School of Engineering and Applied Science (SEAS)
Operating Systems
Project Report

Context Switching in Operating Systems

Mentor: Dr Sanjay Chaudhary
Guided By: Mayank Jobanputra

Group Members:

Maitrey Mehta	(1401040)
Jay Shah	(1401053)
Parth Shah	(1401054)
Bhavya Patwa	(1401063)

Table of Content

I. Acknowledgements

II. Project Definition

III. Technical Details

IV. Algorithms in Implementation

V. Flow of the Project

VI. Implementation flow

VII. List of Programs

VIII. Test Data Sets

IX. Test Results

X. References

XI. Appendix A

I. Acknowledgements

We sincerely thank Prof Sanjay Chaudhary and Mayank Jobanputra for their continuous support and persevered guidance throughout the making of the project.

II. Project Definition

The primary task of our project is to simulate the task of context switching which is usually done by operating system. We have handled most of the scenarios where context switch occurs in our project, moreover when we need to switch a thread to another thread we need a dispatcher which will select next thread to be executed. Thus in our project we have used round robin as our scheduling algorithm which depicts the task of selecting next process, runs up to a time slice and then selects other thread for execution. Finally through a log file we can observe the simulations of our project.

III. Technical Details

A. Context Switch [1]

A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU (central processing unit) context from one process or thread to another. A process (also sometimes referred to as a task) is an executing (i.e., running) instance of a program.

B. Occurrences of Context Switch [1]

There are 3 occurrences of context switch

1. **Occurrence of interrupt:** Reaction to any asynchronous external event, interrupts like clock interrupt, I/O interrupt and memory faults can cause a thread or task to switch
2. **Time slice:** To avoid starvation operating system time slices a running process and selects another process for execution, such an event leads to context switch.
3. **System call:** Operating systems pre-empts a currently running process or thread and replaces it with another process, when such pre-emption occurs event of context switching occurs.

C. Process of Context switch

Whenever a context switch occurs following steps execute in sequence to ensure a successful switching of context.

1. Save the context of processor, as processor (in case of single core) has the value of the current process running.
2. Update the state of current running process from running to another appropriate state.
3. Move the process to appropriate queue
4. Dispatcher dispatches other process by any of the scheduling algorithm
5. Update all the data structures for currently selected process
6. Restore the context of the processor when the selected process was last executed.

D. Dispatcher (Scheduling next thread)

The task of scheduling next process from a pool of ready process is done dispatcher, for this dispatcher implements scheduling algorithm. Dispatcher itself is a small process.

E. Thread Control Block

Thread Control Block (TCB) is a data structure in the operating system kernel which contains thread-specific information needed to manage it. The TCB is "the manifestation of a thread in an operating system"

IV. Algorithms in Implementation

We have implemented round robin as our scheduling, moreover we have additionally implemented our own thread control block and a dispatcher function.

A. Round Robin

A straightforward way to reduce the penalty that short jobs suffer is to use preemption based on a clock. The simplest such policy is round robin. A clock interrupt is generated at periodic intervals. When the interrupt occurs, the currently running process is placed in the ready queue, and the next ready job is selected on a FCFS basis. This technique is also known as time slicing, because each process is given a slice of time before being preempted.

The main question in Round Robin is to decide the quanta or the time slice for which the process should be executed. This time quantum should not be so large such that whole process has completed its execution and still quan-tum is carried on. Similarly, the quantum should not be too small such that there are many fragments of the process and leads to higher context switch-ing penalties.

Thus for our simulation we checked RR for various different quantum and finally settled for 3ms. Before reaching this value we simulated our pro-gram which had 4 threads with different service time and reached to the conclusion that 3ms is the optimum time slice for our set of threads.

