OPERATING SYSTEM

CONTEXT SWITCHING

1. PROJECT DESCRIPTION

1.1 Overview

Context switching is the process to store and restore the state of a CPU in Process Control Block (PCB) so that the previous process can be resumed.

Resuming of the process goes like this –

- Program counter stores the address of the instruction from where the execution will start after getting resumed.
- File manager stores all the data one has written so that when the process will resume, it can retrieve the data and process further.
- Process Control Block store/save the executed data.

After performing all these tasks, control is given to another process and another PCB stores the executed instructions. So, if again our control comes to previous process, then it will load its PCB and can resume it. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

Context switching is generally computationally intensive. That is, it requires considerable processor time, which can be on the order of nanoseconds for each of the tens or hundreds of switches per second. Thus, context switching represents a substantial cost to the system in terms of CPU time and can, in fact, be the most costly operation on an operating system.

Context switching is computationally intensive and much of the design of operating systems is to optimize the use of context switches. Switching between threads of a single process can be more faster than between two processes because threads share the same virtual memory maps. Context switching is fast when implemented by threads because we only have to save and/or restore PC and registers.

The advantages of using threads in context switching are that if one thread is blocked, another thread can run. So if one process is completed than only the other process can run. Additionally, threads do not need to use interprocess communication.

The biggest disadvantage is that there is no protection between threads.

One major advantage claimed for software context switching is that, whereas the hardware mechanism saves almost all of the CPU state, software can be more selective and save only that portion that actually needs to be saved and reloaded.

Associated performance issues, e.g., software context switching can be selective and store only those registers that needs to be storing, whereas hardware context switching stores all the registers whether they are required or not.

Consequently, a major focus in the design of operating systems has been to avoid unnecessary context switching to the extent possible. However, this has not been easy to accomplish in practice. In fact, although the cost of context switching has been declining when measured in terms of the absolute amount of CPU time consumed, this appears to be due mainly to increases in CPU clock speeds rather than to improvements in the efficiency of context switching itself.

