UNIT-IV. Challenges in Multi-core Programming

Syllabus:

**Challenges in Multi-core Programming**

Sequential Models, Concurrency, Challenges for software development, New libraries for C++ developers, Processor Architecture Challenges, Operating systems (OS) roll in concurrent development: Consistent interfaces, Resource Management, OS interaction, Core OS services, Application Program Interfaces, Decomposition and Operating systems Roll, Hiding the Operating systems Roll: Abstraction and Encapsulation, Interface classes for POSIX API.

**Processes, Interface classes and predicates**


4.1 Sequential programming model

- In sequential programming model, the program instructions are executed one at a time.
- The program is viewed as a procedure, and each step involved in it is to be executed by the computer in the specific order and amount.
- Each step in this model is performed by the computer in specific order.
- The program in sequential model is broken into multiple tasks. Each task is performed in sequential-specific order and every task stands in queue and should wait it's turn.
- All these tasks are dependent on each other. If there are three tasks like Task1, Task2 and Task3. Execution of Task2 starts only when Task1 completes. Execution of Task3 starts only when Task2 completes. If Task1 fails, Then Task2 and Task3 can't starts execution.
- In this model, it can executes one task at a time. It has linear progression of tasks.
4.2 Concurrency

- Concurrent events are events that occur within the same time interval.
- When two or more tasks are executing over the same time interval then these tasks are said to be executing concurrently.
- The concurrent tasks do not mean that the tasks are executing exactly at the same instance.
- Example: Two tasks may execute concurrently within the same minute but with each task executing within different fractions of the minute. The first task may execute for first 10sec. Then for next 10sec task2 executes. Then again for next 10sec task1 executes and this process of execution alternately continues. So it appears that both tasks are executing simultaneously.
- Concurrent tasks can execute in a single or multiprocessing environment.
In **single processing environment** the concurrent tasks may execute at the same time period by the context switching. Context switching is the process of storing/copying the current state of execution of the process somewhere in memory and again after specified time interval bring the same for execution with CPU. So that multiple processes can share the same processor in a given period of time.

- In multiprocessor environment if enough number of processers are free then concurrent tasks may execute at the same instant of time on these processors.
- In multiprocessing environment the developers deal with the challenges of concurrency in terms of following three categories-
  1. Software development
  2. Software deployment
  3. Software maintenance

- Concurrent computing is related to but distinct from parallel computing, though these concepts are frequently confused, and both can be described as "multiple processes executing *at the same time". In parallel computing, execution literally occurs at the same instant, for example on separate processors of a multi-processor machine – parallel computing is impossible on a (single-core) single processor, as only one computation can occur at any instant (during any single clock cycle). By contrast, concurrent computing consists of process *lifetimes* overlapping, but execution need not happen at the same instant.

For example, concurrent processes can be executed on a single core by interleaving the execution steps of each process via time slices: only one process runs at a time, and if it does not complete during its time slice, it is *paused*, another process begins or resumes, and then later the original process is resumed. In this way multiple processes are part-way through execution at a single instant, but only one process is being executed at that instant.

Concurrent computations *may* be executed in parallel, for example by assigning each process to a separate processor or processor core, or distributing a computation across a network. This is known as task parallelism, and this type of parallel computing is a form of concurrent computing.

1. **Software Development**

Software development is carried out in various phases. All these phases are collectively called as Software Development Life Cycle (SDLC).

Phases of SDLC are as follows:

Website: [www.sandipwalunj.com](http://www.sandipwalunj.com)
1.1. Requirement Gathering/Requirement specification/Analysis

- Requirement specification is nothing but a kind of agreement between Company and Client.
- It contains what s/w must do and the constraints of software are specified.

1.2. Design

- From software requirement specification some model is developed called design.
- Design like DFD (Data Flow Diagram), E-R Diagram, Algorithms, Flowcharts, UML
  Diagrams(Class Diagram, Usecase Diagram etc.) are created.
- Design determines interior structure of the system.
- There are two approaches for the software design – Architectural design and detailed design.

1.3. Coding/Implementation

- Design is converted into programming code.

1.4. Testing

- Check for errors
- Check whether functioning is appropriate
- Improve quality

1.5. Maintenance

- This phase occurs after the delivery of the software.
- Modifications, correct faults, improve performance, adapt software to change environment.

Software development Methodologies:

1. Agile
2. Build and fix
3. Extreme Programming
4. Incremental
5. Object Oriented
6. Rapid Prototyping
7. Spiral
8. Structured
9. Waterfall

Website: www.sandipwalunj.com
4.3 Challenges for software development

The software developer is facing following challenges of concurrency. For concurrent or multicore programming we have to face following problems.

4.3.1 Software Decomposition

Challenge 1: There are many ways to represent a problem and it's solution.
Challenge 2: Making sure that the decomposition is complete, appropriate and correct.
Challenge 3: Finding the appropriate model for decomposition.

- Decomposition is the process of breaking down a problem statement or solution into smaller/basic parts.
- Problem and solution decompositions are normally performed throughout the analysis and design activities in the SDLC.
- Sometimes the parts are grouped into logical areas such as sorting, searching, calculating, reading, displaying etc.
- These parts are grouped by logical resource such as processor, database, communication and so on.
- Software decomposition is carried out using two approaches:

1. Work Breakdown Structure (WBS):
   a. It is basically used for procedural decomposition.
   b. What the particular piece of software do? is determined by Work Breakdown Structure (WBS).

2. Architectural Artifacts (AA):
   a. It represents Object Oriented Decomposition.
   b. What concepts or things a software solution is divided into are determined by AA.

<table>
<thead>
<tr>
<th>Sr.</th>
<th>Work Breakdown Structure (WBS)</th>
<th>Sr.</th>
<th>Artificial Artifacts (AA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>This is the software decomposition technique, in which the problem or solution is decomposed into number of tasks/procedures.</td>
<td>1</td>
<td>In this technique, problem/solution is decomposed into the set of objects.</td>
</tr>
<tr>
<td>2</td>
<td>Task driven/procedural model is used.</td>
<td>2</td>
<td>Object oriented/declarative model is used.</td>
</tr>
<tr>
<td>3</td>
<td>Scaling problem. It is difficult to scale large problem.</td>
<td>3</td>
<td>It can be easier and better for scaling complex systems.</td>
</tr>
</tbody>
</table>

- Decomposition is written in the form of Problem and Solution.
4.3.2. Task to Task communication:

- The two concurrently executing tasks can communicate with each other for acquiring the resources.
- If the concurrently executing tasks belong to the processes that lie on different operating systems then the communication among these tasks is called interprocess communication (IPC).
- There must have some mechanism to pass data from one task to another when they are executing concurrently.
- Managing interprocess mechanism: The POSIX i.e. Portable Operating System Interface for UNIX.

- Posix provides six basic mechanisms to accomplish communication between processes.

<table>
<thead>
<tr>
<th>Sr.</th>
<th>POSIX Interprocess Communication</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1   | Files with lock and unlock facility | - Data is maintained in a file.  
- Files are used to transfers data between two concurrent processes.  
- Provides locking facility to synchronize access file between concurrent processes. |
| 2   | Pipes | - Type of communication channel between related and unrelated processes.  
- Using read and write facility the file is access. |
| 3   | Shared Memory | - Data is stored in a memory which is accessible outside the address space of processes.  
- Common memory area is maintained.  
- Each process access data from common memory area. |
| 4   | Message queues | - Linklist of messages is maintained which is shared between processes. |
| 5   | Semaphore | - A variable used to achieve synchronization in accessing shared resource between different processes/threads. |
| 6   | Sockets | - Socket provides bidirectional communication between the processes.  
- To communicate it requires Port number and IP address. |

**Challenge 1:** IPC mechanism must be correctly created.

**Challenge 2:** These mechanism requires proper file and user permissions.
Challenge 3: It is difficult to understand correct source and destination processes.
Challenge 4: It is sensitive to correct size data sent and receive.
Challenge 5: Wrong data type or size can cause lockups and failures.

