**Ayush Patwari**

**Operating Systems [CS503]**

**Lab2a Answers and Discussion**

---

## 3. Review of Virtualization

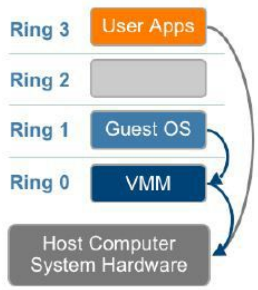
1. In the scenario mentioned in the question, lets assume we have full virtualization and two kernels: Linux and Windows are running apps. Also, let the apps not be using any priviledged instructions e.g IO interrupts. Let a sample app be as follows:

```
.text

/*
* This function will load the argument in the eax register
* Then it uses the bswap call to swap the bits 8-1 to 32-25 and 16-9 to 24-17
*/
.global host2netl_asm
host2netl_asm:

  movl 4(%esp), %eax
  bswapl %eax
  ret
```

This app basically takes an argument passed as a 4-byte words and changes its endian-ness by calling a bswap instruction and storing the result in eax register. There is no syscall and hence no privileged instruction required to be executed. In full virtualization, a guest OS will trap into the VMM only when a privileged instruction needs to be executed since the guest OS is not running in ring 0. However, if an app inside the OS doesn't perform a task which invokes privileges instruction it will not trap and run as if it were running on "bare metal" (hardware) since there is no translation of user space instructions it runs directly on the processor. Although there will be sharing of resources between the multiple OSes but from the OS's point of view, when it is executing it is running without any overheads. Now, even if the app inside a guest OS executes a syscall which does not need to be executed in the privileged level there will be no trap into the VMM and thus no overhead. For e.g in the getpid syscall, although the user uses an INT instruction to perform a syscall, the syscall handler will run in the current OS PL (a ring higher than user ring 3 but lower than 0) and just looks through the process table and can return the pid. It doesn't need to trap into the VMM which currently runs in ring0.

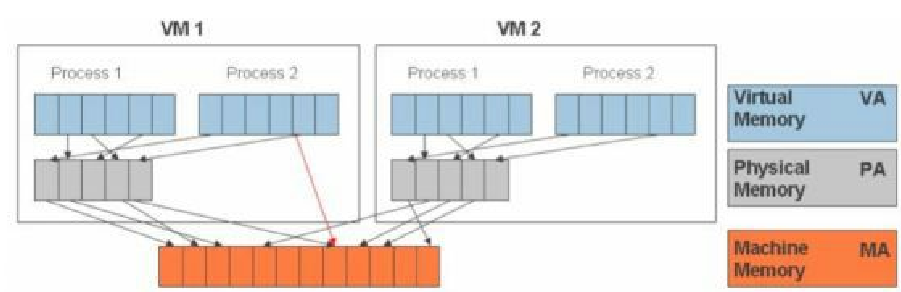
2.
    a. Now, consider the user space apps in the guest OS make system calls which execute privileged instructions for e.g cli or sti instruction in the x86 architecture. The modes of operation for full virtualization is shown below. The virtualization is achieved via the trap-and emulate principle. If the guest OS wants to run a privileged instruction, system will trap to kernel mode (VMM) and the VMM will then simulate the hardware changes

and start the execution again from where the OS trapped in the handler (OS handler address). Suppose a cli instruction is issued, then the VMM will clear the IF (set to 0) and then return to OS handler. In this way full virtualization is achieved at the cost of a trap which is expensive.  So every privileged instruction leads to a trap into the VMM and




significant overhead.