B. Thread Control Block (Implementation in project)

TCB consists of various flags determining the state of the thread, exitstatus which determines whether the thread has completed its execution or not. TCB also contains path for instruction execution, and the identifier of the process that belongs to that thread.

```
Struct_TCB{
    -bool blockedstate, blocked, readyState
    -bool exitStatus
    -char filePath
    -char stackfilename
    -int process id
}
```

blockedstate: This boolean flag ensures that the thread is in block state.

blocked: After declaring a thread in blocked state we cannot directly stop the execution of the thread as saving of processor context is one of the core job hence next thread will not start executing until this variable is set true.

readyState: This variable will be initially false, when a context switch occurs the current processor data has to be saved in the respective stack file and registers and context of new thread has to be loaded in the processor (our resource file), ready state flag becomes true when this is completed hence it signals dispatcher to schedule this particular thread.

exitStatus: Whenever a thread completes its execution it should not be again enqueued for execution setting this flag true after the completion of the thread ensures that a thread which has completed its execution will not be enqueued back.

filepath: The filepath for each thread is the location of *resource.txt* file. This file is depicting the task of a processor, thus the filepath for each thread will be the same and each thread will execute a given set of instruction on this filepath.

stackfilename: The stackfilename can be considered as memory location where the context or the value of registers (in our simulated case variables) of the processor (our resource file) will be saved, thus stackfilename is nothing but a file path in the current working directory where all the context of the thread experiencing context switch will be stored (as in our case a total of 4 threads are executed, hence there will be 4 stackfiles).

processid: This process id is just included as in our log instead of long thread id simple numerical entry ranging from [0,no of threads] can be made.

C. Dispatcher Function (Implementation in project)

The dispatcher implemented in our simulation performs the similar task as of the dispatcher in operating system. Dispatcher changes state of thread from '*running*' to '*blocked*' or vice versa and catches the exit flag of the TCB which helps in maintaining the ready queue of waiting processes. At every timeslice event, we have implemented the functionality of generating log into the dispatcher function and the time penalty for thread execution is eliminated by implementing the context switching within the dispatcher thread itself.

The pseudo code for this dispatcher block is given below

```
func_dispatcher(){
    - call TCB of all thread and maintain a RR scheduling queue
    - load registers(in our case variables) from any one stackfile
    - make state of any one thread to running
    /* beginning of critical section hence mutex lock*/
    - till time slice implement thread
     - save the updated context back to the stackfile
    /*critical section ends unlock mutex*/
}
```

Above pseudo depicts the working of our dispatcher block, TCB structure is called here and all dispatcher than manages all the flag of TCB for all the threads, also it is important to note that that at the beginning of critical section we have locked the thread the main reason behind this is once entered into critical section no interrupt can affect the execution for our case executing instructions is the critical part, and hence for the time slice when thread is executing the given set of instructions and updating the stackfile it is locked, after the time slice thread is again unlocked.

D. Interrupt Handling (Signal handling Implementation in project)

As occurrences of interrupt is one of the reason of context switching, interrupt handling implemented in the simulation ensures that when an external interrupt occurs the calling thread's state is changed to 'interrupted' and the values of register on processor (resource file) will saved into the thread's respective stack file. Also our dispatcher enqueues the calling process to ready queue and not any other queues like block queue, because interrupt due to I/O is not considered, only external interrupts(clt+c) are handled.

Instead of exiting or making graceful exit of the calling thread on interrupt we are again placing the thread in ready queue, the pseudo for this interrupt handler is given below

```
func_catchint(){
    -replace the context of  processor ('resource.txt') with another thread to be cheduleed.
    -declare interrupt state for the calling thread.
    -make dispatcher aware about this change
    -enqueue thread back to  scheduling queue.
}
```

V. Flow of project

Here we wish to show the entire flow in which our project was made. Our whole project is divided into 5 phases which are as follows;

1. Implementing RR without using threads.
2. Implementing dispatcher and basic TCB for 2 Threads.
3. External interrupt Handling.
4. Extending TCB blocks for multiple threads (>2).
5. Simulating the notion of a uniprocessor with interleaving and generating final logs.

1. Implementing RR without threads

A simple implementation of round robin scheduling has been made. A queue is maintained and each process is executed for a pre-defined time slice and then the process is pushed back in the queue.

2. Implementing dispatcher and basic TCB for 2 Threads.

Next, a user defined thread control block structure, which contains file path and a boolean variable blockedstate, is instantiated. Two threads are created, each given a file path through its respective TCB. A dispatcher thread is formed which alternatively runs each thread for a certain timeslice.

