```
* Kernel, Services, Libraries, Application: define the 4 terms, and their
roles.
```

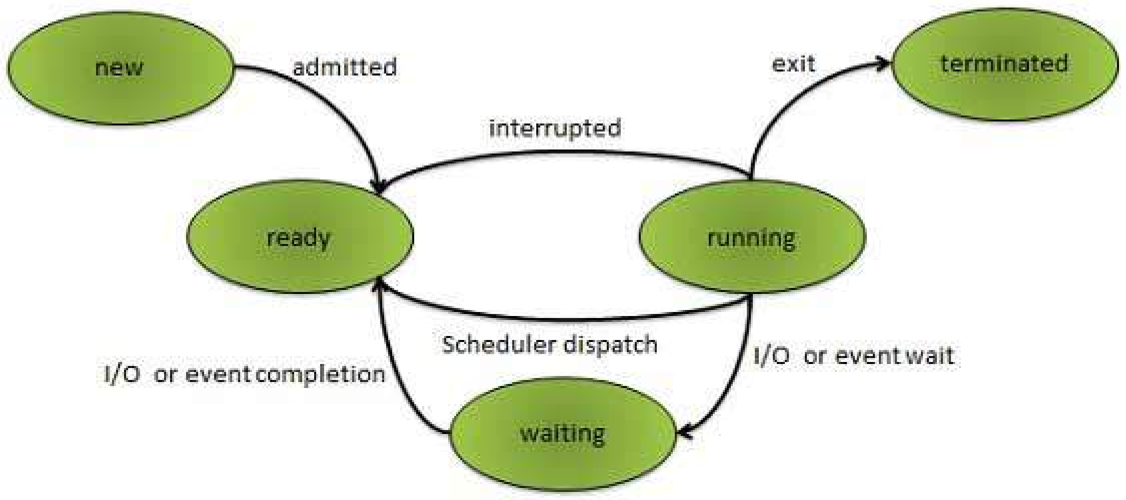
The **kernel** is a computer program that manages input/output requests from software, and translates them into data processing instructions for the central processing unit and other electronic components of a computer. The kernel is a fundamental part of a modern computer's operating system.

System services are programs that load automatically either as part of an application's startup process or the operating **system** startup process to support the different tasks required of the operating **system**.

A **library** is a collection of non-volatile resources used by **computer** programs, often to develop software. These may include configuration data, documentation, help data, message templates, pre-written code and subroutines, classes, values or type specifications.

An application is a program designed to perform a function or suite of related functions of benefit to an end user, with examples being accounting, mathematical analysis, video editing, and word processing.

```
---------------------------------------------------------------------------
* What are the different states for a task in an OS ?
```



```
---------------------------------------------------------------------------
* Name and explain 3 scheduling algorithms
```

FCFS : First-Come-First-Leave : non-preemptive algo
SJF: Shortest-Job-First: shortest cpu burst process first
RR: Round Robin: Each process is run for a time quantum (10-100ms) before being put back in at the end of the ready queue

```
---------------------------------------------------------------------------
* What are signals in UNIX systems ? give 5 common signal names and explain
them
```

Signals are a limited form of inter-process communication used in **Unix**, **Unix**-like, and other POSIX-compliant operating systems. A **signal** is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred.

- Ctrl-C (in older Unixes, DEL) sends an INT signal (**SIGINT**); by default, this causes the process to terminate.
- Ctrl-Z sends a TSTP signal (**SIGTSTP**); by default, this causes the process to suspend execution.
- Ctrl-\ sends a QUIT signal (**SIGQUIT**); by default, this causes the process to terminate and dump core.
- The **SIGKILL** signal is sent to a process to cause it to terminate immediately (**kill**). In contrast to SIGTERM and SIGINT, this signal cannot be caught or ignored, and the receiving process cannot perform any clean-up upon receiving this signal.
- The **SIGSEGV** signal is sent to a process when it makes an invalid virtual memory reference, or segmentation fault, i.e. when it performs a ***seg*mentation *v*iolation**.

----

* How can we execute custom behavior when receiving a signal ? Which signals can't have their default behavior overriden ?

- By **catching** a signal, and do whatever you want after.

- Non-catchable signals: **SIGKILL**, **SIGSTOP**

----

* Describe a mechanism for enforcing memory protection in order to prevent a program from modifying the memory associated with other programs.

The processor could keep track of what locations are associated with each process and limit access to locations that are outside of a program's extent. Information regarding the extent of a program's memory could be maintained by using base and limits registers and by performing a check for every memory access.

