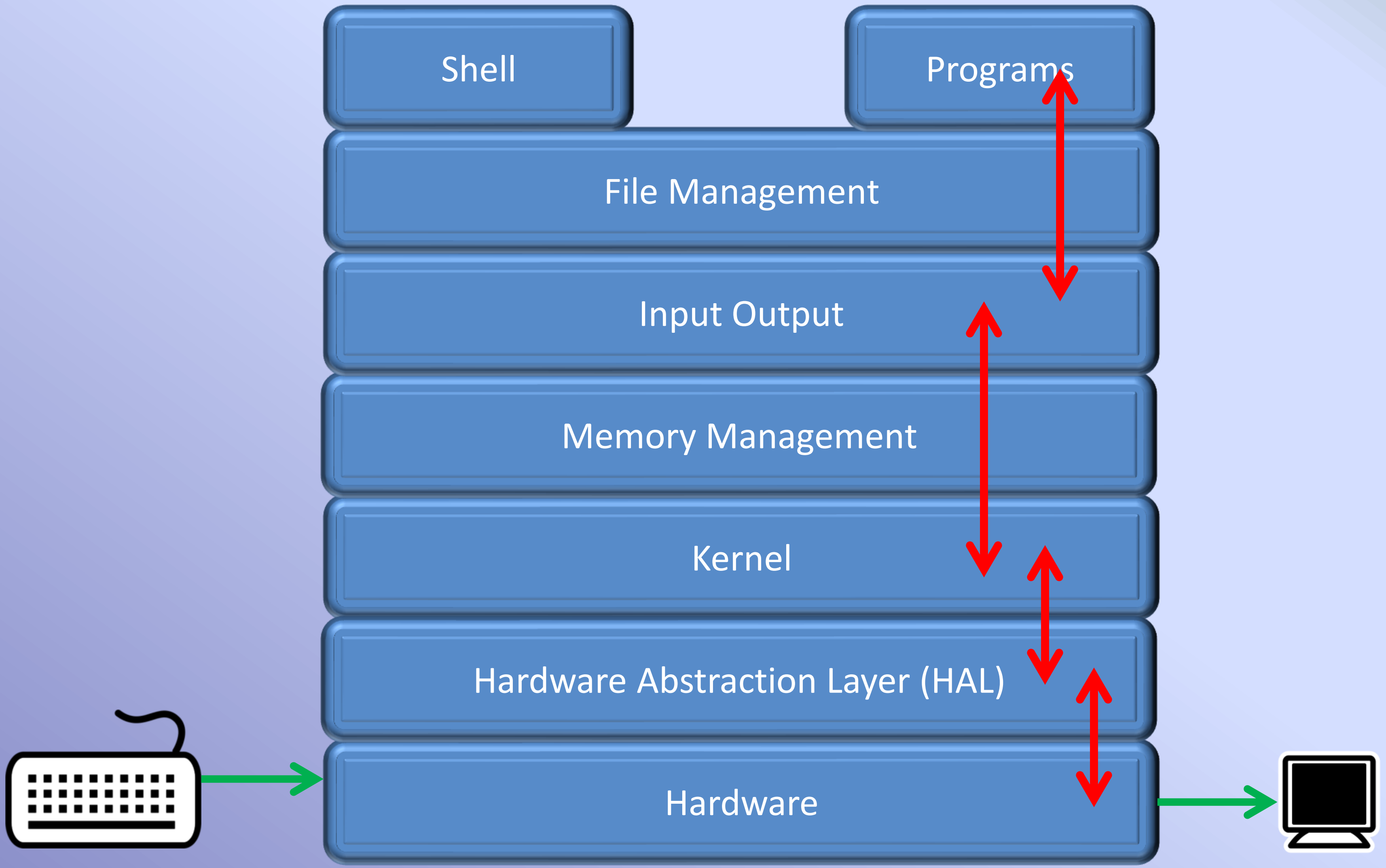