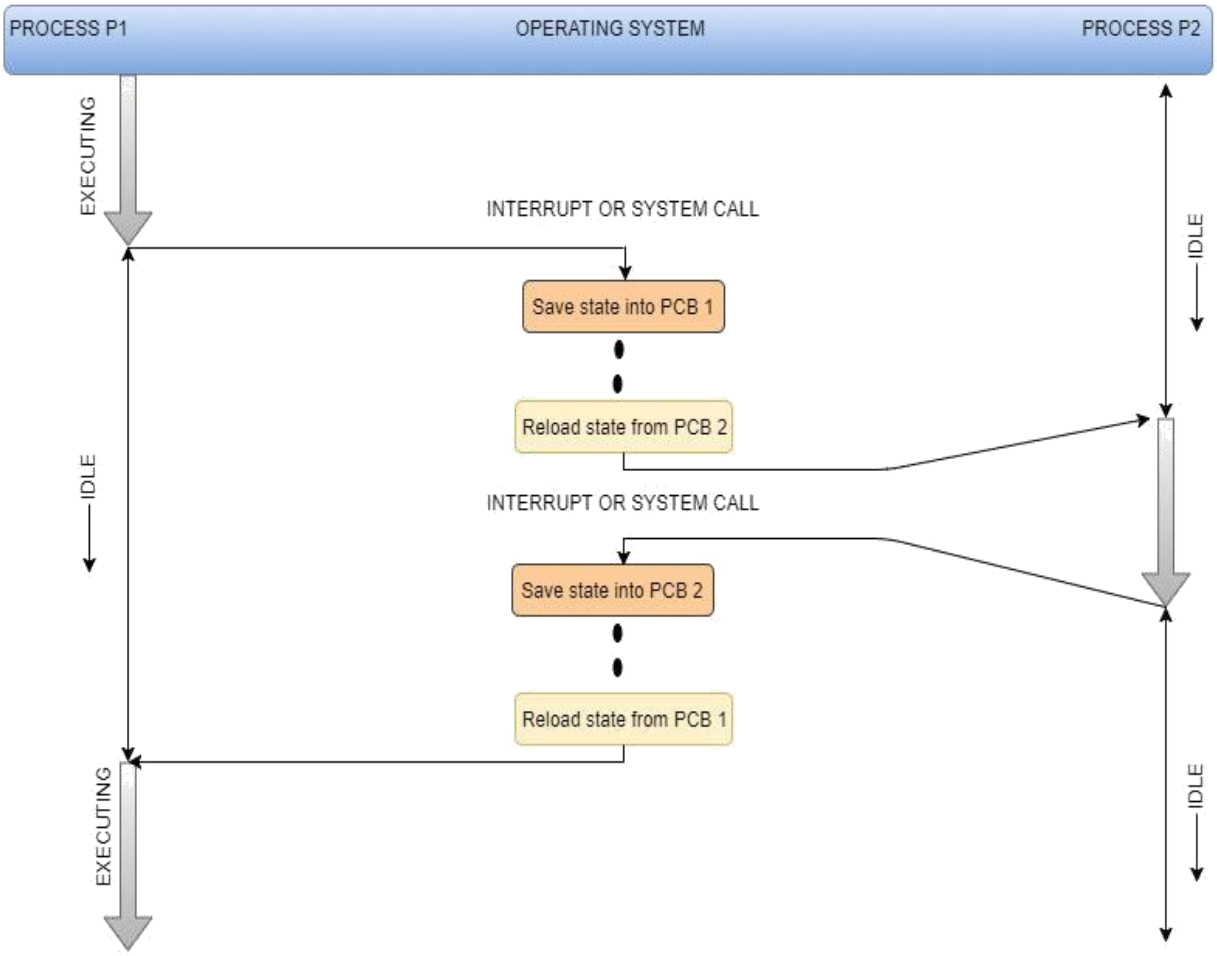
Context switching for threads of a single process is done so that the time required for the execution of the process decrease as compared to the time taken by a regular process execution.

1.2 Assumptions

We have taken one process which is divided into five different threads. We have assumed that the process consists of five different functions which are independent of each other. We have given limits to each function manually so that at one point of time each function get terminated. Instead of implementing any

interrupt we have considered keyboard inputs as interrupts. We haven't done kernel level implementation because it can harm the PC internally.

2. ARCHITECTURE/MODEL/DIAGRAMS



While execution of the one process when interrupt occurs, CPU switches from one process to another process and PCB stores the state of previous process. After execution of the second process, the control comes to previous process and then it re-starts from the point where it stopped earlier.

3. TECHNICAL SPECIFICATIONS/DETAILS

3.1 Threads

- The main concept we have implemented is thread. Some of the sub topics of threads which we have used are pthread_create and pthread_join.

- **Pthread_create** creates new thread which is being called by the new process.

- **pthread_join** waits for all the treads specified by the process to terminate together and for that the specified thread must be joinable.

Thread is a light weight and makes user less resources than a process. It is nothing but a sequence of instructions. If one thread is blocked or in a waiting state even then the another thread from the same process can run. At a time only one thread can be handled by CPU. For parallel threading, in order to switch between instructions, the state must be saved. In order to know what instruction is going to execute next, Program counter can come into the pictures. We have implemented the user level threads for the user level calls.

3.2 Signals

- We have implemented signals to generate interrupts between the threads and also for exiting from the code.

- **SIGINT:** This signal is used to get interrupt from the user. When user presses CTRL +C, it considers as interrupts.

- **SIGQUIT:** This signal is activated when the user wants to quit the process. Shortcut for this is CTRL+\.

- **SIGTSTP:** This signal helps to break out from the process itself. In short it suspends the whole process. This can be done using CTRL+Z.