b. Isolation/protection:
    i. To ensure memory isolation/protection between the guest kernels, the VMM maintains shadow page tables, which is a another layer of memory virtualization. Guest OSes maintain page tables to map virtual addresses for each process to physical pages. However physical pages no longer map directly to memory but rather go through another level of indirection and the VMM maps the guest OS physical memory to actual machine memory using shadow page tables. The mappings from virtual memory to actual memory maybe be cached in TLB hardware for increase in performance. When a guest OS changes the mapping of virtual to physical memory, the tables in VMM are also updated. This scheme ensures that there is isolation of guest OSes and they are not able to get access to memory of other processes. Any attempt made by the OS or an app inside the OS to acces memory location other than allocated to the VM will trap into VMM which will deliver a fault to the guest OS resulting in system crash or panic. Also, since the protection in ensured between apps by the guest OS itself, it is also ensure by the VMM. Whenever memory is allocated to the guest OS by VMM, it zeroes out all pages ensuring that there is no memory leak between OSes or apps within the OSes.

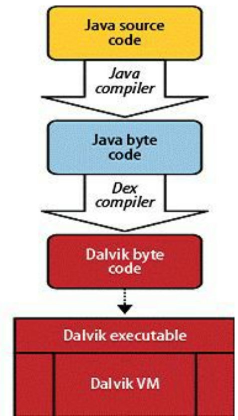



    ii. Modes of operation: As seen in the diagram above, since guest OSes run in user modes and trap into VMM for privileged instructions, they are not able to perform instructions to violate the isolation property and impact other OSes.

3. Sensitive instructions are those that modify the system registers. Examples of such instructions in x86 are: PUSHF,POPF,SGDT,SIDT,SLDT,SMSW. When the behavior of these instruction depends on the mode of operation it leads to problems in virtualization. For e.g popf instruction

pops the top of stack into the EFLAGS register. However it will ignorer the IF flag if not is kernel mode. In the virtualized scenario seen above, the OS will not run in kernel mode but at a lower ring (for e.g. ring 1). Hence system will not trap into the VMM rather fail silently and continue the next instruction which will lead to improper emulation. VMWare uses Binary Translation/Rewriting to rewrite certain ring 0 instructions, such as popf in terms of user mode instructions and cause them to cause a trap to VMM which could them perform proper emulation. To improve performance the instruction translation is cached and used for future use.

4. Dalvik VM:Dalvik is a software virtualization environment or a process virtual machine which adds a virtual layer between the OS (Linux kernel) and the application bytecode to allow for portability across various platforms. It forms a part of the Android Software stack. A *process virtual machine* is designed to run a single program, which means that it supports a single process. This means that when a new applications is created it is allocated a new instance of Dalvik VM which exits along with the application when it dies. A typical compilation flow for Dalvik is shown below:




   a. The code is written in JAVA and compiled using javac to convert to bytecode which is then stored as .class files.


For e.g: a code like this

```
public MainActivity() {
  super();
  currentPosition = 0;
}
```

Is converted to java bytecode of this form:

```
public com.hfad.bitsandpizzas.MainActivity();
  Code:
   0:   aload_0
   1:   invokespecial   #5; //Method android/app/Activity."<init>":()V
   4:   aload_0
   5:   iconst_0
   6:   putfield        #3; //Field currentPosition:I
   9:   return
```

   b. The .class files and any dependency .jar files are then converted to single Dalvik executable format (.dex) file which looks something like this:

```
0x0000: iput-object v1, v0, Lcom/hfad/bitsandpizzas/MainActivity; com.hfad.bitsan
dpizzas.MainActivity$2.this$0 // field@4869
0x0002: invoke-direct {v0}, void java.lang.Object.<init>() // method@13682
0x0005: return-void
```

This file is then packaged along with the resources of the application like images etc. and converted to a .apk package using the **aapt** tool and loaded onto the android device usind an **adb** connection.