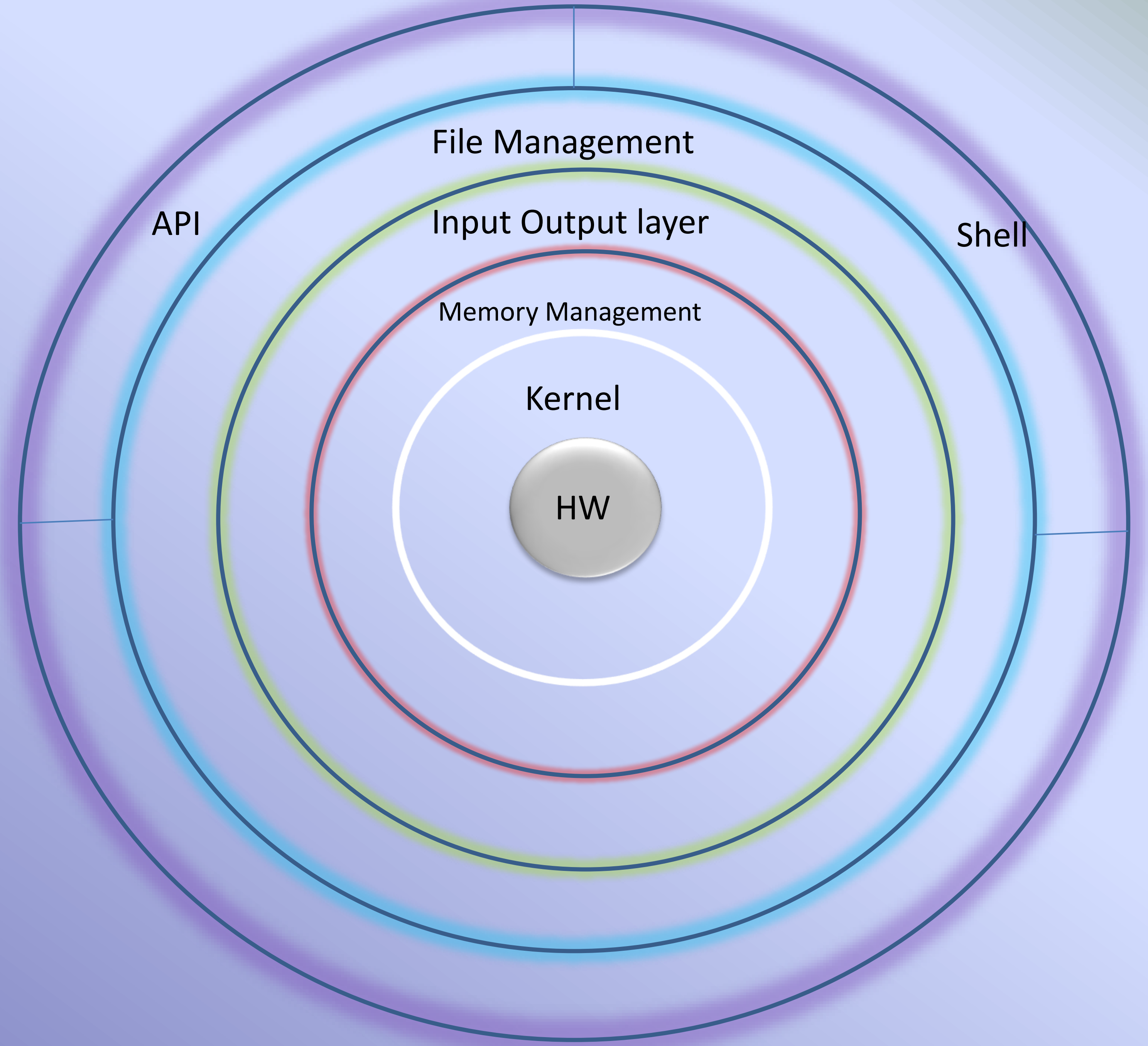
Real Time systems Categories