----

* Some computer systems do not provide a privileged mode of operation in hardware. Is it possible to construct a secure operating system for these computer systems? Give arguments both that it is and that it is not possible.

An operating system for a machine of this type would need **to remain in control**(or monitor mode) at all times. This could be accomplished by two methods:

Software interpretation of all user programs (like some BASIC, APL, and LISP systems, for example). The software interpreter would provide, in software, what the hardware does not provide.

Require meant that all programs **be written in high-level languages** so that all object code is compiler-produced. The compiler would generate (either in-line or by function calls) the protection checks that the hardware is missing.

----

* Which of the following instructions should be privileged?
* a. Set value of a timer.

Privileged

* b. Read the clock.

Unprivileged, every process should be able to read the clock.

* c. Clear memory.

Unprivileged because this only harms the process calling it

* d. Issue a trap instruction.

Unprivileged

* e. Turn off interrupts.

Privileged so that a process cannot monopolize the cpu.

* f. Modify entries in device-status table.

Privileged

* g. Switch from user to kernel mode.

Unprivileged because it's how applications invoke system calls. The catch is the application cannot control where the program counter goes when this switch happens.

* h. Access IO device.

Privileged because reading/writing to a device will interfere with other processes.

* What is the purpose of interrupts? What are the differences between a trap and an interrupt? Can traps be generated intentionally by a user program? If so, for what purpose?

The purpose of interrupts is to alter the flow of execution in response to some event. An interrupt is triggered in hardware and a trap is triggered in software. User programs can generate traps intentionally. They may want to interact with some I/O which requires a system call.

----

* What system calls have to be executed by a command interpreter or shell in order to start a new process?

In Unix systems, a fork system call followed by an exec system call need to be performed to start a new process. The fork call clones the currently executing process, while the exec call overlays a new process based on a different executable over the calling process.

----

* What are the advantages and disadvantages of using the same system-call interface for manipulating both files and devices?

Each device can be accessed as though it was a file in the file system. Since most of the kernel deals with devices through this file interface, it is relatively easy to add a new device driver by implementing the hardware-specific code to support this abstract file interface. Therefore, this benefits the development of both user program code, which can be written to access devices and files in the same manner, and device-driver code, which can be written to support a well-defined API. The disadvantage with using the same interface is that it might be difficult to capture the functionality of certain devices within the context of the file access API, thereby resulting in either a loss of functionality or a loss of performance. Some of this could be overcome by the use of the **input/output control** operation that provides a general-purpose interface for processes to invoke operations on devices.

----

* Describe the actions taken by a kernel to contex-switch between processes.

When switching from context *A* to *B* the registers used by *A* must be saved, then registers must be loaded for *B*. This requires special privileges so must be done by the kernel. Thus, the context must change from *A* to kernel and then to *B*. (Also the kernel will have to update relevent data structures in order to keep track of all this)

----

* It is said that fork(2) uses a copy on write mechanism. Explain fork semantics, and how it works in practice.

When a fork() system call is issued, a copy of all the pages corresponding to the parent process is created, loaded into a separate memory location by the OS for the child process. But this is not needed in certain cases. Consider the case when a child executes an "`exec`" system call (which is used to execute any executable file from within a C program) or exits very soon after the `fork()`. When the child is needed just to execute a command for the parent process, there is no need for copying the parent process' pages, since `exec` replaces the address space of the process which invoked it with the command to be executed.

In such cases, a technique called copy-on-write (COW) is used. With this technique, when a fork occurs, the parent process's pages are not copied for the child process. Instead, the pages are shared between the child and the parent process. Whenever a process (parent or child) modifies a page, a separate copy of that particular page alone is made for that process (parent or child) which performed the modification. This process will then use the newly copied page rather than the shared one in all future references. The other process (the one which did not modify the shared page) continues to use the original copy of the page (which is now no longer shared). This technique is called copy-on-write since the page is copied when some process writes to it.

----

* Give an example of a situation in which ordinary pipes are more suitable than named pipes and an example of a situation in which named pipes are more suitable that ordinary pipes.

Named pipes can be used to listen to requests from other processes(similar to TCP IP ports). If the calling processes are aware of the name, they can send requests to this. Unnamed pipes cannot be used for this purpose.

So the short answer is that you need a named pipe for communication between unrelated processes that might not exist at the same time.