4.3.3. Concurrent access to data or resources by multiple tasks or agents
Due to concurrent access to shared resource/data following problems may occurs:

- **Data Race**: When more than one process executes concurrently and tries to update the data at the same time, the race condition occurs which is called as data race. It becomes unreliable.

- **Deadlock**: Deadlock is a situation in which two or more tasks/processes are waiting for the other to finish so that they can acquire the resource.

![Fig. Deadlock](image)

- **Indefinite Postpones**: If one or more tasks are waiting for some event or condition to occur and if that condition never occurs then it is known as indefinite postponement.

4.3.4. Identifying relationship between concurrently executing tasks

**Synchronization Relationship**:

There are four basic synchronization relationships

4.3.4.1 Start to start (SS) : one task can not start until another task starts.

4.3.4.2 Finish to Start (FS) : one task can not finish until another task starts.

4.3.4.3 Start to Finish (SF) : one task can not start until another task finishes.

4.3.4.4. Finish to Finish (FF) : one task can not finish until another task finishes.

**Timing Consideration:**

Website : [www.sandipwalunj.com](http://www.sandipwalunj.com)
Sometimes the synchronization relationships are specified along with the timing specification information.

While designing the synchronization relationships, time and events need to be considered.

4.3.5. **Controlling resource contention between tasks**

The resource contention occurs when multiple tasks compete for the use of same resource. Controlling such contention is one of the challenges in multiprogramming environment.

4.3.6. **Determining sufficient number of processes or threads**

Depending upon the type of applications the requirement for number of processors may vary. Number of threads and processors required are depending on the type of applications.

4.3.7. **Finding reliable and reproducible debugging and testing**

- Testing a sequential program is straightforward process. If we start from some input data, then the output or the flow of program can be easily predicted in sequential program.
- Testing of concurrent or parallel tasks become difficult because of several issues such as operating systems scheduling policies, dynamic workload, process or thread priorities and so on.

4.3.8. **Communicating the designs having multiprocessing components**

- There is a challenge for documenting the parallel design. There exists an UML(Unified Modeling Language) using which such designs can be represented.
- Following are the two types of the diagrams or models that are commonly used to represent the system architecture.

1. **Component diagram** : It is used to represent the dependancies and organization among a set of physical modules.

2. **Deployment diagram** : It shows runtime configuration of processing nodes, hardware and software components in a system.

4.3.9. **Implementing multiprocessing and multithreading in c++**

- C++ does not include any keyword primitives for parallelism.

  There is no way in c++ to support for execution of parallel statements.

Website : [www.sandipwalunj.com](http://www.sandipwalunj.com)
How to provide parallelism or multithreading in c++? We have to use different libraries they have provided. In modern programming environment there availability of high quality libraries that can be embedded with c++ in order to support for parallelism and distributed programming.

4.5 New libraries for C++ developers
To support parallelism in c++ they have provided two ways

1. System Libraries
   - System libraries are provided by operating system itself.
   - Example: POSIX thread library is used in conjunction with c++. It contains the set of system calls that support the parallelism.
   - POSIX:
     - Stands for Portable Operating System Interface for Unix.
     - It is a single UNIX specification.
     - The POSIX threads are included in the IEEE Std. 1003.1-2001 standard.
     - C++ developer can use POSIX to implement parallelism in C++ applications because it provides APIs(Application Programming Interfaces) and system libraries for creating threads and processes.
     - Using POSIX developer can use-
       1. POSIX threads
       2. POSIX Spawns functions
       3. The exec() family of functions.

2. User Level Libraries

4.5 Processor architecture challenges
   - There are four popularly used multi-core architectures.
     1. Opteron
     2. Cell
     3. UltraSparc T1
     4. Intel Core-2
   - Different compilers support multicore programming support such as GNU C++, Intel C/C++ compiler and SUN C/C++ compiler.
   - In order to use multicore features, developer has to be familiar with compiler and linker precise features.
Each compiler has its indivisual set of switches and directives which provides support to multiprocessing and multithreading.

Developer must know compiler and linker specific features.

In some cases example- The Cell processor, multiple types of compilers needed to generate single executable program. There is problem of getting non portable software.

The processor architecture facing main challenge that to take more advantage of multicore architecture without affecting the portability of software application.

Operating systems (OS) roll in concurrent development: Consistent interfaces, Resource Management, OS interaction, Core OS services, Application Program Interfaces, Decomposition and Operating systems Roll, Hiding the Operating systems Roll: Abstraction and Encapsulation, Interface classes for POSIX API.

4.6 Operating Systems (OS) roll in concurrent developement:

Operating System:

Definition:

"Operating System is a software program which acts as an interface between software and hardware of a computer system".

As per the application programmer and Sysetm programmer point of view the role of operating system is as follows:

- **Software interface**: Operating system provides reliable and definite interface to the hardware resources of the computer.
- **Resource Management**: Management of the hardware resources and other executing software applications and programs.

4.6.1 Consistent Interface

- The devices like printer, keyboards, disk drives, sound card, video adapters, monitor each having their own set of instructions.
- The same type of devices are designed by different manufacturers they may have their own set of instructions because of that it have different ways to access these devices.
- Earlier to invention of operating system concept, to use these devices developer must have knowledge of instruction sets of these devices.
- In order to work on these devices developer has to write device drivers codes.
- For same devices of different manufacturers, developer need to write different codes because their different set of instructions.
- Operating system provides the developer with common interfaces to similar devices.

Website: [www.sandipwalunj.com](http://www.sandipwalunj.com)
- Operating system provides the couple of software layers between developer program and hardwares of the computer system. These software layers are called as **API (Application Programming Interface)** or **System Programming Interface (SPI)**.

### 4.6.2 Resource management
- Operating system manages the access to various resources such as processors, memory, Input/Output devices.
- In multiprogramming environment, there are multiple programs executed or waiting for execution at same instant. The number of processor and memory is limited, it's operating systems job to manage the allocation of resources to various processes.
- Operating systems manages all the software and hardware resources.
- Operating system manages and schedules processes and threads.

### 4.6.3 The interaction between the operating system and the developer

![Diagram](image)

Fig. Operating Systems role

Website: [www.sandipwalunj.com](http://www.sandipwalunj.com)
<table>
<thead>
<tr>
<th>Levels</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Level 4** | - Highest level  
- In this level detail of parallel programming is not shown.  
- Example: Application framework used during parallel programming is STAPL(Standard Template Adaptive Parallel Library)  
- With STAPL application framework the developer can develop the c++ programs to achieve parallelism. |
| **Level 3** | - It is represented by template or class libraries like Intel Threading Building Blocks (TBB) library.  
- The TBB library is a set of high-level generic components which summarizes much of the detail of multiprocessing and multithreading.  
- Using TBB, developer invokes high level algorithm templates and objects. |
| **Level 2** | - In this level, developer makes use of the API provided by the operating system.  
- The most commonly used API in multiprocessing and multithreading environment is POSIX API.  
- It interacts with operating system for process and thread management.  
- Knowledge of process and thread management, interprocess communication, knowledge of operating system API’s related to process and thread management is required. |
| **Level 1** | - Lowest level.  
- Programming in this level has direct access to hardware with few or no software interaction.  
- For working in this level knowledge of operating system internals, hardware interfaces and kernel level interfaces is required. |

### 4.7 Core OS services

#### 1. Process Management

Operating system manages following things related to process

- Process execution
- Resource allocation to process
2. Memory management
- Allocation of memory
- De-allocation
- Proper utilization of memory

3. File System Management
- Files are stored on storage devices
- Data read/write/update/delete functionality is controlled by OS.