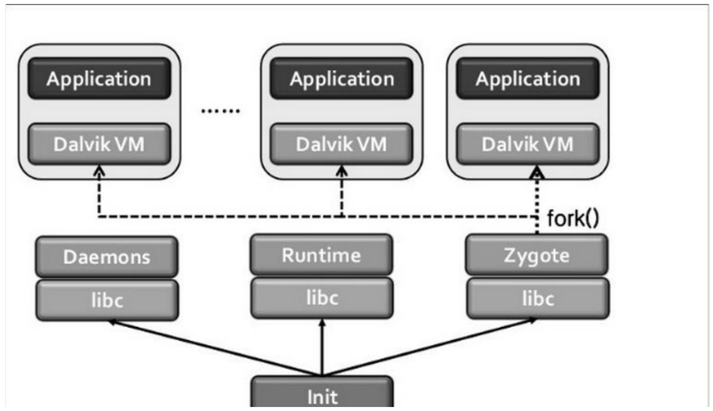
c. When an application is run, a dummy incomplete android process Zygote is run (forked to form a new process with core libraries and memory) and bind to this .apk. Now, the code for the application which is stored in the .dex is converted to the native compiled code/ELF shared object using to **dex2oat** tool. It will then look like any other native machine code(for e.g on a x86 arcitecture) it looks like as shown. This native code is then mapped to the process memory.

```
0x001db888:         85842400E0FFFF      test    eax, [esp + -8192]
suspend point dex PC: 0x0000
GC map objects:  v0 (r5), v1 (r6)
0x001db88f:                 83EC1C      sub     esp, 28
0x001db892:               896C2410      mov     [esp + 16], ebp
0x001db896:               89742414      mov     [esp + 20], esi
0x001db89a:               897C2418      mov     [esp + 24], edi
0x001db89e:                   8BF8      mov     edi, eax
0x001db8a0:                 890424      mov     [esp], eax
```

Comaparison with Full Virtualization:

a. Overhead : We can see that first the dalvik converts bytecode into its own bytecode format and then applies Just-In-Time inpterpretor to convert the instructions to the machine code of the underlying architecture. This has inherent overheads just like any process virtual machine since each instruction is going through a tranlsation layer before it can be executed (which is a tradeoff for portability) whereas in fully virtualization (As we discussed earlier) user mode instructions are directly executed on the processor whereas kernel mode instructions trap into the VMM. Hence full virtualization allows much faster running of instructions since it controls the hardware and call run process instruction directly on the processor. However, Dalvik being a only a software based virtualization system cannot provide this kind of performance features. But several techniqes are used to create optimised DEX using cached versions of previous code chunks which can be loaded directly from the cache rather than interpreting it.

b. Process/Isolation: Since Dalvik VM is a process virtualization environment it allows for a single process to be run within its virtualized environment and the process has no idea beyond that. Hence, it provides a kind of sandbox for the application and hence isolation/protection is achieved between different processes. An example system is shown below:




Hence, we see that Dalvik is able to provide a virtualization environment based on only software which allows application to be portable and run on any platform at the cost of interpretation overhead at every instruction.

References:

1. http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf
2. https://github.com/dogriffiths/HeadFirstAndroid/wiki/How-Android-Apps-are-Built-and-Run
3. https://en.wikipedia.org/wiki/Dalvik_(software)
4. http://www.slideshare.net/jserv/understanding-the-dalvik-virtual-machine

## 4. Static Priority Scheduling of XINU Processes

1. The results obtained on running first experiment is as shown below:

```
[XBOOT] Loading Xinu...
Ethernet Link is Up
MAC address is 98:4f:ee:00:0b:a2

Xinu for galileo -- version #120  (patwaria)  Sat Sep 19 18:29:16 EDT 2015

 250113568 bytes of free memory.  Free list:
           [0x001531E0 to 0x0EFD8FFF]
           [0x0FDEF000 to 0x0FDEFFFF]
     79720 bytes of Xinu code.
           [0x00100000 to 0x00113767]
    131208 bytes of data.
           [0x00117220 to 0x001372A7]


Lab2 - Part A, Problem 4 - Static Priority Scheduling of XINU Processes
P1111111111P2222222222P3333333333P4444444444
```