Ordinary pipes are useful in situations where the communication needs to happen only between two specified process, known beforehand. Named pipes in such a scenario would involve too much of an overhead in such a scenario.

```
* Describe the differences among short-term, medium-term and long-term
scheduling.
```

- **Short-term** (CPU scheduler): selects a process from those that are in memory and ready to execute, and allocates the CPU to it. **Ready queue => executing**
- **Medium-term** (memory manager): selects processes from the ready or blocked queue and removes them from memory, then reinstates them later to continue running. (**swapping**)
- **Long-term** (job scheduler): determines which jobs are brought into the system for processing. **Disk => Ready queue**

```
* Using the following program, identify the values of pid at line A, B,
C, and D, assuming that the actual pids of the parent and child are 2600
and 2603.
```

```
#include <err.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main()
{
        pid_t pid, pid1;

        pid = fork();

        if (pid < 0) {
                err(1, "fork failed");
        } else if (pid == 0) {
                pid1 = getpid();
                printf("pid = %u\n", pid); /* A */
                printf("pid1 = %u\n", pid1); /* B */
        } else {
                pid1 = getpid();
                printf("pid = %u\n", pid); /* C */
                printf("pid1 = %u\n", pid1); /* D */

                wait(0);
        }

        return 0;
}
```

A = 0

B = 2603 (child)

C = 2603 (child) D = 2600 (father)

```
---------------------------------------------------------------------------

* Using the following program, explain what the output will be at line A.

        #include <stdio.h>
        #include <sys/types.h>
        #include <sys/wait.h>
        #include <unistd.h>

        int value = 5;

        int main()
        {
                pid_t pid = fork();

                if (pid == 0) {
                        value += 15;
                        return 0;
                } else if (pid > 0) {
                        wait(0);
                        printf("value = %d\n", value); /* A */
                        return 0;
                }
        }
```

Value = 5, because resources aren't shared between the father and child, but just copied

```
---------------------------------------------------------------------------

* Which of the following components of program state are shared across
threads in a multithreaded process ?
  a. Register values No
  b. Heap memory Yes
  c. Global variables Yes
  d. Stack memory No
---------------------------------------------------------------------------

* What are two differences between user-level threads and kernel-level
threads? Under what circumstances is one type better than the other?
```

(1) **User-level threads are unknown by the kernel**, whereas the
kernel is aware of kernel threads.
(2) On systems using either M:1 or M:N
mapping, **user threads are scheduled by the thread library** and the kernel
schedules kernel threads.
(3) Kernel threads need not be associated with
a process whereas **every user thread belongs to a process**. Kernel threads
are generally more expensive to maintain than user threads as they must
be represented with a kernel data structure.

```
---------------------------------------------------------------------------
```

* Can a multithreaded solution using multiple user-level threads achieve better performance on a multiprocessor system than on a single-processor system? Explain.

A multithreaded system comprising of multiple user-level threads cannot make use of the different processors in a multiprocessor system simultaneously. The operating system sees only a single process and will not schedule the different threads of the process on separate processors. Consequently, there is no performance benefit associated with executing multiple user-level threads on a multiprocessor system.

----

* Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single threaded solution on a single processor system?

When a kernel thread suffers a page fault, another kernel thread can be switched in to use the interleaving time in a useful manner. A single-threaded process, on the other hand, will not be capable of performing useful work when a page fault takes place. Therefore, in scenarios where a program might suffer from frequent page faults or has to wait for other system events, a multithreaded solution would perform better even on a single-processor system.

----

* Describe the actions taken by a thread library to context-switch between user-level threads.

Context switching between user threads is quite similar to switching between kernel threads, although it is dependent on the threads library and how it maps user threads to kernel threads. In general, context switching between user threads involves taking a user thread of its **LWP** (**light-weight process)** and replacing it with another thread. **This act typically involves saving and restoring the state of the registers**.

----

* Why is it important for the scheduler to distinguish IO-bound programs from CPU-bound programs?

IO-bound programs only perform a small amount of computation before the IO. Such programs do not use up their entire CPU quantum. CPU-bound programs use their entire quantum.

----

* Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

```
* Servers can be designed to limit the number of open connections. For
example, a server may wish to have only N socket connections at any point
in time. As soon as N connections are made, the server will not accept
another incomming connection until an existing connection is released.
Explain how semaphores can be used by a server to limit the number of
concurrent connections.

A semaphore is initialized to the number of allowable open socket
connections.When a connection is accepted, the acquire() method is called;
when a connection is released, the release() method is called. If the
system reaches the number of allowable socket connections, subsequent calls
to acquire() will block until an existing connection is terminated and the
release method is invoked.
---------------------------------------------------------------------------

* Explain why implementing synchronization primitives by disabling
interrupts is not appropriate in a single-processor system if the
synchronization primitives are to be used in user-level programs.

If a user-level program is given the ability to disable interrupts, then it
can disable the timer interrupt and prevent context switching from taking
place, thereby allowing it to use the processor without letting
other processes to execute.
---------------------------------------------------------------------------

 * What is a deadlock?
```

A **lock** occurs when multiple processes try to access the same resource at the same time.