4. I/O Management
- OS manages Input/output operations to and from input/output devices.

5. Interprocess Communication Manager
  - Communication between different processes management

Processes, Interface classes and predicates

Website: www.sandipwalunj.com

**Multi-core and Multiprocessor**

- A multi-core is an architecture design that places multiple processors on single chip.
- Each processor is called core.
- This design is called as chip multiprocessors (CMP) because it allows for single chip multiprocessing.
- Various types of chip multiprocessors:
  a. Dual core: Two processors
  b. Quad core: Four processors
  c. Octa core: Eight processors
- Multi-core configuration

**Processes and Threads**

**Thread:**

- Thread is a light weight process having their own single path of execution.
- Threads share the same memory space of process that created them.

**Process:**

- A process is nothing but a program that is currently in executing.
- Process is an execution instatnce of an application.
- Process is a heavy weight task that requires its own address space.
- Operating system assigns the processor to each process.
- Every process executes on processor for given time slice i.e. Quantum.
- When process is preempted another process is given to processor.
- Operating system schedular switches code of one process to code of another process.
- Every process gets chance to execute their instructions using the processor.
- Two types of processes:

1. Kernel Process:

- Processes that executes operating system code is called kernel process. It also called as system processes.

Website: [www.sandipwalunj.com](http://www.sandipwalunj.com)
- These processes are used for administrating the whole system.
- Roles of these processes are as follows
  - allocating memory, checking devices

2. User Process:
- User processes are the processes which executes their own code.
- User processes sometimes calls the system processes.
- There are two modes in which user processes works.

1. User mode
- When user process executes its own code, the process is said to be in user mode.
- In user mode processes can not execute certain privileged m/c instructions.

2. Kernel Mode
- When user process makes a system function call such as read(), write() or open()- it is actually executing operating system instructions.
- During execution of operating system call the user process is put on hold until the call has completed.
- The kernel gets the processor to complete the execution of system call. Then the user processes are said to be in the kernel mode.
- The kernel processes cannot be preempted by any user process.

Process Control Blocks(PCB):
  - It stores information related to the processes.
  - This information describes process to the operating system.
  - The PCB is a part of heavy weight process.
  - Operating system uses information stored in PCB to control and manage every processes.
  - While executing the concurrent processes, the operating system switches between the processes the current state of executing process is saved in the process control block, so that the process can be resume it's state.
  - The PCB contains following information details:
    1. Current state of the process.
    2. Priority of the process.
    3. Pointers to allocated resources.
    4. Child process id, Calling process id, parent process id
    5. Pointers to location of process's memory.
7. Status and control registers.

Information in PCB is organized in following categories:

1. **Process Identification**
- Process id (PID), parent process id (PPID)

2. **Process State Information**
- Running process is stored in registers of CPU.
- At the time of context switch contents of current process is stored in registers and new process execution starts. If old process gains control of processor, the information is obtained to continue execution.
- Process state information contains the contents of user, control, status registers and stack pointer

3. **Process Control Information**
- This information is used by operating system in order to coordinate the currently active processes.
- Process control information is nothing but
  1. Information related to current state of the process.
  2. Priority of process.
  3. Allocated resources and memory
  4. Pointers to parent and child processes.
  5. Scheduling related information.
  6. Process privileges, messages and signals is also part of the process control information.

**Anatomy of process:**

**Address Space of processes:**
- The address space of processes is divided into three logical segments: Stack segment, Data segment and Text Segment.
- PCB, Stack segment, Data segment and PCB combinly gives process image.
- **Data Segment:**
  - Data segment stores various types of variables used by programs.
  - It includes global, external and static variables for the process.
- **Stack Segment:**
  - Local variables are allocated.
- Stores parameter passed to the functions.
- Stack segment maintains two stacks:
  - User stack: Stores data to handle the process running in user mode.
  - Kernel Stack: Stores data to handle the process running in kernel mode.

**Text Segment:**
- It contains instructions called by program code.
- It is present at the bottom of the address space

---

**Fig. Anatomy of process**

**Process Identification**

**Process State Information**

**Process Control Information**

**Kernel Stack**

**User Stack**

**Variables**

**Programming Instructions**

---

**Multitasking:**
Multitasking means performing more than one task at a time concurrently. We can achieve
multitasking in two ways – processes based and thread based multitasking.

**Multithreading:**
A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.

**What is difference between Thread and process?**

<table>
<thead>
<tr>
<th>Sr</th>
<th>Process</th>
<th>Sr</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Process is an heavy weight task which is currently in execution.</td>
<td>1</td>
<td>Thread is a light weight process.</td>
</tr>
<tr>
<td>2</td>
<td>Every process is having their own address space.</td>
<td>2</td>
<td>Every thread share the address space of a process that created them.</td>
</tr>
<tr>
<td>3</td>
<td>Program can be broken down into one or more processes.</td>
<td>3</td>
<td>A process creates one or more threads to accomplish it's task.</td>
</tr>
<tr>
<td>4</td>
<td>Operating system takes more time to create a process.</td>
<td>4</td>
<td>Operating system takes less time to create a thread.</td>
</tr>
<tr>
<td>5</td>
<td>Process-based multitasking is the feature that allows your computer to run two or more programs concurrently or at the same time.</td>
<td>5</td>
<td>Thread based multitasking is a single program can perform two or more tasks simultaneously.</td>
</tr>
<tr>
<td>6</td>
<td>Process-based multitasking allow you to run more than one program at a time i.e. you can run the Java compiler at the same time you are using a text editor.</td>
<td>6</td>
<td>E.g. A text editor can format text at the same time it is printing, here these two actions are being performed by two separate threads.</td>
</tr>
<tr>
<td>7</td>
<td>In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.</td>
<td>7</td>
<td>In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.</td>
</tr>
<tr>
<td>8</td>
<td>Multitasking processes require more overhead than multitasking threads.</td>
<td>8</td>
<td>Multitasking threads require less overhead than multitasking processes.</td>
</tr>
<tr>
<td>9</td>
<td>Processes are heavyweight tasks that require their own address space.</td>
<td>9</td>
<td>Threads are lightweight tasks that share same address spaces.</td>
</tr>
<tr>
<td>10</td>
<td>Inter process communication is expensive and limited.</td>
<td>10</td>
<td>Inter thread communication is inexpensive.</td>
</tr>
<tr>
<td>11</td>
<td>Context switching from one process to another is costly.</td>
<td>11</td>
<td>Context switching from one thread to the other is low cost.</td>
</tr>
</tbody>
</table>


Website : [www.sandipwalunj.com](http://www.sandipwalunj.com)
<table>
<thead>
<tr>
<th>12</th>
<th>Process, is a heavy weight</th>
<th>12</th>
<th>Threads are lightweight.</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Use of the CPU is reduce down, because idle time can be kept to a maximum.</td>
<td>13</td>
<td>Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.</td>
</tr>
</tbody>
</table>

**POSIX Function for handling the processes:**

<table>
<thead>
<tr>
<th>Sr</th>
<th>Purpose</th>
<th>POSIX Function</th>
</tr>
</thead>
</table>
| 1 | Create processes | posix_spawn()  
posix_spawnp() |
| 2 | Initialises attributes | posix_spawnattr_init() |
| 3 | For adding file actions | posix_spawn_file_actions_addclose()  
posix_spawn_file_actions_adddup2()  
posix_spawn_file_actions_addopen()  
posix_spawn_file_actions_destroy()  
posix_spawn_file_actions_init() |
| 4 | For Scheduling processes. | posix_spawnattr_setschedparam()  
posix_spawnattr_setschedpolicy()  
posix_spawnattr_getschedparam()  
posix_spawnattr_getschedpolicy()  
sched_setscheduler()  
sched_setparam() |
| 5 | For setting and retrieving attribute values. | posix_spawnattr_setsigndefault()  
posix_spawnattr_getsigndefault()  
posix_spawnattr_setsignmask()  
posix_spawnattr_getsignmask()  
posix_spawnattr_setflags()  
posix_spawnattr_getflags()  
posix_spawnattr_setpgroup()  
posix_spawnattr_getpgroup() |
| 6 | For destroying attributes. | posix_spawnattr_destroy() |

**How to create process:**

The `posix_spawn()` and `posix_spawnp()` functions shall create a new process (child process) from the specified process image. The new process image shall be constructed from a regular executable file called the new process image file.

