What is context switch?

A context switch is the process of storing and restoring the state of a process or thread so that execution can be resumed from the same point at a later time. This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system.

Why Context Switch is require?

Context switching allows for one CPU to handle numerous processes or threads without the need for additional processors. Any operating system that allows for multitasking relies heavily on the use of context switching to allow different processes to run at the same time.

When to switch?

There are three scenarios where a context switch need to occur.

1. Multitasking
2. Interrupt Handling
3. User and Kernal Mode Switching

Multitasking

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Most commonly, within some scheduling scheme, one process needs to be switched out of the CPU so another process can run. Within a preemptive multitasking operating system, the scheduler allows every task to run for some certain amount of time, called its time slice. If a process does not voluntarily yield the CPU a timer interrupt fires, and the operating system schedules another process for execution instead. This ensures that the CPU cannot be monopolized by any one processor-intensive application.

Interrupt Handling

Interrupts are raised by hardware or programs to get OS attention. There are two types of interrupts.

- Hardware interrupts: raised by external hardware devices.
- Software interrupts: raised by user programs.

On a single-CPU machine, by definition it can only be running one thread of control at a time. It only has one register set, one ALU, etc. So if the interrupt handler is running there simply are no resources with which to execute a context switch.

When an interrupt occurs, only the context that the interrupt handler actually needs to use needs to be saved and then subsequently restored. If interrupt handler is written in a high-level language, this will pretty much be equivalent to a full thread context switch; because there are no constraints on what resources such an interrupt handler might touch. In assembly language written interrupt handler, we can keep track of exactly which registers it touches and save only those. This allows the execution of the interrupt handler to be extremely fast, reducing its impact on the rest of the system, and/or allowing it to handle interrupts at a higher rate.

What happens while Context Switch?

Single Core CPU ,

1. All context switches are initiated by an 'interrupt'. This could be an actual hardware interrupt that runs a driver, (eg. from a network card, keyboard, memory-management or timer hardware), or a software call, (system call), that performs a hardware-interrupt-like call sequence to enter the OS. In the case of a driver interrupt, the OS provides an entry point that the driver can call instead of performing the 'normal' direct interrupt-return & so allows a driver to exit via the OS scheduler if it needs the OS to set a thread ready.
2. Non-trivial systems will have to initiate a hardware-protection-level change to enter a kernel-state so that the kernel code/data etc. can be accessed.
3. Core state for the interrupted thread has to be saved. On a simple embedded system, this might just be pushing all registers onto the thread stack and saving the stack pointer.
4. It may be necessary to mark the thread stack position where the change to interrupt-state occurred to allow for nested interrupts.
5. The driver/system call runs and may change the set of ready threads by adding/removing TCB's from internal queues for the different thread priorities, eg. network card driver may have set an event or signaled a semaphore that another thread was waiting on, so that thread will be

added to the ready set, or a running thread may have called sleep() and so elected to remove itself from the ready set.

6. The OS scheduler algorithm is run to decide which thread to run next, typically the highest-priority ready thread that is at the front of the queue for that priority.
7. The saved stack pointer from the TCB for that thread is retrieved and loaded into the hardware stack pointer.
8. The core state for the selected thread is restored. On my simple system, the registers would be popped from the stack of the selected thread. More complex systems will have to handle a return to user-level protection.
9. An interrupt-return is performed, so transferring execution to the selected thread.

Multicore CPU:

The scheduler may decide that a thread that is currently running on another core may need to be stopped and replaced by a thread that has just become ready. It can do this by using its inter processor driver to hardware-interrupt the core running the thread that has to be stopped. Rest all steps are similar as single core CPU.

Implementation

Objective

The purpose behind doing this project is to understand the basic concept of operating system and to learn how operating system does a context switch.To understand the need of context switch and what operating system do while context switching. So we simulated a simple dispatcher which does context switch based on time slice.

Input

Multiple files are taken as an input. Each file is acting as a different individual process. Each process is made of a set of instructions.

Output

We implemented round robin scheduling algorithm. So each process is given equal time to execute. Dispatcher will switch the process when time slice occurs.While switching, the variables of the currently running process is stored in the file and the data of the next process is loaded to the main memory.

How our code works?

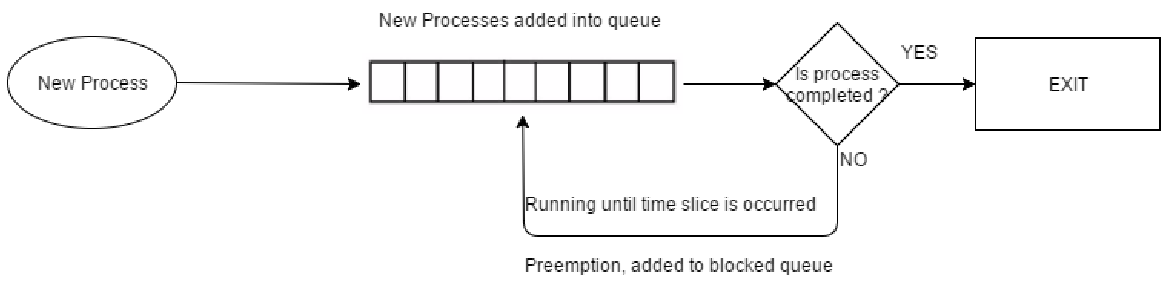
1. Files take as an input. File contains a set of instruction
   a. Process execution
      i. Fetch the instruction
      ii. Decode the instruction
      iii. Perform the operation on decoded instruction
      iv. Update the value of variables in the main memory
      v. Store the updated variable in the secondary memory
   b. On time slice
      i. Storing the variables of currently running process in a secondary memory.

ii. Suspend the currently running process and put it at the end of the queue.
iii. Dequeue the first suspended process and load it into the main memory. Restore the variables of that process.
iv. Perform the process execution.

c. On graceful exit
   i. Store data of interrupted process
   ii. Put that process in the queue of suspended process

2. Maintaining the log file
   a. After execution of each instruction, write the basic details such as timestamp ,type of the instruction in the log file.
   b. Maintaining the log file for dispatcher.

Flowchart


New Processes added into queue
New Process
Is process completed ?
YES
EXIT
NO
Running until time slice is occurred
Preemption, added to blocked queue