Components of OS

Layered Architecture of OS




File Management
API
Input Output layer
Shell
Memory Management
Kernel
HW

Program execution

- Loads a program into memory.
- Executes the program.
- Handles program's execution.
- Provides a mechanism for process synchronization.
- Provides a mechanism for process communication.
- Provides a mechanism for deadlock handling.

What is a Process

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

	Entity of a process
1	Object Code- List of instructions for the CPU
2	Data – Data that needs to be processed by instructions
3	Resources – Memory, printer etc
4	Status : New, Ready, Running Waiting, Terminated

A process has four sections

S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

How do we create a process

We use the C-API

- **fork()** API for this process in Linux
- **CreateProcess()** on Windows

Live Example of creating process in Linux

Fork()

The fork() function is used to create a new process by duplicating the existing process from which it is called. The existing process from which this function is called becomes the parent process and the newly created process becomes the child process. As already stated that child is a duplicate copy of the parent but there are some exceptions to it.

Fork() has an interesting behavior while returning to the calling method. If the fork() function is successful then it returns twice. Once it returns in the child process with return value '0' and then it returns in the parent process with child's PID as return value. This behavior is because of the fact that once the fork is called, child process is created and since the child process shares the text segment with parent process and continues execution from the next statement in the same text segment so fork returns twice (once in parent and once in child).

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

int var_glb; /* A global variable*/
int main(void)
{
    pid_t childPID;
    int var_lcl = 0;
    childPID = fork();
    if(childPID >= 0) // fork was successful
    {
        if(childPID == 0) // child process
        {
            var_lcl++;
            var_glb++;
            printf("\n Child Process :: var_lcl = [%d], var_glb[%d]\n",
                                                      var_lcl, var_glb);
        }
else //Parent process
        {
            var_lcl = 10;
            var_glb = 20;
            printf("\n Parent process :: var_lcl = [%d], var_glb[%d]\n", var_lcl, var_glb);
        }
    }
    else // fork failed
    {
        printf("\n Fork failed, quitting!!!!!!\n");
        return 1;
    }