**Syntax:**

```c
int posix_spawn(pid_t *pid, const char *path, const posix_spawn_file_actions_t *file_actions,const posix_spawnattr_t *attrp, char *const argv[], char *const envp[]);
```

```c
int posix_spawnp(pid_t *pid, const char *file, const posix_spawn_file_actions_t *file_actions,const
```

Website: [www.sandipwalunj.com](http://www.sandipwalunj.com)
posix_spawnattr_t *attrp, char *const argv[], char * const envp[]);

Arguments details:

**pid** : Represents the process id of newly created process.

**Path** : It is a argument of posix_spawn() function specifies path name that identifies the new process image file to execute.

**File** : The *file* parameter to posix_spawnp() shall be used to construct a pathname that identifies the new process image file. If the *file* parameter contains a slash character, the *file* parameter shall be used as the pathname for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable *PATH* (see the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables). If this environment variable is not defined, the results of the search are implementation-defined.

**Argv** : The argument *argv* is an array of character pointers to null-terminated strings. The last member of this array shall be a null pointer and is not counted in *argc*. These strings constitute the argument list available to the new process image. The value in *argv*[0] should point to a filename that is associated with the process image being started by the posix_spawn() or posix_spawnp() function.

**Envp** : The argument *envp* is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

**file_actions**: If *file_actions* is a null pointer, then file descriptors open in the calling process shall remain open in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see *fcntl*()). For those file descriptors that remain open, all attributes of the corresponding open file descriptions, including file locks (see *fcntl*()), shall remain unchanged.

If *file_actions* is not NULL, then the file descriptors open in the child process shall be those open in the calling process as modified by the spawn file actions object pointed to by *file_actions* and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed.

**return value**: Upon successful completion, posix_spawn() and posix_spawnp() shall return the process ID of the child process to the parent process, in the variable pointed to by a non-NULL *pid* argument, and shall return zero as the function return value. Otherwise, no child process shall be created, the value stored into the variable pointed to by a non-NULL *pid* is unspecified, and an error number shall be returned as the function return value to indicate the error. If the *pid* argument is a null pointer, the process ID of the child is not returned to the caller.

Website : www.sandipwalunj.com
Program : Write a program to create processes -

1. One process creates directory of specified name.

2. Second process displays path of current directory.

```c
#include <iostream>
#include <string.h>
#include <spawn.h>
#include <sys/wait.h>
using namespace std;

int main(void)
{
    int status;
    int pid1, pid2;
    char str1[]="mkdir";
    char str2[]="Sandip";
    char str3[]="pwd";
    char *arg1[]={str1,str2,(char*)0};
    char *arg2[]={str3,(char*)0};

    int val1 = posix_spawnp(&pid1,"/bin/mkdir",NULL,NULL,arg1,NULL);
    int val2 = posix_spawnp(&pid2,"/bin/pwd",NULL,NULL,arg2,NULL);

    cout<<"\n\n't Process ID : "<<pid1 " Status " <<val1;
    cout<<"\n\n't Process ID : "<<pid2 " Status " <<val2;

    wait(&status);
```
wait(&status);
return 0;
}

/*
Output:
[staff@Computer ~]$ g++ POSIX.cpp
[staff@Computer ~]$ ./a.out
    Process ID : 2370 Status 0
    /home/staff
    Process ID : 2371 Status 0
[staff@Computer ~]$ */

ps utility:
ps command is used for displaying the currently running processes along with their id's.

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>Process ID</td>
</tr>
<tr>
<td>PPID</td>
<td>Parent Process ID</td>
</tr>
<tr>
<td>UID</td>
<td>User name of process owner.</td>
</tr>
<tr>
<td>C</td>
<td>Processor utilization for scheduling.</td>
</tr>
<tr>
<td>PRI</td>
<td>Priority of the process. Higher value denotes lower priority.</td>
</tr>
<tr>
<td>ADDR</td>
<td>Address of the process.</td>
</tr>
<tr>
<td>NI</td>
<td>Nice value which is used in priority computation.</td>
</tr>
<tr>
<td>SZ</td>
<td>Size in blocks of the core image of the process.</td>
</tr>
<tr>
<td>WCHAN</td>
<td>The event for which the process is waiting or sleeping; if blank, the process is running.</td>
</tr>
<tr>
<td>TTY</td>
<td>The controlling terminal for the process.</td>
</tr>
<tr>
<td>STIME</td>
<td>Starting time of the process.</td>
</tr>
<tr>
<td>TIME</td>
<td>Total execution time of the process.</td>
</tr>
<tr>
<td>CMD</td>
<td>The command name and the argument.</td>
</tr>
<tr>
<td>S</td>
<td>State of the process</td>
</tr>
<tr>
<td>F</td>
<td>Flags</td>
</tr>
</tbody>
</table>

Website: www.sandipwalunj.com
There are some options using which we can control which processes can be displayed:

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-a</td>
<td>Write information about all the processes associated with terminal except session leader from the list.</td>
</tr>
<tr>
<td>-A</td>
<td>Write information of all processes.</td>
</tr>
<tr>
<td>-d</td>
<td>Write all processes except session leader.</td>
</tr>
<tr>
<td>-f</td>
<td>Generate full listing.</td>
</tr>
<tr>
<td>T</td>
<td>All the processes in this terminal are displayed.</td>
</tr>
<tr>
<td>A</td>
<td>All processes including those of other users.</td>
</tr>
<tr>
<td>R</td>
<td>Only running processes are displayed.</td>
</tr>
<tr>
<td>-l</td>
<td>Long format display.</td>
</tr>
<tr>
<td>-j</td>
<td>Jobs format displayed.</td>
</tr>
</tbody>
</table>

The STAT header displays status of process. The code used for it is described as given below:

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Uninterruptible sleep</td>
</tr>
<tr>
<td>R</td>
<td>Running or Runnable</td>
</tr>
<tr>
<td>S</td>
<td>Interruptible Sleep</td>
</tr>
<tr>
<td>T</td>
<td>Stopped, either by a job control signal or because it is being traced.</td>
</tr>
<tr>
<td>Z</td>
<td>Zombie process- that terminates but no with no parent.</td>
</tr>
</tbody>
</table>

sandip@sandip-HP-ProBook-4411s:~$ ps

```
PID TTY TIME CMD
4525 pts/0   00:00:00 bash
5220 pts/0   00:00:00 ps
```