   This shows that the output of the [main process] is interleaved with that of the child processes. Before spawning a new process [main] prints ‘P’ and resumes it. Since the created process has priority (25) which is higher than that of [main] (INITPRIO = 20), as soon it is added to the readyqueue it is rescheduled in the next step and prints 11. Now since the current process has the highest priority (other processes being [main] and [prnull]) it will continue to be in readyqueue and keeps on printing 11. Since, main gets a chance to run only after [myproc1] ends it will print ‘P’ again and resume [myproc2]. Once again myproc2 will continue executing since it will be the highest priority process in the ready queue till it completes execution. Also, we can note that since XINU has static priority scheduling, the priority of [main] doesn’t dynamically increase even though it’s waiting time is increasing while myproc1(2, 3, 4) executes since it has higher priority.
2. The results of changing INITPRIO to 35 (while keeping child process priority 25) is as shown:

```
[XBOOT] Loading Xinu...
Ethernet Link is Up
MAC address is 98:4f:ee:00:0b:a2

Xinu for galileo -- version #121  (patwaria)  Sat Sep 19 18:53:52 EDT 2015

 250113568 bytes of free memory.  Free list:
           [0x001531E0 to 0x0EFD8FFF]
           [0x0FDEF000 to 0x0FDEFFFF]
     79720 bytes of Xinu code.
           [0x00100000 to 0x00113767]
    131208 bytes of data.
           [0x00117220 to 0x001372A7]


Lab2 - Part A, Problem 4 - Static Priority Scheduling of XINU Processes
PPPP1122334411223344223311441122334422331144
```

   Here [main] has highest priority 35.  Even after starting the child processes it remains the highest priority ready process and executes completely and hence we can see the four P’s printed. After this the other  child processes are schedules in a round-robin manner since all of them have same priority and XINU policy is that if the processes have same priority it will schedule them in round-robin manner. Also, we note that the round-robin is not exactly strictly visible (11 is printed after 22 and 33 in two cases) This could be because [myproc1] was still waiting for last IO request to be handled and [myproc2]'s IO request was handled earlier and hence it came into the readyqueue.

3. The results of setting priorities of children as 20, 20, 20 and 50 is shown:

```
[XBOOT] Loading Xinu...
Ethernet Link is Up
MAC address is 98:4f:ee:00:0b:a2

Xinu for galileo -- version #122  (patwaria)  Sat Sep 19 19:54:37 EDT 2015

 250113568 bytes of free memory.  Free list:
           [0x001531E0 to 0x0EFD8FFF]
           [0x0FDEF000 to 0x0FDEFFFF]
     79720 bytes of Xinu code.
           [0x00100000 to 0x00113767]
    131208 bytes of data.
           [0x00117220 to 0x001372A7]


Lab2 - Part A, Problem 4 - Static Priority Scheduling of XINU Processes
P11P22P33P4444444444112233112233112233112233
```

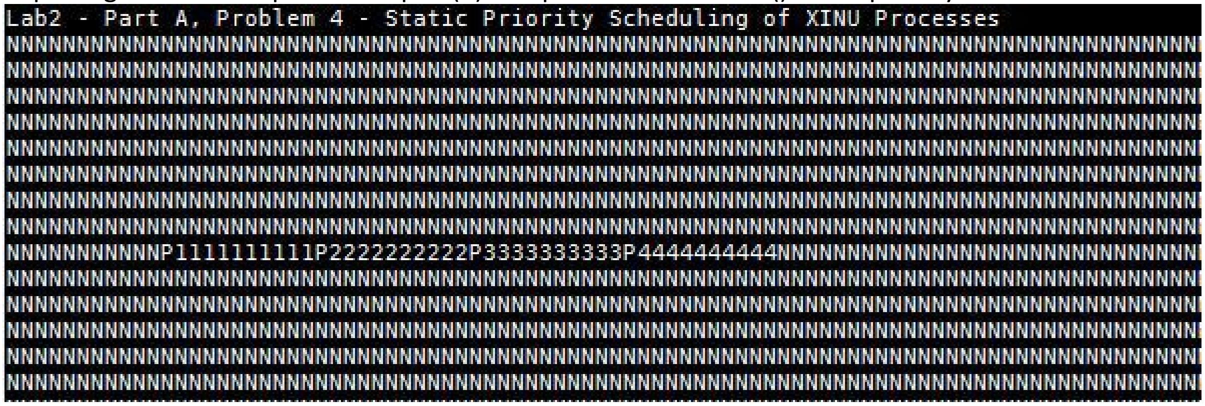
Here it is interesting to see that INITPRIO = 20 and hence processes [main], [myproc1], [myproc2], [myproc3] all have same static priorities. [myproc4] has the highest priority of 50. So we see that till the time [main] resumes [myproc4], the scheduler works in round-robin fashion, however when [myproc4] enters readyqueue it executes all its tasks and prints five '44' before any other process is scheduled. After [myproc4] exits, the scheduler again follows round-robin.