    return 0;
}
```

CPU Scheduling

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

- First Come First Serve(FCFS) Scheduling
- Shortest-Job-First(SJF) Scheduling
- Priority Scheduling
- Round Robin(RR) Scheduling
- Multilevel Queue Scheduling

First Come First Serve(FCFS) Scheduling

•Jobs are executed on first come, first serve basis.

•Easy to understand and implement.

•Poor in performance as average wait time is high.

Quick Reference : Queue in a super Market

Shortest-Job-First(SJF) Scheduling

•Best approach to minimize waiting time.

•Actual time taken by the process is already known to processor.

• A perfect implementation of this method is not difficult

Priority Scheduling

•Priority is assigned for each process.

•Process with highest priority is executed first and so on.

•Processes with same priority are executed in FCFS manner.

•Priority can be decided based on memory requirements, time requirements or any other resource requirement.

Round Robin(RR) Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preemptive processes.

Round Robin(RR) Scheduling

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preempted and other process executes for given time period.
- Context switching is used to save states of preemptive processes.

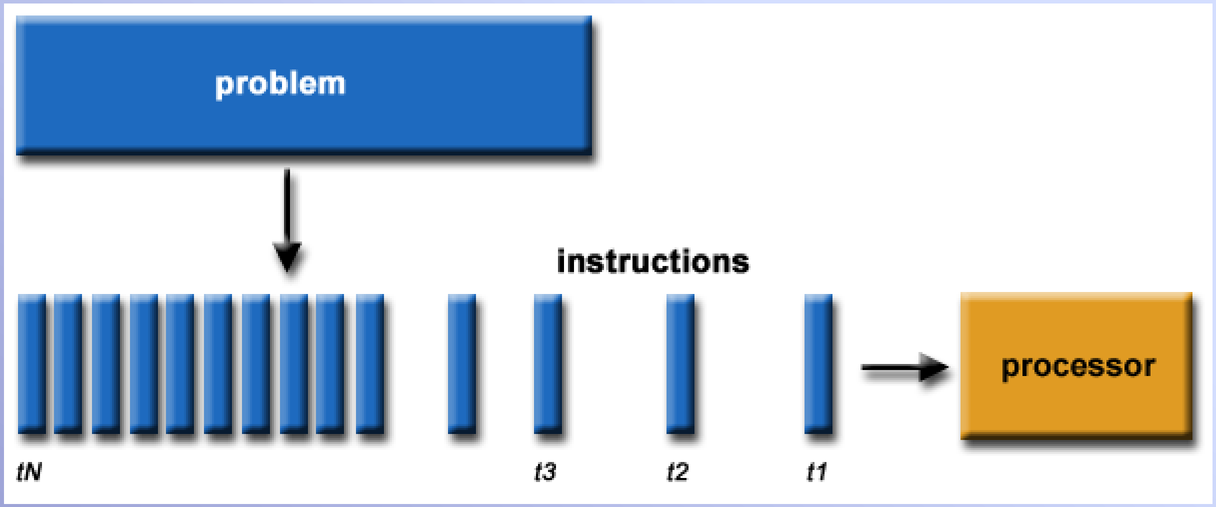
Multilevel Queue Scheduling

Multiple queues are maintained for processes.
Each queue can have its own scheduling algorithms.
Priorities are assigned to each queue.

Introduction to Parallel computing

Serial Computing

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time



Parallel Computing

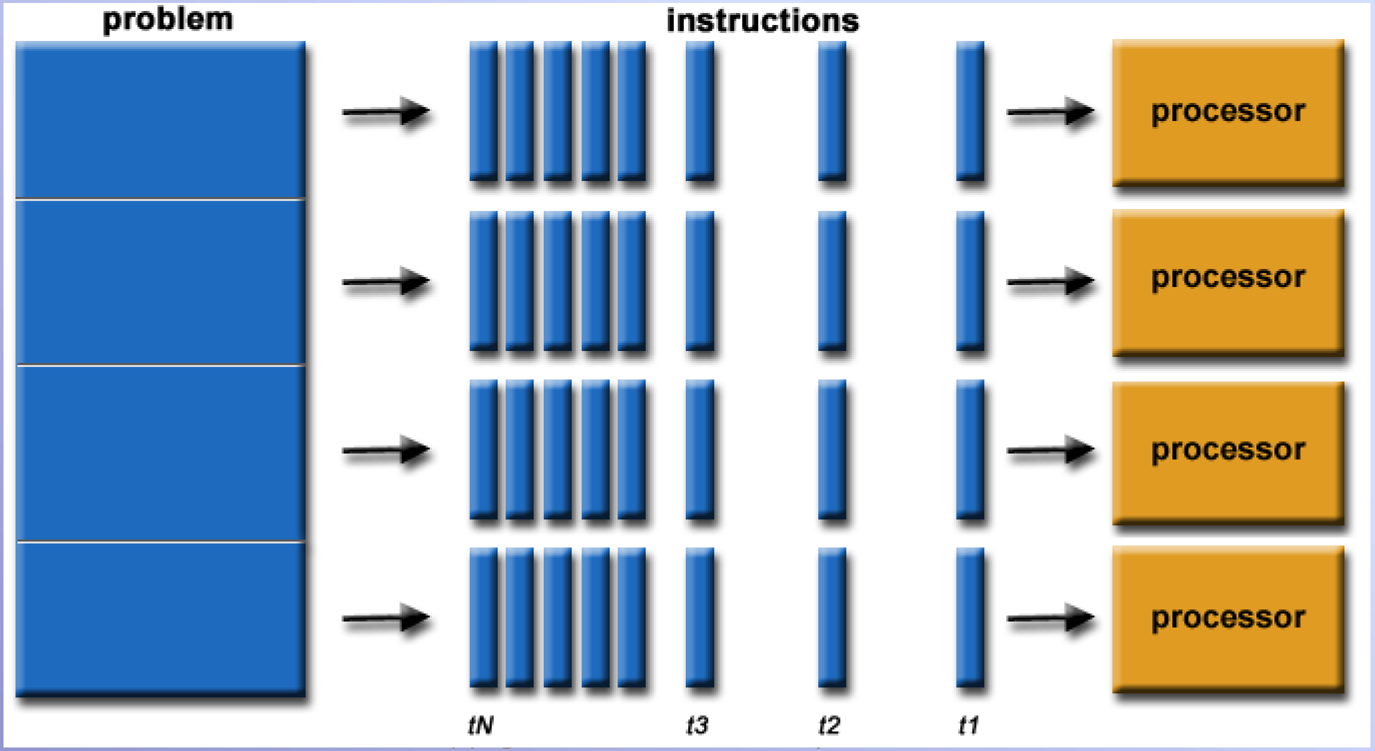
In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:

•A problem is broken into discrete parts that can be solved concurrently

•Each part is further broken down to a series of instructions

•Instructions from each part execute simultaneously on different processors

•An overall control/coordination mechanism is employed



What is a thread ?

A concept created towards multiprocessing

In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.