sandip@sandip-HP-ProBook-4411s:~$ ps Tux
<table>
<thead>
<tr>
<th>USER</th>
<th>PID</th>
<th>%CPU</th>
<th>%MEM</th>
<th>VSZ</th>
<th>RSS</th>
<th>TTY</th>
<th>STAT</th>
<th>START</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>sandip</td>
<td>1998</td>
<td>0.0</td>
<td>0.1</td>
<td>517860</td>
<td>4108</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/bin/gnome-</td>
</tr>
<tr>
<td>sandip</td>
<td>2008</td>
<td>0.0</td>
<td>0.3</td>
<td>427576</td>
<td>10652</td>
<td>?</td>
<td>Ssl</td>
<td>09:25</td>
<td>0:00</td>
<td>gnome-session -</td>
</tr>
<tr>
<td>sandip</td>
<td>2054</td>
<td>0.0</td>
<td>0.0</td>
<td>12616</td>
<td>320</td>
<td>?</td>
<td>Ss</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/bin/ssh-ag</td>
</tr>
<tr>
<td>sandip</td>
<td>2057</td>
<td>0.0</td>
<td>0.0</td>
<td>24600</td>
<td>600</td>
<td>?</td>
<td>S</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/bin/dbus-l</td>
</tr>
<tr>
<td>sandip</td>
<td>2058</td>
<td>0.0</td>
<td>0.0</td>
<td>26472</td>
<td>2804</td>
<td>?</td>
<td>Ss</td>
<td>09:25</td>
<td>0:00</td>
<td>/bin/dbus-daem</td>
</tr>
<tr>
<td>sandip</td>
<td>2068</td>
<td>0.0</td>
<td>0.1</td>
<td>337392</td>
<td>3368</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/lib/at-spi</td>
</tr>
<tr>
<td>sandip</td>
<td>2072</td>
<td>0.0</td>
<td>0.0</td>
<td>24120</td>
<td>1712</td>
<td>?</td>
<td>S</td>
<td>09:25</td>
<td>0:00</td>
<td>/bin/dbus-daemo</td>
</tr>
<tr>
<td>sandip</td>
<td>2075</td>
<td>0.0</td>
<td>0.1</td>
<td>124828</td>
<td>3376</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/lib/at-spi</td>
</tr>
<tr>
<td>sandip</td>
<td>2083</td>
<td>0.0</td>
<td>0.6</td>
<td>838204</td>
<td>19232</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:01</td>
<td>/usr/lib/gnome-</td>
</tr>
<tr>
<td>sandip</td>
<td>2097</td>
<td>0.0</td>
<td>0.1</td>
<td>285592</td>
<td>4276</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/bin/pulsea</td>
</tr>
<tr>
<td>sandip</td>
<td>2106</td>
<td>0.0</td>
<td>0.1</td>
<td>196460</td>
<td>3172</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/lib/gvfs/g</td>
</tr>
<tr>
<td>sandip</td>
<td>2110</td>
<td>0.0</td>
<td>0.0</td>
<td>336776</td>
<td>3064</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/lib/gvfs/</td>
</tr>
<tr>
<td>sandip</td>
<td>2121</td>
<td>0.8</td>
<td>3.2</td>
<td>1343024</td>
<td>100336</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:36</td>
<td>compiz</td>
</tr>
<tr>
<td>sandip</td>
<td>2127</td>
<td>0.0</td>
<td>0.0</td>
<td>20260</td>
<td>964</td>
<td>?</td>
<td>S</td>
<td>09:25</td>
<td>0:02</td>
<td>syndaemon -i 1.</td>
</tr>
<tr>
<td>sandip</td>
<td>2134</td>
<td>0.0</td>
<td>0.0</td>
<td>178136</td>
<td>2756</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/lib/dconf/</td>
</tr>
<tr>
<td>sandip</td>
<td>2137</td>
<td>0.0</td>
<td>0.3</td>
<td>490180</td>
<td>9704</td>
<td>?</td>
<td>Sl</td>
<td>09:25</td>
<td>0:00</td>
<td>/usr/lib/gnome-</td>
</tr>
</tbody>
</table>
sandip 2138 0.2 1.1 1164680 34136 ? Sl 09:25 0:10 nautilus -n
sandip 2139 0.0 0.3 342672 9584 ? Sl 09:25 0:00 /usr/lib/policy
sandip 2140 0.0 0.6 760308 18496 ? Sl 09:25 0:00 nm-applet
sandip 2141 0.0 0.1 288032 5924 ? Sl 09:25 0:00 /usr/lib/gvfs/g
sandip 2142 0.0 0.1 58080 3740 ? S 09:25 0:00 /usr/lib/x86_64
sandip 2143 0.0 0.1 285532 3264 ? Sl 09:25 0:00 /usr/lib/gvfs/g
sandip 2144 0.0 0.0 191384 2736 ? Sl 09:25 0:00 /usr/lib/gvfs/g
sandip 2145 0.0 0.1 203672 3336 ? Sl 09:25 0:00 /usr/lib/gvfs/g
sandip 2146 0.0 0.5 440092 16520 ? Sl 09:25 0:00 /usr/lib/x86_64
sandip 2147 0.0 0.5 387248 15840 ? Sl 09:25 0:00 /usr/bin/gnome-
sandip 2148 0.0 0.1 498052 3736 ? Sl 09:25 0:00 /usr/lib/gvfs/g
sandip 2149 0.0 0.3 506108 11348 ? Sl 09:25 0:00 /usr/lib/x86_64
sandip 2150 0.0 0.0 4444 624 ? Ss 09:25 0:00 /bin/sh -c /usr
sandip 2151 0.0 0.4 341376 12852 ? Sl 09:25 0:00 /usr/bin/gtk-wi
sandip 2152 0.0 1.1 697652 36144 ? Sl 09:25 0:03 /usr/lib/unity/
sandip 2153 0.0 0.1 431084 4900 ? Sl 09:25 0:00 /usr/lib/x86_64
sandip 2154 0.0 0.0 270196 3056 ? Sl 09:25 0:00 /usr/lib/gvfs/g
sandip 2437 0.0 0.5 416200 16972 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2439 0.0 0.2 543088 6464 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2449 0.0 0.2 666776 8240 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2451 0.0 0.8 1033256 24808 ? SI 09:26 0:00 /usr/bin/python
sandip 2453 0.0 0.3 712232 9944 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2455 0.0 0.1 728908 5708 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2457 0.0 0.2 815464 6648 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2533 0.0 0.2 306176 9120 ? SI 09:26 0:00 /usr/bin/friend
sandip 2556 0.0 0.3 363784 9856 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2568 0.0 0.0 4444 628 ? S 09:26 0:00 sh -c /usr/lib/
sandip 2569 0.0 0.0 188520 2828 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2576 0.0 0.2 557324 7444 ? SI 09:26 0:00 /usr/lib/x86_64
sandip 2578 0.0 0.5 591572 18188 ? SI 09:26 0:00 /usr/bin/python
sandip 2607 0.0 0.3 513956 11748 ? SI 09:26 0:00 update-notifier
sandip 2648 0.0 0.1 442656 4132 ? SI 09:27 0:00 /usr/lib/x86_64
sandip 2792 0.0 0.1 209268 3252 ? SI 09:30 0:00 /usr/lib/libreo
sandip 2810 1.1 4.8 1412056 150544 ? SI 09:30 0:45 /usr/lib/libreo

Website : www.sandipwalunj.com
sandip  3181  0.0 0.0 120044 3072 ?  Sl  09:38  0:00 /usr/lib/gvfs/g
sandip  4516  0.0 0.0  4444  624 ?  Ss 10:17  0:00 /bin/sh -c gnom
sandip  4517  0.3 0.6 608220 19644 ?  Sl 10:17  0:03 gnome-terminal
sandip  4524  0.0 0.0 14836  820 ?  S  10:17  0:00 gnome-pty-helpe
sandip  4525  0.0 0.1  26196 3224 pts/0  Ss 10:17  0:00 bash
sandip  5221  0.0 0.0  22644 1308 pts/0  R+ 10:36  0:00 ps Tux

sandip@sandip-HP-ProBook-4411s:~$ ps -lf

F S UID   PID PPID  C PRI NI ADDR SZ WCHAN STIME TTY      TIME CMD

0 S sandip  4525  4517  0  80  0 - 6549 wait  10:17 pts/0 00:00:00 bash

0 R sandip  5222  4525  0  80  0 - 5661 -  10:36 pts/0 00:00:00 ps -l

sandip@sandip-HP-ProBook-4411s:~$ ps -l

F S UID   PID PPID  C PRI NI ADDR SZ WCHAN TTY      TIME CMD

0 S 1000  4525  4517  0  80  0 - 6549 wait  pts/0 00:00:00 bash

0 R 1000  5232  4525  0  80  0 - 3553 -  pts/0 00:00:00 ps

sandip@sandip-HP-ProBook-4411s:~$

Parent-Child relations:

/*

Prepared by : Sandip M. Walunj, Assistant Professor, Computer Engg. Dept., SITRC, Nashik. Website : www.sandipwalunj.com
Program : Write a program to create process and display child process and parent process ID's.

```c++
#include<iostream>
#include<unistd.h>
#include<spawn.h>
#include<sys/wait.h>
#include<string.h>
using namespace std;

int main()
{
    int status;
    pid_t pid;
    char str1[]="";
    char* arg1[]={str1,(char*)0};
    int ret = posix_spawnp(&pid,"/bin/ps",NULL,NULL,arg1,NULL);
    cout<<"\n\t Child Process Id : "<<pid<<"\n";
    cout<<"\n\t Calling Process Id : "<<getpid()<<"\n";
    cout<<"\n\t Parent Process Id : "<<getppid()<<"\n";
    wait(&status);
    return 0;
}
```
Child Process Id : 5403
Calling Process Id : 5402
Parent Process Id : 4477

<table>
<thead>
<tr>
<th>PID</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>4477</td>
<td>pts/1</td>
<td>00:00:00</td>
<td>bash</td>
</tr>
<tr>
<td>5402</td>
<td>pts/1</td>
<td>00:00:00</td>
<td>a.out</td>
</tr>
<tr>
<td>5403</td>
<td>pts/1</td>
<td>00:00:00</td>
<td>ps</td>
</tr>
</tbody>
</table>

Process States:

Throughout the lifetime processes enters in the number of different states. These states are as follows:

1. Ready or Runnable
2. Running
3. Waiting or Blocked or sleeping
4. Stopped
5. Zombie
When process is created then it is ready to execute their instructions.

Process has to wait for processor to get available.

Each process is allowed to execute their task for given time slice or quantum.

The processes that are waiting for the processors are placed in the ready queue.

The processes that are in the ready queue are called runnable processes.

When processor becomes available, the processes in the ready queue can gets the processor to execute their task. The process which is using the processor is called **running state** of that process. When the time slice of the running process is run out i.e. time out, then processor is removed from it and that process is again sent in the ready queue. Then new process in the ready queue is selected for running.

System processes are not preempted. When system processes gets CPU/Processor they run...
up to completion. The preempted processes are stored in ready queue.

- If process time slice is not complete and it doesn't continue to execute then it voluntarily gives up the processor. If the running process waits for certain event to occur or makes request to access I/O devices then it goes in the **sleeping** state. When expected event occurs or I/O request satisfies then that process goes into **runnable state**.

- When running process get signal to stop it enters in the Stopped state. When process receives signal to continue, process enters in the ready state. Process enters into the stopped state when being debugged or some situation has occurred.

- When process executes all its instructions, it normally exit the system. The process is then removed from process table. The PCB entry of the process is destroyed. Resources allocated to the process are deallocated and return back to the pool of available resources.

- The process which is unable to execute and cannot exit the system is called the zombie process. When process table contains too many zombie processes then the performance of the system gets degraded.

<table>
<thead>
<tr>
<th>Sr.</th>
<th>State Transition</th>
<th>Circumstances</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ready to Running</td>
<td>Dispatch</td>
<td>Processor currently executes teh process.</td>
</tr>
<tr>
<td>2</td>
<td>Running to Ready</td>
<td>Time runout</td>
<td>- Time slice or quantum is overed.</td>
</tr>
<tr>
<td></td>
<td>Running to Ready</td>
<td>Preempt</td>
<td>If any higher priority process want processor for their execution then currently running process is preempted i.e. Remove from running state to runnable/Ready state before their time slice run out.</td>
</tr>
<tr>
<td></td>
<td>Running to Sleeping</td>
<td>Block</td>
<td>Process gives up processor before time slice ruout when it waits for specific event to occur or I/O request fulfilled.</td>
</tr>
<tr>
<td></td>
<td>Running to Stopped</td>
<td>Stop signal</td>
<td>When process receives signal to stop, it gives up processor.</td>
</tr>
<tr>
<td></td>
<td>Running to Zombied</td>
<td>Terminated</td>
<td>The processs has been terminated and is waiting for the parent process to retrieve its exit status.</td>
</tr>
<tr>
<td></td>
<td>Running to Exit</td>
<td>Execution is over</td>
<td>When process completes execution of their all instructions. The process has been terminated and the parent process gets the exit status and the process exits the system.</td>
</tr>
<tr>
<td>3</td>
<td>Sleeping to Ready</td>
<td>unblock</td>
<td>The event for which process was waiting is occured. Or I/O request is fulfilled then process placed back in ready queue.</td>
</tr>
<tr>
<td>4</td>
<td>Stopped to Ready</td>
<td>Continue signal</td>
<td>The process has to receive the signal to continue then</td>
</tr>
</tbody>
</table>
Process Scheduling:

- The role of scheduler is to decide which process should be assigned to a processor first.
- The processes that are ready for execution are placed in the priority queue.
- Each process is given a priority class and placed in a priority queue. All the runnable processes with the same priority are placed in the same priority queues.
- There are number of priority queues, each one representing a different priority class. These priority queues are used by multilevel priority queues.
- The scheduler assigns the process having highest priority that means process at the head/start in that priority class to the processor.

Scheduling Policies:

The POSIX API provides following two scheduling policies:

**1. First In First Out (FIFO)**
- Processor is assigned to the processes as per their arrival time in the queue.
- When running process time slice has expired, it is placed at the head of it's priority queue.
- When sleeping process become runnable, the process is placed at the end of it's priority
A process can make a system call and give up the processor to another process with the same priority level. The process is then placed at the end of its priority queue.

2. Round Robin

- In round robin scheduling, all processes are considered as equal.
- Round Robin scheduling policy allocates processes using FIFO scheduling, but when the time slice run out, the process is placed at the back of the ready queue.
- When running process time slice has expired, it is placed at the back of the queue. And next process in the queue is assigned the processor.
- Round Robin (RR) scheduling policy allocates processors using FIFO scheduling, but when the time slice expires, the process is placed at the back in the ready queue.

Process Priorities:

- Each process has a nice value that is used to calculate the priority level of the calling process.
- The process inherits the priority of the process that creates it.
- Priority of a process can be lowered by raising its nice value.
- Only superuser and kernel processes raise the priority levels.
- The priority level of the process can be changed using nice() function.
- The nice() function returns new nice value of the process, on being unsuccessful the function will return -1.
- setpriority() function is used to set priority of a processes.
- Processes have two types of priorities

1. Static Priority
   - Static priority of a process cannot change.

2. Dynamic Priority
   - Dynamic priority of a process can be changed.
   - Priority value can be adjusted to appropriate value.

Syntax:

```c
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int value);
```

when return value,
0 = successful operation
-1 = unsuccessful operation

here,
which argument have following possible arguments:
PRIO_PROCESS : Identifies who argument as a process id
PRIO_PGRP : Identifies who argument as a process group Id.

Website: www.sandipwalunj.com
PRIO_ISER : Identifies who arguments as a user ID.

The range of nice value is in the range of -20 to 19.