3. **External interrupt Handling.**

   Next, we included a hard-interrupt handling function which switched the thread when the interrupt key is pressed.

4. **Extending TCB for multiple threads (>2).**

   Next, we extended the project for multiple threads. A queue was maintained to implement round robin algorithm and a boolean variable exitstatus was added to the TCB structure in order to prevent queuing already completed threads.

5. **Simulating notion of a uniprocessor with interleaving and generating final logs.**

   Until now, the representation of the thread execution was done through for loop which ran through certain number of iterations while in synchronization with other threads to maintain the interleaving behaviour of our simulation.

   Now, we include real instruction execution within the thread functions and emulating the context switching by saving the registers into a stack file and restoring the same for the upcoming thread.
   Note that all the threads would execute the instructions and store the output in the common shared file. During context switch the saving and restoration of registers would occur between the respective thread's stack file and the common shared file.
   An additional boolean variable readystate had been added to TCB structure to avoid time penalties and anomalies caused by the dispatcher starting timeslice calculations even before the first thread starts its instruction execution.
   A log file has been maintained at each time slice, interrupt and thread completion events

VI. Implementation flow

All the functions mentioned in the flow diagram is mentioned and explained below:

- ***main():*** This main function of our simulation is the starting of our executable code all the local variables, thread initialization is done in this main function.

- ***copyfile():*** One of the most necessary function for our simulation as we need to copy the content of stackfile to resource file very oftenly, this copy functions helps us copying the content to and from resource file to respective stack file.

- ***ThreadFunction():*** This function is actual implementation of our context switch, for context switching as mentioned in phase 5 we executed some instructions and maintained the stackfile of each treads executing these instructions. ThreadFunction deals with taking a set of instruction loading the values of register and executing and updating the values of register.

    - **Pri():** Gives the instruction for execution.

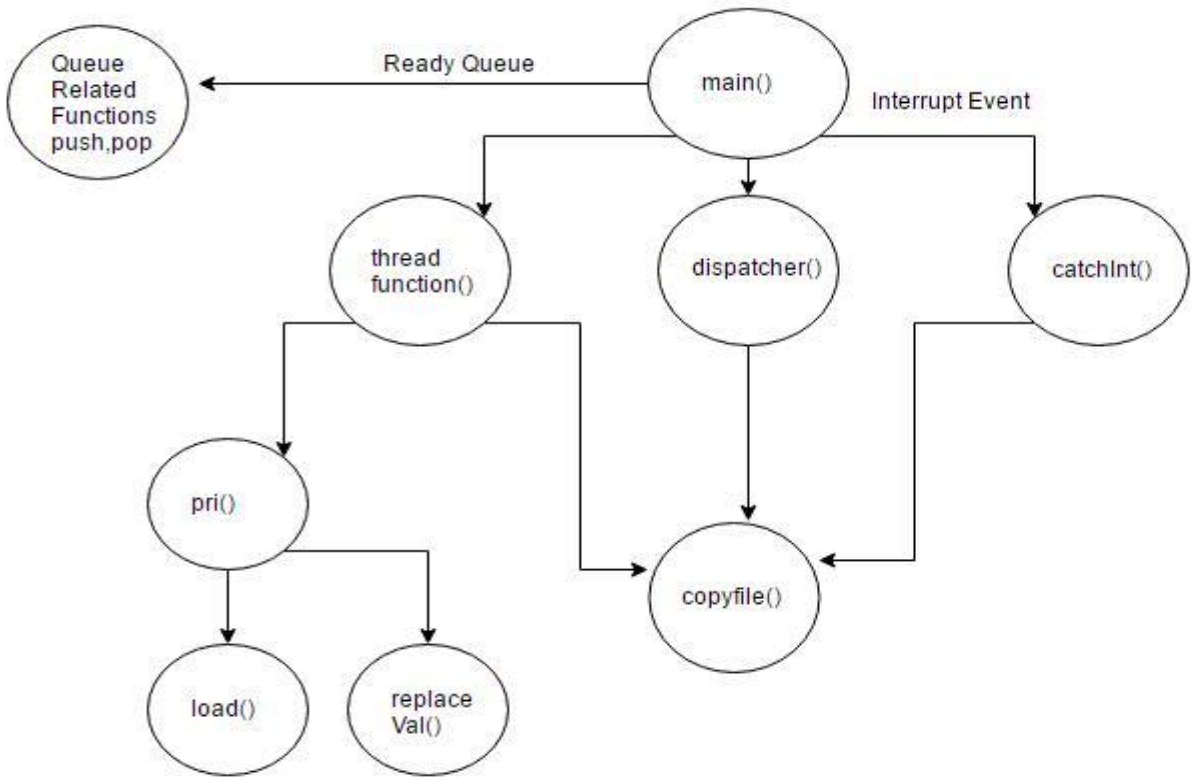
    - **Load():**The instruction from Pri() contains the information about which mathematical function has to be performed and on which variable this task has to be performed, Load() function will load all the essential variable needed for executing an instruction.