Thread is an execution unit which consists of its own program counter, a stack, and a set of registers. Threads are popular way to improve application through parallelism.

The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel.

As each thread has its own independent resource for process execution, multiple processes can be executed in parallel by increasing number of threads.

•The main program a.out is scheduled to run by the native operating system. a.out loads and acquires all of the necessary system and user resources to run. This is the "heavy weight" process.

•a.out performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.

•Each thread has local data, but also, shares the entire resources of a.out. This saves the overhead associated with replicating a program's resources for each thread ("light weight"). Each thread also benefits from a global memory view because it shares the memory space of a.out.

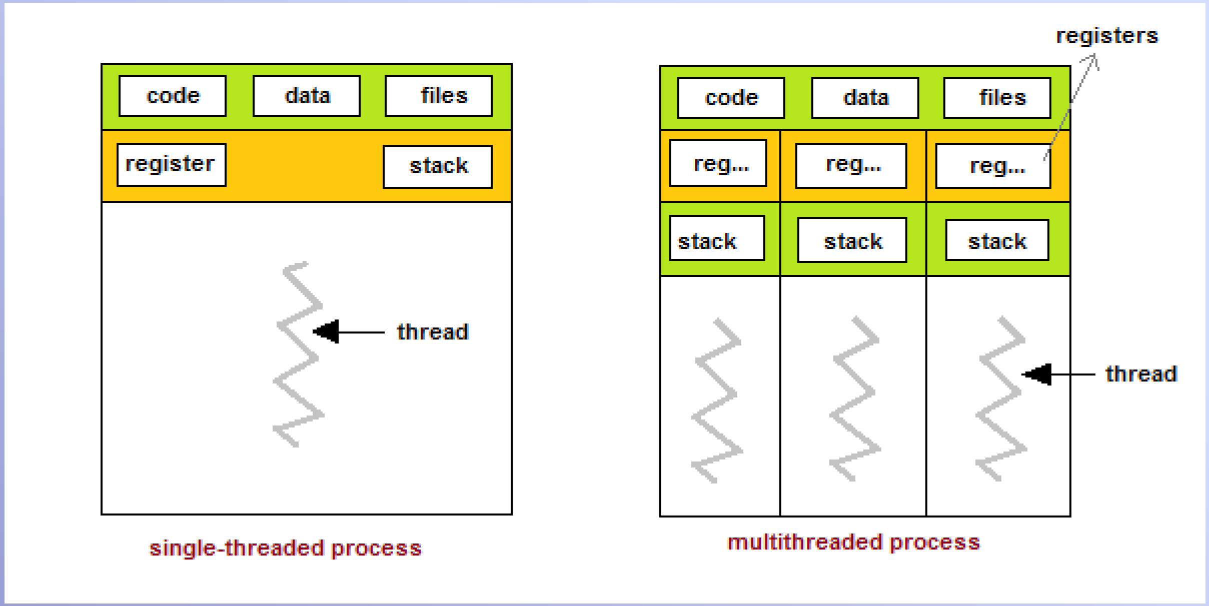
•A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.

•Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.

•Threads can come and go, but a.out remains present to provide the necessary shared resources until the application has completed.

Relation between process and thread

As each thread has its own independent resource for process execution, multiple processes can be executed in parallel by increasing number of threads.



Difference Between threads and processes

Processes do not share their address space while threads executing under same process share the address space.

From the above point its clear that processes execute independent of each other and the synchronization between processes is taken care by kernel only while on the other hand the thread synchronization has to be taken care by the process under which the threads are executing

Context switching between threads is fast as compared to context switching between processes

The interaction between two processes is achieved only through the standard inter process communication while threads executing under the same process can communicate easily as they share most of the resources like memory, text segment etc