4. Results for sleepms:
    a. Replacing inner for loop with sleepms(1) and print 'N' in nulluser(). Child priority = 25

```
Lab2 - Part A, Problem 4 - Static Priority Scheduling of XINU Processes
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNP11P22P33P4411223344NNN11223344NNNN11223344NNN11223344NNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
```

When we add a sleepms() call instead of the inner for loop the process, it forces the process to go into PR_SLEEP state and is context switched out. So we can see that even though [main] has a lower priority it is scheduled after [myproc1] prints '11' and goes to sleep. [main] resumes [myproc2] which prints '22' and goes to sleep. Similarly for [myproc3] and [myproc4]. After [main] has completed execution scheduling goes in round-robin fashion. However we see that the [prnull] gets a chance to execute and print 'N' in between. This is because at some point all the processes are in a PR_SLEEP state and hence the OS just runs the null process till some other process is ready to be executed.

b. Replacing inner for loop with sleepms(0) and print 'N' in nulluser(). Child priority = 25




Here the sleepms() argument is set to zero. The result is similar to that is experiment 1. This shows that sleepms() with no sleep doesn't change of the current process from PR_CURRENT to PR_SLEEP and it continues to remain in the readyqueue. Since all the child process have higher priority the explanation is similar to that in experiment 1. Each child process first completes all its tasks and then [main] is able to spawn a new process.

5. Proposed implementation of *prcputime*:
   a. Current Clock handling mechanism: The clocktime is handled through the clkhandler which is called in the clkdsip.S which is the interrupt service handler for clock events. The hardware is set to run at 1ms clock ticks and every time a clock tick happens clkhandler is evoked. In clkhandler a global uint32 variable [clktime] is updated on completion of 1000ms.
   b. **Addition**: To calculate the CPU time of each process in ms, I implemented the following scheme. There is an unused variable uint32 [ctr1000] to measure clock time from boot in ms which is initialized to zero. In the clkhandler I update the [ctr1000] by 1 every time the handler is called. Whenever a process is being context switched out/terminated it calls the resched to give up the CPU control. As mentioned, a new field prcputime is introduced in the proctab entry of each process. Whenever a process is being context switched out, its prcputime is updated with the value of ctr1000 – lastclktime (lastclktime) is a static variable introduced in resched which is initialized to 0, and updated to ctr1000 after every call to resched). Hence whenever a process utilizes a time slice and is context switched out the last time slice is added to its prcputime. The result of the implementation can be seen below. The format is as follows: Each line shows the CurTime from boot in ms, process name and its *prcputime* till now. The processes are the same as in previous experiments, [prnull], [main], 4 child processes. [main] sleeps for 5s before exiting during which [prnull] is executing and its prcputime increases.