3.3 Handler

- Handler handles the threads whenever the signal arrives. We have created five handlers.

- In all the handler functions we have taken signal as an argument. We have declare the threads and afterwards we have created it.

3.4 Structure

- A struct name stores has been implemented in our code. There we have declared two variables that is thread_id of type pthread_t and num of type integer.

3.5 Files

- Files are used to store values or data. We have made use of file in order to store the data about the threads like its' latest value stored or the values retrieved. In short in our project file has worked as PCB i.e. process control block.

- A file named logfile (which is a text file) has been created for the same purpose as mentioned above.

3.6 Functions implemented

- **main() –** In the main function we have first opened the file named logfile.txt. Then we have generated the process id using getpid() function. Instance of signals (sigint and sigquit) are made. After creating the main thread i.e. for addition, thread join function is called.

- **addition() –** In this function the variable named val is initialized with value 50 which is incremented by one till it reaches a value of 70.

- **fib() –** This function implements the Fibonacci series till 'n' reaches value equal to 10.

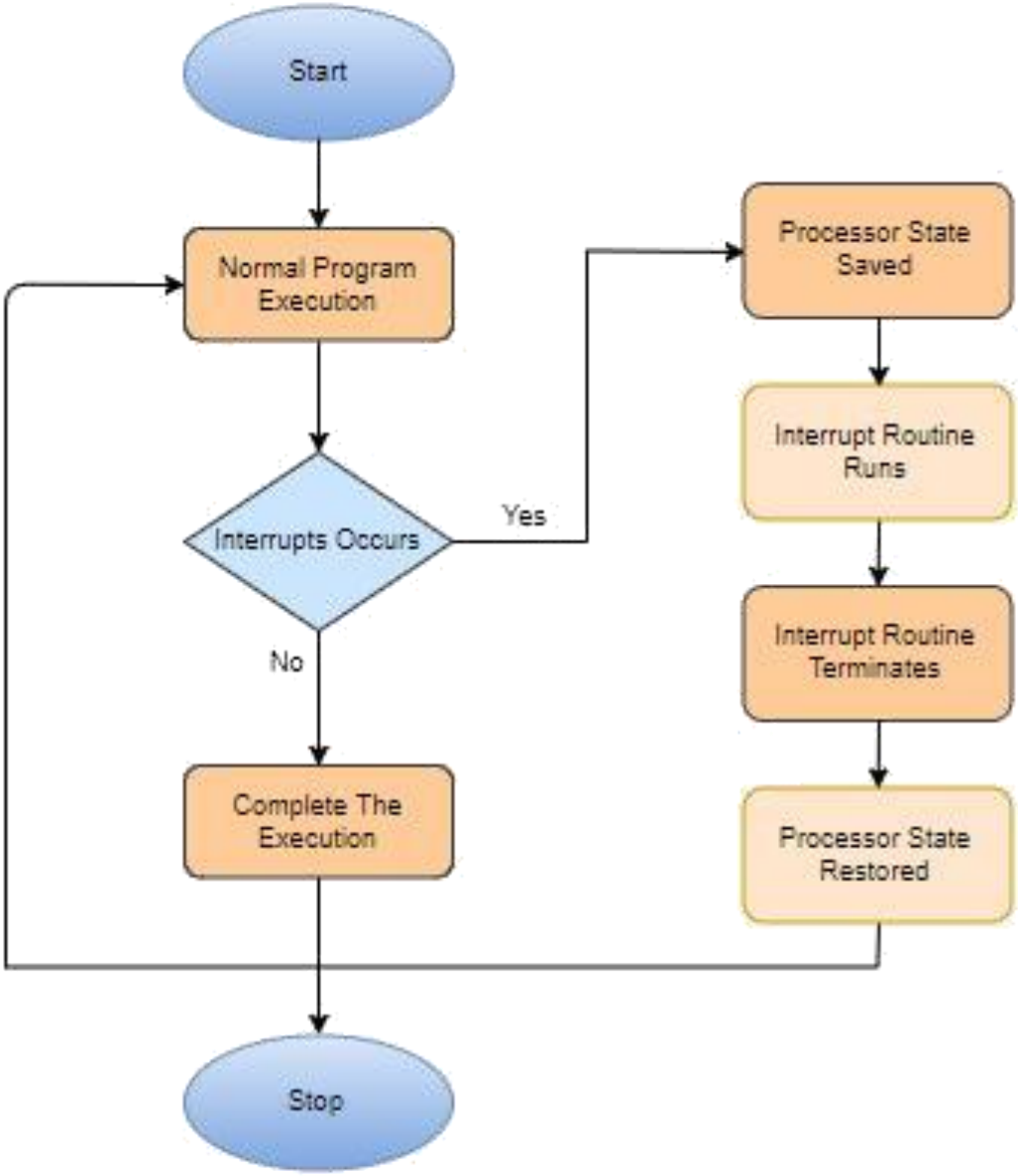
- **pat() –** This function tries to create a pyramid or triangular like shape using its logic.

