Process and Threads

Process & threads:

Process creation: Posix provides various APIs for process creation. Applications designed to execute concurrent tasks and applications designed for user interaction generally need such APIs

Concurrent program applications:

- Applications designed to execute parallel task are referred as concurrent programs.
- Such programs start with single thread of execution and have the ability to dynamically start a parallel context.
- Concurrent applications are designed to achieve either of the following objectives.
    1) Application of concurrency to achieve primary functionality of the software (multi threading).
    2) Application of concurrency to achieve faster execution and better utilization of CPU (parallel programming)

Case study to better explain concurrency (a game software)

Description:
The balloon shooter game, displays set of balloons on the console and the action key is configured to shoot the balloons, depending on the result of the action event the score is either incremented or decremented.
The following modules needs to be implemented to write this software

1) User Interface
2) Graphics
3) Action
4) Score

Windows version pseudo code

```
    int hit =0;
    int inc = 0;
    int  main()
{
    ui(();
    createthread(display);
    createthread(action);
    createthread(score);
}
```

- Create thread is a windows API that starts a parallel task with a function provided as argument.
- Task created using createthread() is called user level threads.
- Each user level thread contains a code segment of its own and shares rest of address space with parent (main thread).
- User level threads get scheduled (allocating CPU time) by user level scheduler. User level scheduler maintains a thread object to identify and store state of a user level thread.
- Kernel or its subsystem doesn't identify user level threads.

Unix version pseudo code

- Each task created by fork is a kernel supported thread.
- Kernel supported threads contains its own address space in user mode and PCB in kernel mode.
- Kernel and its subsystems identify kernel supported thread using PID and PCB.
- Kernel supported thread are scheduled and managed by scheduler.

Advantages of user level threading

- Better response time on average hardware

  User level threading is better because of lack of context switches.

- Developer friendly frame work (easier to write code, debug, maintain)
  1) All shared data can be accessed by user level thread implicitly by making it available in the data segment (as they have common PCB)
  2) Attaching the parent process to debugger allows accessing stack or data of any thread

Advantages of kernel level threading

- Kernel supported threads are more fault tolerant because the process which causes threat is halted while other process will run.
- Optimal usage of resources.

Fork()

- Fork creates a new process. It is done by duplicating the calling process.

- Since fork creates a child process as an exact duplicate of the calling process whatever instructions appear after fork call are executed twice, one in the parent and other in the child context. The order of execution is not guaranteed.
- On success, the PID of the child process is returned in the parent and 0 is returned in the child. On failure, -1 is returned in the parent and no child process is created.
- It is a common practice to follow fork call with a conditional construct. It helps assigning unique job to execute in parent and child.
- The stack and data segment is copied to child process.
- Child process created using fork can continue to run even after termination of parent process.
- When the immediate parent terminates init process (PID 1) takes over as parent.
- When a process terminates after executing instructions in the core segment, it is put into exit state. A process in exit state cannot content for CPU time (i.e. it can never become a context)
- Resources allocated by a process are not released /freed until PCB of the process is destroyed.
- It is always parent process's responsibility to ensure that the child process is destroyed on termination. To destroy terminated child processes the following methods can be applied.
    1) Synchronous clean up
    2) Asynchronous clean up
    3) Auto clean up.

1) Synchronous

- This method requires parent process to be suspended until termination of child process.
- When child is terminated instruct the kernel to destroy the child and resume execution.
- Wait call can also be used to collect the exit value of the child process. Exit value collected can be inspected to know the actual cause of termination using any of the following resources
    1) WIFEXITED (status) : returns true if the child terminated normally
    2) WEXITSTATUS(status): returns the exit status of the child
    3) WIFSIGNALED(status): returns true if the child process was terminated by a signal.
    4) WTERMSIG (status): returns the number of the signal that caused the child process to terminate.

2) Asynchronous

- This method requires parent to register a signal handler for SIGCHILD.
- Process manger delivers SIGCHILD to parent process whenever child process is terminated, stopped, continued.

3) Auto

- Terminated child process can be set to automatically destroyed mode without any further instruction from the parent.
- This requires the parent process to enable the default handler for SIGCHILD with a flag SA_NOCLDWAIT.
- Auto clean up is recommended to be applied only when child exit status is not important to parent.