|  |
| --- |
| dszxcxcPwda |
| Context Switching In kernel Level Threads |
|  |
| OS Project |

**IET, Ahemdabad University**

April 30, 2015

Authored by: Param Trivedi (121033)

Pooja Dhamija (121034)

Context Switching In kernel Level Threads

OS Project

Index:

1. What is context switching?
2. Basic Understanding
3. Action of kernel to context switch among threads
4. States of thread
5. How swapping process is done in memory
6. Kernel code in 3.19.2
7. Understanding Switch\_to.h
8. Understanding Process.c
9. Understanding kthread.c
10. Understanding Thread\_info.c
11. Implementation
12. Final implementation

**What is context switching?**

A context switch is basically the kernel suspending *execution of one process* on the CPU and resuming execution of some other process that had previously been suspended.

 So there are two parts:

1) Program Counters

2) Registers.

Suspended execution of a process Is stored in an **register** which is later resumed if the process is called.

A **program counter** is a specialized register that indicates the position of instruction sequence in CPU and which holds either the address of the instruction being executed or the address of the next instruction to be executed.

**Basic Understanding of context switching:**

Context switching can be described in slightly more detail as the kernel performing the following activities with regard to processes (including threads) on the CPU:

(1) Suspending the progress of one process and storing the CPU's *state* (i.e., the context) for that process in a particular location in memory.

(2) Retrieving the context of the next process from memory and restoring it in the CPU's registers.

(3) Returning to the location stored in the program counter in order to resume the process.

**Action of Kernel to Context Switch Among Threads:**

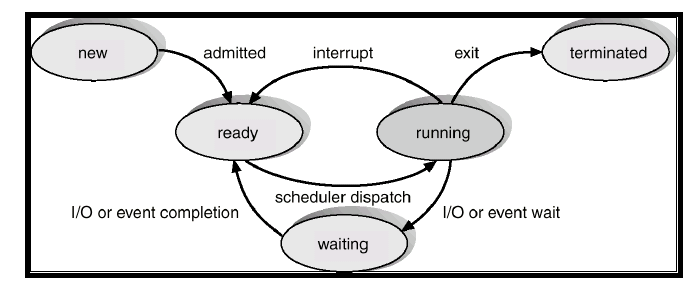
The threads share a lot of resources with other peer threads belonging to the same process. So a context switch among threads for the same process is easy.

It involves switch of register set, the program counter and the stack. It is relatively easy for the kernel to accomplished this task.

Whenever a context switch is being done the thread that is being switched it’s content must be saved in the PCB (Process Control Block):

* The process state.
* The program counter, PC.
* The values of the different registers.
* The CPU scheduling information for the process.
* Memory management information regarding the process.
* Possible accounting information for this process.
* I/O status information of the process.

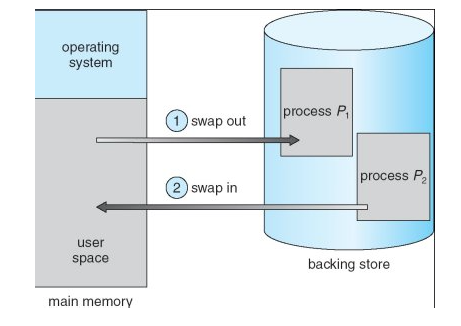
**States of Thread:**



Process Steps:

1. A new process is created
2. Now it enters **READY** state if a thread is currently running
3. Now when a thread issues an interrupt then a current thread which is **RUNNING** goes into waiting state. It remains in the **WAITING** state untill another interrupt is issued or the running thread completes it task.

**How swapping process is done in Memory:**



Here there are two process. Process P1 and Process P2 .Now here as shown in the figure Process P1 is running in the user space. Now Process P2 tries to execute the task and so process P1 will have to swap out from user space and stored it in a stack. Now Process P2 is brought in the user space and starts executing.

**Kernel Code in 3.19.2:**

The documentation in Kernel 3.19.2 mentioned that in ia-64 architcture “switch\_to.h” is responsible for context switching. Switch\_to.h is found in many other architectures and we have to choose from one of the architecture so we have chosen ia64 architecture. In the documentation, we found the schedular folder in which the text file sched\_arch.txt gives the infornation of the context switching performed between threads and gives the name of the header file switch\_to.h which should be included in the source code.

Then we saw the switch\_to.h header file. It says that if two threads are in different address space then schedule( ) function takes care and to switch to the new address space using switch\_mm.h file.

After that we encoutered “process.c” which uses “switch\_to.h” . In “process.c” data of the thread in saved in a stack register when thread is switched.

**Understanding “switch\_to.h”:**

The path for the switch to header file is as follows:

switch\_to.h /linux-3.19.2/arch/ia64/include/switch\_to.h.

In switch\_to.h basically two functions are defined:

extern void ia64\_save\_extra (structtask\_struct \*task);

extern void ia64\_load\_extra (structtask\_struct \*task);

Task\_struct is the structure in which it stores the information of the process like pid, address, priority etc. .The state for the thread can be fetched from task->thread.fph. Context switching must be done before calling is64\_switch\_to().

**Understanding “process.c”:**

Ia64\_do\_show\_stack () is the function in which the buffer is used for the overflow condition. In show\_regs there will be the registers which will get printed in the kernel only but we were not be able to see the printed output. In this code, the copy\_thread function is used to copy the thread from user level and from kernel thread. In this the stack will have following quantities: memory stack, register pointer. When the new process comes then it will be considered as the child node and it is defined as the structure in the code.structswitch\_stack \*child\_stack.

When the threadis copied or switched then the current\_pt\_reg is stored in pt\_reg.

The thread is called by fork system call and it is suspended by using suspend().

