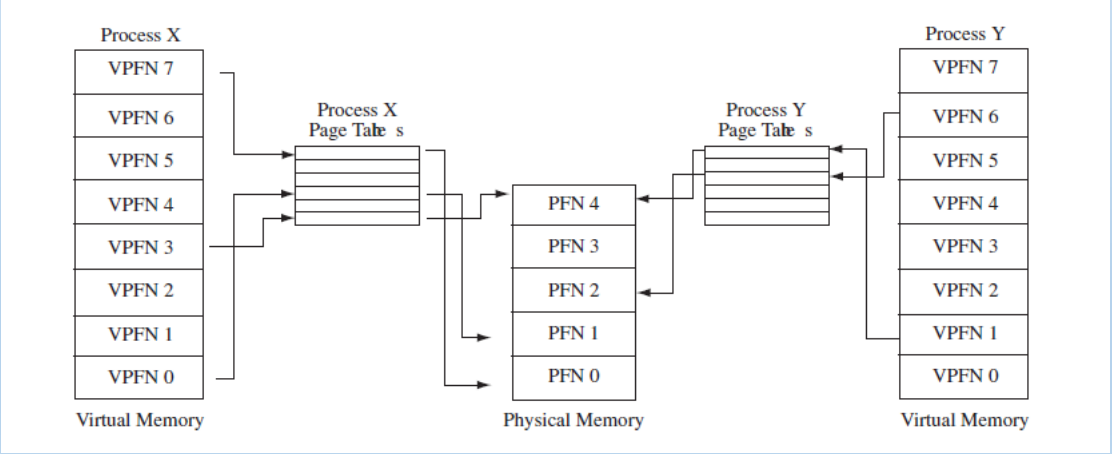
- ***Not Recently Used*** (NRU),
  - Swaps out pages that were not used within a certain time period.
- ***Second Chance***,
  - FIFO scheme with a reference bit, if “0” will be swapped out (a reference bit is set to “1” when access occurs, and reset to “0” after the check).
- ***Clock Paging***,
  - Pages replaced according to clock (how long they have been in memory), in clock order, if they haven’t been accessed (a reference bit is set to “1” when access occurs, and reset to “0” after the check).

Thrashing

- While every OS has its own swap algorithm, all are trying to reduce the possibility of **thrashing**,
  - A situation in which a system's resources are drained by the OS constantly swapping in and out data from memory.
- To avoid thrashing, a kernel may implement a working set model,
- which keeps a fixed number of pages of a process in memory at all times

Virtual memory

- 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.



Kernel Memory Space

- The kernel's memory space is the portion of memory in which the kernel code is located,
- some of which is accessed via system calls by higher-level software processes, and is where the CPU executes this code from.
- Code located in the kernel memory space includes required IPC mechanisms, such as those for message-passing queues.
- Another example is when tasks are creating some type of fork/exec or spawn system calls.
- Software running in user mode can only access anything running in kernel mode via system calls

I/O and File System Management

- File systems are essentially a collection of files along with their management protocols
- Some embedded OSes provide memory management support for a temporary or permanent file system storage scheme on various memory devices, such as Flash, RAM, or hard disk.
- File system algorithms are middleware and/or application software that is mounted(installed) at some mount point (location) in the storage device.

Middleware file system standards

File System	Summary
FAT32 (File Allocation Table)	Where memory is divided into the smallest unit possible (called sectors). A group of sectors is called a cluster. An OS assigns a unique number to each cluster, and tracks which files use which clusters. FAT32 supports 32-bit addressing of clusters, as well as smaller cluster sizes than that of the FAT predecessors (FAT, FAT16, etc.)
NFS (Network File System)	Based on RPC (Remote Procedure Call) and XDR (Extended Data Representation), NFS was developed to allow external devices to mount a partition on a system as if it were in local memory. This allows for fast, seamless sharing of files across a network.
FFS (Flash File System)	Designed for Flash memory.
DosFS	Designed for real-time use of block devices (disks) and compatible with the MS-DOS file system.
RawFS	Provides a simple *raw file system* that essentially treats an entire disk as a single large file.
TapeFS	Designed for tape devices that do not use a standard file or directory structure on tape. Essentially treats the tape volume as a raw device in which the entire volume is a large file.
CdromFS	Allows applications to read data from CD-ROMs formatted according to the ISO 9660 standard file system.

Purpose

- kernel typically provides file system management mechanisms for, at the very least:
- Mapping files onto secondary storage, Flash, or RAM (for instance).
- Supporting the primitives for manipulating files and directories.
  - File Definitions and Attributes: Naming Protocol, Types (i.e., executable, object, source, multimedia, etc.), Sizes, Access Protection (Read, Write, Execute, Append, Delete, etc.), Ownership, and so on.
  - File Operations: Create, Delete, Read, Write, Open, Close, and so on.
  - File Access Methods: Sequential, Direct, and so on.
  - Directory Access, Creation and Deletion.

I/O Management in embedded OSes

- Provides an additional abstraction layer (to higher level software) away from the system's hardware and device drivers.
- An OS provides a uniform interface for I/O devices that perform a wide variety of functions via the available kernel system calls
  - providing protection to I/O devices since user processes can only access I/O via these system calls, and
  - managing a fair and efficient I/O sharing scheme among the multiple processes.

OS Standards Example: POSIX (Portable Operating System Interface)

- Standards may greatly impact the design of a system component
- One of the key standards implemented in
- off-the-shelf embedded OSes today is POSIX
  - It is based upon the IEEE (1003.1-2001) and The Open Group (The Open Group Base Specifications Issue 6) set of standards that define a standard operating system interface and environment.

POSIX functionality

- POSIX provides OS-related standard APIs and definitions for process management, memory management, and I/O management functionality

OS Subsystem	Function	Definition
Process Management		
	Threads	Functionality to support multiple flows of control within a process. These flows of control are called threads and they share their address space and most of the resources and attributes defined in the operating system for the owner process. The specific functional areas included in threads support are: • Thread management: the creation, control, and termination of multiple flows of control that share a common address space. • Synchronization primitives optimized for tightly coupled operation of multiple control flows in a common, shared address space.
	Semaphores	A minimum synchronization primitive to serve as a basis for more complex synchronization mechanisms to be defined by the application program.
	Priority scheduling	A performance and determinism improvement facility to allow applications to determine the order in which threads that are ready to run are granted access to processor resources.

Cntd..

OS Subsystem	Function	Definition
Process Management	Real-time signal extension	A determinism improvement facility to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signal functions.
	Timers	A mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed.
	IPC	A functionality enhancement to add a high-performance, deterministic interprocess communication facility for local communication.
Memory Management		
	Process memory locking	A performance improvement facility to bind application programs into the high-performance random access memory of a computer system. This avoids potential latencies introduced by the operating system in storing parts of a program that were not recently referenced on secondary memory devices.
	Memory mapped files	A facility to allow applications to access files as part of the address space.
	Shared memory objects	An object that represents memory that can be mapped concurrently into the address space of more than one process.
I/O Management	Synchronionized I/O	A determinism and robustness improvement mechanism to enhance the data input and output mechanisms, so that an application can ensure that the data being manipulated is physically present on secondary mass storage devices.
	Asynchronous I/O	A functionality enhancement to allow an application process to queue data input and output commands with asynchronous notification of completion.
...	...	...