Types of Threads

Threads

User threads, are above the kernel and without kernel support. These are the threads that application programmers use in their programs.

Kernel threads are supported within the kernel of the OS itself. All modern OSs support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously.

How do we create threads

Thread libraries provides programmers with API for creating and managing of threads.

Thread libraries may be implemented either in user space or in kernel space. The user space involves API functions implemented solely within user space, with no kernel support. The kernel space involves system calls, and requires a kernel with thread library support.

•POSIX Pitheads, may be provided as either a user or kernel library, as an extension to the POSIX standard.

•Win32 threads, are provided as a kernel-level library on Windows systems.

•Java threads - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on, i.e. either Pitheads or Win32 threads depending on the system

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <stdlib.h>

char n[1024];
sem_t len;

void * read1()
{
   while(1)
   {
        printf("\nThread-1:Enter a string\n");
        scanf("%s",n);
        printf("\n");
        sem_post(&len);
   }
}

void * write1()
{
      while(1)
      {
         sem_wait(&len);
         printf("\nThread-2:The string entered is :");
         printf("==== %s\n",n);
      }

}
```

```
int main()
{
        int status;
       pthread_t tr, tw;
       char *ptr ;
       const int x = 100;

       ptr = (char*) malloc(100);
       printf("Address of pointer variable  is %x ", &ptr);
       printf("Address on heap is %x ", ptr);
       printf("Address on const  is %x ", &x);
       printf("Address on global  is %x ", &len);
       pthread_create(&tr,NULL,read1,NULL);
       pthread_create(&tw,NULL,write1,NULL);

       printf("Pthread value = %d\n", tr);
       printf("Pthread value = %d\n", tw);
       pthread_join(tr,NULL);
       pthread_join(tw,NULL);
}
```