#include<iostream>
#include<unistd.h>
#include<sys/resource.h>

using namespace std;

int main()
{
  int pid=2;
  int value=10;
  int nice_val;
  int return_value;

  nice_val = getpriority(PRIO_PROCESS,pid);
  cout<<"\n\n\t Nice value is : "<<nice_val<<"\n";

  return_value = setpriority(PRIO_PROCESS,pid,value);
  cout<<"\n\n\t Nice value is : "<<return_value<<"\n";

  nice_val = nice(18);
  cout<<"\n\n\t Nice value : "<<nice_val;
  return 0;
}

/*

output:

Website : www.sandipwalunj.com
sandip@sandip-HP-ProBook-4411s:~$ g++ ProcessPriority.cpp

sandip@sandip-HP-ProBook-4411s:~$ ./a.out

Nice value is : 0

Nice value is : -1

Nice value : 18

sandip@sandip-HP-ProBook-4411s:~$ */

Context Switch:

- Processes which are ready to execute and waiting for processor are maintain in the ready queue.
- When processor becomes available, the it is assigned for process for particular time slice.
- When time slice of currently running process is time out then context switch occurs the operating system saves the context of currently running process and when that process gets the processor back this context is used by the process to resume it's work.
- During the context switch,
  1. PCB of preempted process is updated.
  2. The process state field is updated.
  3. The contents of processor registers, state of the stack, user and process identification and privileges are updated.
- Context switch occurs when
  1. Process voluntarily gives up the processor.
  2. Process is preempted
  3. Process changes from user mode to kernel mode.
  4. when process waits for particular event to occur or process is requesting for Input/output device.

Activities in process creation:

Following are the activities need to handle when new process is created:

1. New entry is made in the main process table.
2. The new PCB(Process Control Block) is created.

Website: [www.sandipwalunj.com](http://www.sandipwalunj.com)
3. Initialization of PCB required.
4. PCB contains unique process ID and parent process ID.
5. Program counter is set to point to the program entry point.
6. System stack pointers are set.
7. Process is initialized with requested attributes.
8. If process has not given any priority value then it is set to default priority value.
9. Process is placed in ready queue.
10. The address space is allocated for created process.

The fork() function call:

fork():
- The fork() function call is used to create the process.
- When program calls fork() then a duplicate process called child process is created.
- The parent process continues to executing the program and at the same time child process also continues its execution.
- The return value of the parent process is the child process ID.
- Return value of the child process is 0. With fork call two processes come with two different values.
- fork is an operation whereby a process creates a copy of itself.
- It is a system call implemented in kernel.
- System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a **new** process, which becomes the **child** process of the caller. After a new child process is created, **both** processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():
  - If fork() returns a negative value, the creation of a child process was unsuccessful.
  - fork() returns a zero to the newly created child process.
  - fork() returns a positive value, the **process ID** of the child process, to the parent. The returned process ID is of type **pid_t** defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process.
Program 1:

```cpp
using namespace std;

int main(void)
{
    pid_t pid;

    pid = fork();

    if (pid == -1)
    {
        /*
         * When fork() returns -1, an error happened.
         */
        perror("fork failed");
    }
    else if (pid == 0)
    {
        /*
         * When fork() returns 0, we are in the child process.
         */
        cout<<"Hello from the child process!\n";
        cout<<"\n\tParent process ID : "<<(int)getpid();
        cout<<"\n\tChild process ID : "<<(int)pid;
    }
}
```
else
{
    /*
    * When fork() returns a positive number, we are in the parent process
    * and the return value is the PID of the newly created child process.
    */
    cout<<"\n\t Child process ID : "$<<(int)getpid();
}
cout<<"\n";
return 0;
}
/*
Output:
[staff@Computer ~]$ g++ Fork1.cpp
[staff@Computer ~]$ ./a.out

    Child process ID : 4636

Hello from the child process!

    Parent process ID : 4637
    Child process ID : 0

[staff@Computer ~]$ */
Program 2:

```c
#include <sys/types.h> /* pid_t */
#include <sys/wait.h> /* waitpid */
#include <stdio.h>    /* printf, perror */
#include <stdlib.h>   /* exit */
#include <unistd.h>   /* _exit, fork */
#include <iostream>
using namespace std;

int main(void)
{
  pid_t pid;

  pid = fork();

  if (pid == -1) {
    /*
     * When fork() returns -1, an error happened.
     */
    perror("fork failed");
    exit(EXIT_FAILURE);
  }
  else if (pid == 0) {
    /*
     * When fork() returns 0, we are in the child process.
     */
```
cout<<"Hello from the child process!\n";
_exit(EXIT_SUCCESS); /* exit() is unreliable here, so _exit must be used */
}
else {
    /*
    * When fork() returns a positive number, we are in the parent process
    * and the return value is the PID of the newly created child process.
    */
    int status;
    (void)waitpid(pid, &status, 0);
}
return EXIT_SUCCESS;
}
/*
Output:
[staff@Computer ~]$ g++ Fork.cpp
[staff@Computer ~]$ ./a.out
Hello from the child process!
[staff@Computer ~]$ *
*/

Program 3

#include <stdio.h>
#include <sys/types.h>
#define MAX_COUNT 200

void ChildProcess(void);          /* child process prototype */
void ParentProcess(void);         /* parent process prototype */

Website : www.sandipwalunj.com
void main(void)
{
    pid_t pid;
    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}

void ChildProcess(void)
{
    int i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from child, value = %d\n", i);
    printf("*** Child process is done ***\n");
}

void ParentProcess(void)
{
    int i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}

The exec() family :

⇒ The exec family of functions replaces the calling process with a new process.
⇒ The fork() call duplicates the process whereas when exec() call immediately stop executing
    that process and begins execution of new process from the beginning.
⇒ In short, exec family will initiate the program within the program.
⇒ The new process created by an exec function is an executable file and can immediately
    executable.
⇒ If function is unsuccessful, it returns -1 to the calling process.

Exce() fails during following situations:
1. File does not exists.
2. File is not exectable.
3. Permissions denied such as search or executable.
4. Problems with symbolic links.

Program:
#include <sys/types.h> /* pid_t */

Website: www.sandipwalunj.com
```cpp
#include <sys/wait.h>  /* waitpid */
#include <stdio.h>     /* printf, perror */
#include <stdlib.h>    /* exit */
#include <unistd.h>    /* _exit, fork */
#include <iostream>

using namespace std;

int main(void)
{
    pid_t pid;

    pid = fork();

    if (pid == -1) {
        /*
         * When fork() returns -1, an error happened.
         */
        perror("fork failed");
        exit(EXIT_FAILURE);
    }

    else if (pid == 0) {
        /*
         * When fork() returns 0, we are in the child process.
         */
        cout<<"Hello from the child process!\n";
        execl("/bin/ls","ls","",(char*)0);
```
else {
    /*
    * When fork() returns a positive number, we are in the parent process
    * and the return value is the PID of the newly created child process.
    */

}

return 0;
}

execl(), execle() and execlp():
These functions pass the command line argument as a list.

execv():
The family of execv(), execve() and execvp() functions pass command line arguments in a vector of
pointers to null terminated strings.

Process Environmental variables:

⇒ When program is executed, it receives information about the context in which it was
   invoked by the two ways.
1. By command line argument variables passed to main function : argc, argv[]
2. Environmental variables
   ⇒ The standard environmental variables used for getting information about the user's
      - Home directory
      - terminal Type
      - Current locale
     ⇒ These variables are also used to transfer any user-defined information between parent and
        child process.
     ⇒ System environment variables are initialized by start-up files.
     ⇒ These are common system variables.

⇒ Environment variables are as:

$HOME : Absolute path name representing home directory.

Website : www.sandipwalunj.com 44
$PATH   : List of directories to search for commands.
$USER   : Your user-id.
$SHELL  : The absolute path name of your login shell.
$TERM   : The terminal Type.
$TZ     : It specifies the time zone.
$MAIL   : The absolute path of your mailbox.

Functions:
1. Char* getenv(const char *name) : This function returns the string that is the value of the environment variable(name)
2. int setenv(const char *name, const char *value, int replace) : It is used to set different values to environment variables.
3. int unsetenv(const char *name) : It is used to remove some environment variable specified by name.