CurTime: 2, [prnull] CPUTime(ms) : 1

Lab2 - Part A, Problem 4 - Static Priority Scheduling of XINU Processes

CurTime: 9, [Main process] CPUTime(ms) : 7

CurTime: 109, [prnull] CPUTime(ms) : 101

CurTime: 110, [Main process] CPUTime(ms) : 8
P11
CurTime: 111, [myproc1] CPUTime(ms) : 1

CurTime: 112, [Main process] CPUTime(ms) : 9

CurTime: 113, [myproc2] CPUTime(ms) : 1
P11
CurTime: 114, [myproc1] CPUTime(ms) : 2
22
CurTime: 115, [myproc2] CPUTime(ms) : 2

CurTime: 116, [Main process] CPUTime(ms) : 10

CurTime: 117, [myproc1] CPUTime(ms) : 3
22
CurTime: 118, [myproc2] CPUTime(ms) : 3
11
CurTime: 119, [myproc1] CPUTime(ms) : 4

CurTime: 120, [Main process] CPUTime(ms) : 11

CurTime: 121, [myproc2] CPUTime(ms) : 4
11
CurTime: 122, [myproc1] CPUTime(ms) : 5

CurTime: 124, [myproc2] CPUTime(ms) : 6
11
CurTime: 125, [myproc1] CPUTime(ms) : 6
22
CurTime: 126, [myproc2] CPUTime(ms) : 7

CurTime: 127, [Main process] CPUTime(ms) : 12

CurTime: 128, [myproc1] CPUTime(ms) : 7
22
CurTime: 129, [myproc2] CPUTime(ms) : 8

CurTime: 130, [myproc1] CPUTime(ms) : 8

CurTime: 131, [Main process] CPUTime(ms) : 13
22
CurTime: 132, [myproc2] CPUTime(ms) : 9

CurTime: 133, [Main process] CPUTime(ms) : 14

CurTime: 134, [myproc3] CPUTime(ms) : 1

CurTime: 135, [myproc2] CPUTime(ms) : 10
P33
CurTime: 137, [myproc3] CPUTime(ms) : 3

CurTime: 138, [Main process] CPUTime(ms) : 15

CurTime: 139, [myproc4] CPUTime(ms) : 1
P33
CurTime: 140, [myproc3] CPUTime(ms) : 4
44
CurTime: 141, [myproc4] CPUTime(ms) : 2

CurTime: 142, [Main process] CPUTime(ms) : 16

CurTime: 143, [myproc3] CPUTime(ms) : 5
44
CurTime: 144, [myproc4] CPUTime(ms) : 3

3
CurTime: 146, [myproc3] CPUTime(ms) : 7
44
CurTime: 147, [myproc4] CPUTime(ms) : 4
3
CurTime: 148, [myproc3] CPUTime(ms) : 8

CurTime: 149, [Main process] CPUTime(ms) : 17

CurTime: 150, [myproc4] CPUTime(ms) : 5
33
CurTime: 151, [myproc3] CPUTime(ms) : 9
44
CurTime: 152, [myproc4] CPUTime(ms) : 6

CurTime: 153, [Main process] CPUTime(ms) : 18

CurTime: 154, [myproc3] CPUTime(ms) : 10
4
CurTime: 156, [myproc4] CPUTime(ms) : 8

CurTime: 157, [myproc3] CPUTime(ms) : 11
4
CurTime: 158, [myproc4] CPUTime(ms) : 9
33
CurTime: 159, [myproc3] CPUTime(ms) : 12

CurTime: 160, [Main process] CPUTime(ms) : 19

CurTime: 161, [myproc4] CPUTime(ms) : 10

CurTime: 162, [myproc3] CPUTime(ms) : 13

CurTime: 163, [myproc4] CPUTime(ms) : 11

CurTime: 164, [Main process] CPUTime(ms) : 20

CurTime: 5164, [prnull] CPUTime(ms) : 5101

CurTime: 5165, [Main process] CPUTime(ms) : 21

We note that this scheme is able to add even the last time slice before a process is exiting, since before exiting the process make a call to resched and its *prcputime* will be updated to include the last time slice as well. At the end [prnull] has a total time of 5101ms because, it was executing for 5000ms while [main] was sleeping.