Why do we create threads

- Responsiveness

- Resource sharing, hence allowing better utilization of resources.

- Economy. Creating and managing threads becomes easier.

- Scalability. One thread runs on one CPU. In Multithreaded processes, threads can be distributed over a series of processors to scale.

- Context Switching is smooth. Context switching refers to the procedure followed by CPU to change from one task to another.

How Linux handles threads and process scheduling

•A process consists of a number of threads sharing the same address space.

•Kernel threads under Linux are implemented as processes that share resources.

•The scheduler does not differentiate between a thread and a process

•There is only one scheduler (which is the process scheduler) that schedules the "threads" (which are actually processes that share the same resources)

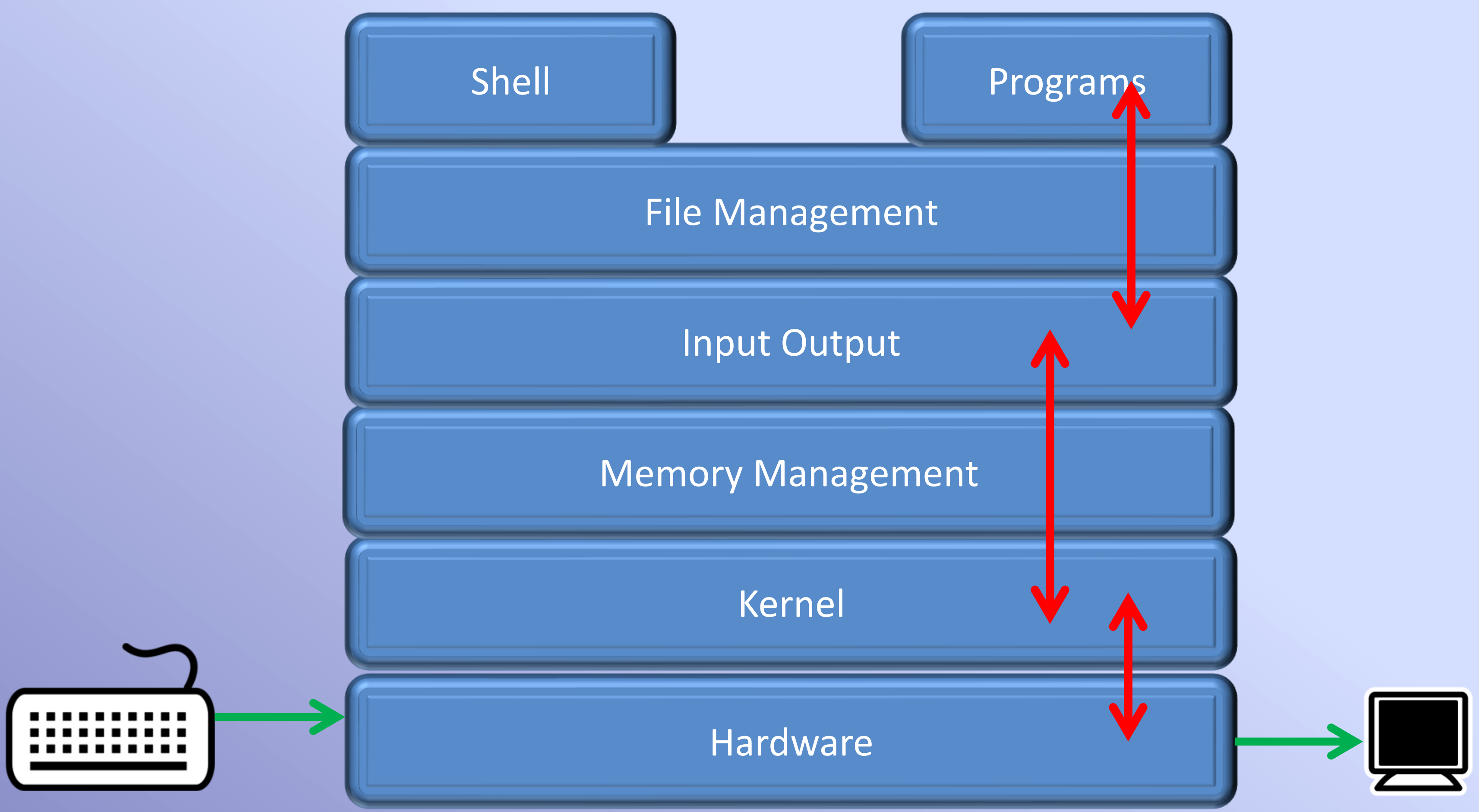
•Each thread has it's own stack and running execution state (registers etc.) just like "normal" processes. A context switch between two threads that belong to the same process just means that the switch might be a bit faster due to shared memory that may be still in the cache.

•To avoid simultaneous access of the same memory, the threaded program me has to use mutexes or locks - the scheduler does not help in this case.

What is SMP

SMP (symmetric multiprocessing) is the processing of programs by multiple processors that share a common operating system and memory. In symmetric (or "tightly coupled") multiprocessing, the processors share memory and the I/O bus or data path. A single copy of the operating system is in charge of all the processors. SMP, also known as a "shared everything" system, does not usually exceed 16 processors.

How does printf() and scanf() works



Raspberry Pi

Find out what type of processor & OS is used in Raspberry Pi and the

Inter Process Communication

Process need to exchange data for computing purpose for this most of the Operating systems has implemented the following methods

•Signals
•Pipe
•Sockets
•Message Queues
•Semaphore
•Shared Memory

Notes

How Linux handles threads and process scheduling

The Linux kernel scheduler is actually scheduling tasks, and these are either threads or (single-threaded) processes.

A process is a set of threads sharing the same address space.

Threads on Linux are kernel threads (in the sense of being managed by the kernel, which also creates its own threads), created by the Linux specific clone syscall (which can also be used to create processes on Linux). The **pthread_create**() function is probably built (on Linux) above clone inside NPTL and Gnu Libc (which integrated NPTL on Linux).

Kernel threads under Linux are implemented as processes that share resources. The scheduler does not differentiate between a thread and a process

There is only one scheduler (which is the process scheduler) that schedules the "threads" (which are actually processes that share the same resources)
Each thread has it's own stack and running execution state (registers etc.) just like "normal" processes. A context switch between two threads that belong to the same process just means that the switch might be a bit faster due to shared memory that may be still in the cache. To avoid simultaneous access of the same memory, the threaded program me has to use mutexes or locks - the scheduler does not help in this case.
http://stackoverflow.com/questions/8463741/how-linux-handles-threads-and-process-scheduling