- **add_5() -** In this function the variable named val_1 is initialized with value 1000 which is incremented by five till it reaches 1025.

- **alphabets() -** This function prints the alphabets sequentially and increments by one whenever an interrupt occur.

- **quit() –** This function is just for termination of the whole process. It is called by the handler whenever CTRL+/ is pressed.

4. FLOW CHARTS



5. ALGORITHM

Handlers Specification

handler_1: handles signal arrives from main function

handler_2: handles the quitting signal, arrives from main function

handler_3: handles signal arrives form addition operation

handler_4: handles signal arrives form alphabets operation

handler_5: handles signal arrives form Fibonacci operation

Create threads to handle these interrupts using inbuilt function pthread_create according to their specification.

Function: addition

Function which handles the signal generated from main function.

1. Define an integer(i).
2. Initialize a value for increment.
3. Increment the value of integer(i) after every 2 seconds.
4. Repeat step-3 until value of integer reaches 20.
5. Store current data.
6. Repeat the process until next interrupt arrives.

Function: fib

It is used by handler_1 in order to handle the signal for Fibonacci operation.

1. Store the values from previous signal.
2. Define two integers for Fibonacci sequence.
3. Add previous to integer in order to get next Fibonacci number after every 2 seconds.
4. Repeat step-3, 10 times.
5. Store current data.
6. Repeat the process until next interrupt arrives.

Function: pat

It is used by handler_3 in order to handle the signal for pattern.

1. Store the values from previous signal.
2. Print the pattern.
3. Delay of 2 seconds.
4. Store current data.
5. Repeat the process until next interrupt arrives.

Function: add

It is used by handler_4 in order to handle the signal for addition.

1. Store the values from previous signal.
2. Initialize an integer.

3. Increment the value of integer by 1 after every 2 second.
4. Repeat step-3, 5 times.
5. Store the current data.
6. Repeat process until next interrupt arrives.

Function: alphabets

It is used by handler_5 in order to handle the signal for addition.

1. Store the values from previous signal.
2. Print the alphabets.
3. Store the current data.
4. Repeat process until next interrupt arrive.

Function: quit

It is used by handler_2 in order to handle the signal for quitting.

1. Print message.
2. Return.

Function: main

1. Open the file.
2. Take reference for thread.
3. Print process ID.
4. If interrupt for quit is generated Then use handler_2.
5. Else, check any other interrupt is generated or not.
6. If any other interrupt is generated Then create thread using inbuilt function pthread_create.
7. Use pthread_join function to have synchronization.
8. If signal arrives form function Then use handler_1.
9. If signal for addition arrives Then use handler_3.
10. If signal for alphabets arrives Then use handler_4.
11. If signal for Fibonacci operations arrives Then use handler_5.