One process loses out and must wait for the other to finish.

A **deadlock** occurs when the waiting process is still holding on to another resource that the first needs before it can finish.

```
---------------------------------------------------------------------------
* On a system with paging, a process cannot access memory that it does not
own. Why? How could the operating system allow access to other memory? Why
should it or should it not?
```

An address on a paging system is a logical page number and an offset. The physical page is found by searching a table based on the logical page number to produce a physical page number. Because the operating system controls the contents of this table, it can limit a process to accessing only those physical pages allocated to the process. There is no way for a process to refer to a page it does not own because the page will not be in the page table. To allow such access, an operating system simply needs to allow entries for non-process memory to be added to the process page table. This is useful when two or more processes need to exchange data – they just read and write to the same physical addresses (which may be at varying logical addresses). This makes for very efficient interprocess communication.

```
---------------------------------------------------------------------------
```

* Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.

A page fault occurs when an access to a page that has not been brought into main memory takes place. The operating system verifies the memory access, aborting the program if it is invalid. If it is valid, a free frame is located and I/O is requested to read the needed page into the free frame. Upon completion of I/O, the process table and page table are updated and the instruction is restarted.

----

* What is the copy-on-write feature, and under what circumstances is it beneficial to use this feature?

Copy on Write allows processes to share pages rather than each having a separate copy of the pages. However, when one process tried to write to a shared page, then a trap is generated and the OS makes a separate copy of the page for each process. This is commonly used in a fork() operation where the child is supposed to have a complete copy of the parent address space. Rather than create a separate copy, the OS allows the parent and child to share the parent's pages. However, since each is supposed to have its own private copy of the pages, the pages are copied when one of them attemps a write.

----

* Why do some systems keep track of the type of a file, while others leave it to the user and other simply do not implement multiple file types? Which system is "better"?

Some systems allow different file operations based on the type of the file (for instance, an ascii file can be read as a stream while a database file can be read via an index to a block). Other systems leave such interpretation of a file’s data to the process and provide no help in accessing the data. The method that is “better” depends on the needs of the processes on the system, and the demands the users place on the operating system. If a system runs mostly database applications, it may be more efficient for the operating system to implement a database-type file and provide operations, rather than making each program implement the same thing (possibly in different ways). For general-purpose systems it may be better to only implement basic file types to keep the operating system size smaller and allow maximum freedom to the processes on the system.

----

* If the operating system knew that a certain application was going to access file data in a sequential manner, how could it exploit this information to improve performance?

The prefetch could improve the performance, since prefetching the subsequent blocks of future needed can reduce the waiting time by the process for future requests.

----

* Consider a file system that uses inodes to represent files. Disk blocks are 4kb in size, and a pointer to a disk block requires 8bytes. This file system has 6 direct blocks, as well as single, double and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

(6 * 4 /KB/) + (2048 * 4 /KB) + (2048 * 2048 * 4 /KB/) +
(2048 * 2048 * 2048 * 4 /KB

----

* Why must the bit map for file allocation be kept on mass storage, rather than in main memory?

In case of system crash (memory failure) the free-space list would not be lost as it would be if the bit map had been stored in main memory.

----

* What is ioctl(2)? What are the issues solved by this system call? What are the problems caused by it? Propose a mechanism to solve its problems.

`ioctl` (an abbreviation of **input/output control**) is a system call for device-specific input/output operations and other operations which cannot be expressed by regular system calls. It takes a parameter specifying a request code; the effect of a call depends completely on the request code.

Modern operating systems support diverse devices, many of which offer a large collection of facilities. Some of these facilities may not be foreseen by the kernel designer, and as a consequence it is difficult for a kernel to provide system calls for using the devices.

To solve this problem, the kernel is designed to be extensible, and may accept an extra module called a **device driver** which runs in kernel space and can directly address the device.

----

* Describe how you could obtain a statistical profile of the amount of time spent by a program executing different sections of its code. Discuss the importance of obtaining such a statistical profile.

One could issue periodic timer interrupts and monitor what instructions or what sections of code are currently executing when the interrupts are delivered. A statistical profile of which pieces of code were active should be consistent with the time spent by the program in different sections of its code. Once such a statistical profile has been obtained, the programmer could optimize those sections of code that are consuming more of the CPU resources.