**Understanding “kthread.c”:**

The reason for understanding this file was that in our implementation we were hoping to print the id of threads and their status. Now kthread.c is responsible for making kernel level threads and so it contains the PID of a thread.

**Understanding “thread\_Info.c”:**

Now in Linux Kernel – 3.19.2/Kernel/thread\_info.c contains general functions which can be used on the threads.

Function 1:

#definecurrent\_thread\_info() ((structthread\_info \*) ((char \*) current + IA64\_TASK\_SIZE)).

This function gives basic information about a current running thread.

Function 2:

#definetask\_thread\_info(tsk) ((structthread\_info \*) ((char \*) (tsk) + IA64\_TASK\_SIZE))

This function gives information of threads which are in ready state

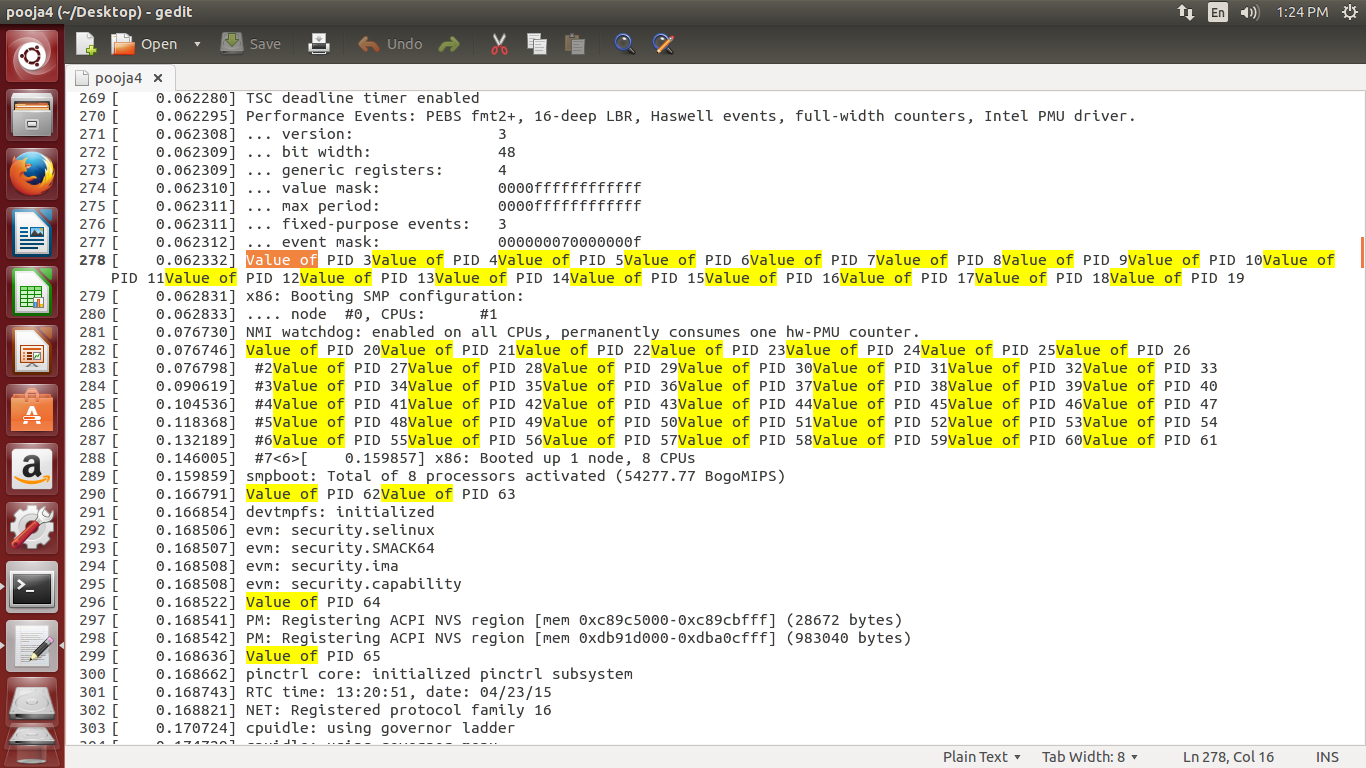
**Implementation:**

In the initial stage, we thought to write the code in c in which there will be context switch between the threads and the stack information is to be printed but we were not able to do it. To print the stack was the problem for us.

In process.c we tried to print child\_rbs and rbs but we were not getting any answer.

We came to know that these variables in which it was declared were not running .so we tried to get the information of the currently running process. So we used ps-aux or ps-ef. So at kernel level the users which were root were in the kernel level modes. But how to relate that process was difficult job.

In kernel level, we were told to print counter which will increment or decrement according to the context switching between threads. For this , we tried to print the process id from the kthread.c program and see the log file. The process id was getting printed but how to connect the process id with the threads and the stack value was difficult job.



The variables which were declared in the source code were not clear. It means that what the variable store, we could not understand. So this was the major problem of our project.

Then we tried to use the thread\_info and print the value in which it would be stored. The function which we found was current\_thread\_info() ((structthread\_info \*) ((char \*) current + IA64\_TASK\_SIZE)). This function was used in many .c file but we used process.c file but we were facing some problem.

**Final Implementation:**

Algorithm:

Step1: Mythread.Start() // This function starts a thread.

Step2: Mythread.suspend() // This function suspends a current thread.

Step3: Mythread.resume() // This functions resumes a suspended thread.

Step4: In main Function:

1. Create thread T1 and T2.
2. T1.start() // Start T1.
3. T1.suspend() // Suspend T1 and Start T2 (Any thread in waiting state).
4. T2.suspend() // T1 will resume and T2 suspends.