    - **replaceVal():** Whenever an instruction is executed the register has to be updated with the recent value after the execution, the task of replacing the older value of register with the newer one is done by replaceVal().

- **Dispatcher ():** Performs the task discussed in the implementation of algorithm (III.C).

- **Catchint ():** Performs the task in the discussed in the implementation of algorithms (III.D).

For proper understanding given in appendix A is a case for implementation flow.



VII. List of Programs [2] [3]

C*ontextSwitch.c:* This is our main simulation file through which our project is executed.

decode.h: Decodes instruction and calls load function to load values from the common resource file.

load.h: This returns the values corresponding to the variables called from the common resource file

replacevar.h: It contains the replaceval function which writes the calculated value back to the common resource file.

fileone.cpp: A simple simulation of test data in ***filein_1.txt.*** It is used to verify that the values calculated in this program is same as that in the file ***stack_filein_1.txt.***

filetwo.cpp: A simple simulation of test data in ***filein_2.txt.*** It is used to verify that the values calculated in this program is same as that in the file ***stack_filein_2.txt.***

VIII. Test Data Sets

- ***filein_1.txt:*** File containing instructions for thread 1.
- ***filein_2.txt:*** File containing instructions for thread 2.
- ***filein_3.txt:*** File containing instructions for thread 3.
- ***filein_4.txt:*** File containing instructions for thread 4.
- ***initResource.txt:*** Has the initial value of each variable (simulated registers).
- ***instructions.txt:*** Contains integer code corresponding to all operation.
- ***resource.txt:*** The common shared file on which resource variable values of the executing process will be held.

IX. Test Results

Process_log.txt: This file contains a final log generated for

X. References

[1] www.linfo.org/**context_switch**.html
[2] Pthread primer- A guide to Multithreaded programming
[3] https://users.cs.cf.ac.uk/Dave.Marshall/C/

Appendix A

Example for Implementation flow discussed in V.

- Number of threads: 3
- Registers (variables) value initially in stackfile : a=5, b=6, c= 7
- Common shared file: ‘resource.txt’
- Separate Stack file: stack1.txt, stack2.txt, stack3.txt

Instructions to execute on all 3 thread
Below is the example after which 3 instructions are given.
ADD a b c – This notation means addition of a and b and saving result to c

ADD a b c
SUB c b a
MUL a c b

The final answer from each thread should be thus

c = 11
a = 5
b = 55

Implementing our flow

- Main function initializes all the thread.

- Dispatcher dispatches one of the thread.
    - Lets say thread 1 is dispatched.

- **CopyFile():**Copy file is called which copies all the variable from stackfile to resource.txt
    - Now resource file is updated by stackfile1 registers.

- **pri()**: Instruction is decoded.
    - ADD a b c

- **load():** variables needed for instruction is loaded
    - variables a, b, c are now used for performing given mathematical operation.

- **Replaceval():** variable which is modified is updated here.
    - resource.txt now replaces c with 11 instead with 7.
    - Resource.txt is agained copied in stackfile1.txt

- After this execution according to timeslice either thread 2 is selected or second instruction is decoded.