Using system() to spawn() processes:

system() :
➔ It is used to execute the command or executable program.
➔ It causes execution of fork(), exec() and shell.
➔ System() function executes a fork() and a child process calls an exec() with shell that executes the given command or program.
➔ Using this function one can execute the command from within the program.
➔ System() function creates subprocess which is running the standard Bourne shell(/bin/sh) and then hands over the command to execute on the shell for execution.
➔ Need to include header file stdlib.h

Syntax:
int system(const char *string);
The string parameter can be a system command or the name of an executable file.

Example:
#include<stdio.h>
int main()
{


Website : www.sandipwalunj.com
int x = system("ps");
return x;
}

Killing process:
When a process is terminated, the PCB is erased and the address space and resources used by the
terminated process are de-allocated.

Kill():
- Kill function is used to terminate any target process.
- Pid is passed as a parameter to the process.
- Pid denotes the id of the process that is to be killed.
- Another parameter is Sig, it indicates the signal that can be sent to the specified process.
- Kill() function is used to cause the termination of another process.
- Kill process send signal to the process or processes specified by parameter pid.

Syntax:
```c
int kill(pid_t pid, int sig)
```

- need to include following header files
  ```c
  #include <sys/types.h>
  #include <signal.h>
  
  int kill(pid_t pid, int sig)
  ```

**DESCRIPTION**

The `kill()` function sends the signal given by `sig` to `pid`, a process or a group of processes. `Sig`
may be one of the signals specified in `sigaction(2)` or it may be 0, in which case error checking is
performed but no signal is actually sent. This can be used to check the validity of `pid`.

For a process to have permission to send a signal to a process designated by `pid`, the real or
effective user ID of the receiving process must match that of the sending process or the user must
have appropriate privileges (such as given by a set-user-ID program or the user is the super-user).

A single exception is the signal SIGCONT, which may always be sent to any descendant of the
current process.

If `pid` is greater than zero:
- `Sig` is sent to the process whose ID is equal to `pid`.

If `pid` is zero:
- `Sig` is sent to all processes whose group ID is equal to the process group ID of the sender,
and for which the process has permission; this is a variant of `killpg(2)`.  

Website: www.sandipwalunj.com
If \( \text{pid} \) is -1:

If the user has super-user privileges, the signal is sent to all processes excluding system processes (with P_SYSTEM flag set), process with ID 1 (usually `init(8)`), and the process sending the signal. If the user is not the super user, the signal is sent to all processes with the same uid as the user excluding the process sending the signal. No error is returned if any process could be signaled.

For compatibility with System V, if the process number is negative but not -1, the signal is sent to all processes whose process group ID is equal to the absolute value of the process number. This is a variant of `killpg(2)`.

**RETURN VALUES**

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and `errno` is set to indicate the error.

**exit() and abort():**

There are two functions a process can call for self termination `exit()` and `abort()`.

**Exit() :**

- This function causes normal termination of the calling process.
- All open file descriptors associated with the process will be closed.
- The function flushes all open streams.
- The value of status can be 0, EXIT_FAILURE or EXIT_SUCCESS.
- Return value 0 means file is terminated successfully.

**Syntax:** `void exit(int status)`

**abort():**

- `abort()` function causes an abnormal termination of the calling process.

**Syntax:** `void abort(void)`

**Process Resources**

There are three types of process resources.

**Types of resources:**

1. **Hardware Resources**

   Hardware resources are physical resources that are connected to the computer.
   Example: Processor, Memory, I/O devices – printer, scanner, monitor, keyboard etc.
   All these devices can be shared by different processes.

2. **Software resources**

   Example: Shared libraries.
   Shared library provides common set of services of functions to processes.
Processes can shared applications, programs and utilities.
In such cases only copy of the program is brought into memory.
There are separate copies of data for each program.

3. Data Resources
Data resources are objects, system data such as environment variables, files, handles, global variables are all shared resources and modified by processes.

Setting resource limit:
- In some situations it needs to set limit for resource use by the process.
- POSIX library provides some functions for setting limit on the utilization of the system resources.
- The limitation is on following factors.
  1. Size of file and core file creation.
  2. Amount of memory usage.
  3. Amount of CPU usage.
  4. Number of open file descriptors.
  5. Size of process stack.
POSIX functions:

\[
\text{int setrlimit(int resource, const struct rlimit *rlp);} \\
\text{int getrlimit(int resource, struct rlimit *rlp);} \\
\text{int getrusage(int who, struct rusage *r_usage);} \\
\]

Some constants and their meaning:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIM_INFINITY</td>
<td>There is no limit.</td>
</tr>
<tr>
<td>RLIM_SAVED_MAX</td>
<td>It is unrepresentable and saved hard limit.</td>
</tr>
<tr>
<td>RLIM_SAVED_CUR</td>
<td>It is unrepresentable and saved soft limit.</td>
</tr>
<tr>
<td>RLIM_INFINITY</td>
<td>The hard and soft limit can be set with this constant which means the resource is unlimited.</td>
</tr>
</tbody>
</table>

The parameter resource of the getrlimit and setrlimit functions can be controlled with the help of following values.

<table>
<thead>
<tr>
<th>Resource Definition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLIMIT_AS</td>
<td>Maximum size of process's total available memory in bytes.</td>
</tr>
<tr>
<td>RLIMIT_CORE</td>
<td>Maximum size of core file. When value is 0 then no core dump file is created.</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>RLIMIT_CPU</td>
<td>CPU time limit in seconds that can be used by the process.</td>
</tr>
<tr>
<td>RLIMIT_DATA</td>
<td>Maximum size of the process's data segment in bytes.</td>
</tr>
<tr>
<td>RLIMITFSIZE</td>
<td>Maximum size of files that the process may create.</td>
</tr>
<tr>
<td>RLIMIT_NOFILE</td>
<td>Specifies the value one greater than the maximum file descriptor number that can be opened by this process.</td>
</tr>
<tr>
<td>RLIMIT_NPROC</td>
<td>Maximum number of process's that can be created for the particular user ID of the calling process can be limited.</td>
</tr>
<tr>
<td>RLIMIT_STACK</td>
<td>Maximum size of processes stack in bytes.</td>
</tr>
</tbody>
</table>

**Asynchronous processes:**

- Asynchronous processes execute independent of each other.
- Two processes run until complete without disturbance about each other.
- Asynchronous processes may or may not have parent child relationship.
- **With parent child relationship:**
  - If one process creates another process then both run independently but at some point of time the parent process gets the exit status of the child process.
- **Without Parent child relationship:**
  - If processes do not have parent child relationship they can share the same parent.
  - Asynchronous processes can execute serially, simultaneously or may overlap.
  - Using fork(), fork-exec() and posix_spawn() functions the asynchronous processes can be created.

**Fig. Asynchronous Processes**
Synchronous processes:

➢ Synchronous processes are those processes that have interleaved execution or execution along with time synchronization.

➢ Example: process1 and process 2 are synchronous processes. Process1 is parent process that creates process2. When process 2 is running the process 1 suspends it's execution and on return of exit code from process 2 then process 1 resumes its execution.

➢ Using system() function the synchronous process can be created. A shell is created within which the system command executes the submitted command. The parent process is suspended until the child process terminates and system() call returns.

\[
\text{Fig. Synchronous Processes}
\]

wait() :

➢ Wait() function suspends the execution of the process untill a child process terminates.

➢ After the child process terminates, the parent process collects the exit status.

➢ Syntax:

\[
\text{pid_t wait(int *status)}
\]

\[
\text{pid_t waitpid(pid_t pid,int *status, int options)}
\]

Predicates, processes and interface classes

➢ The predicate is a function object that returns boolean value.

➢ The predicate is a statement that can be evaluated to true or false.

➢ The predicate gives declarative interpretation to a sequence of actions.
References: