CS552 Operating Systems **Fall 2013**

Assignment 1

Assigned: October 9 *Due: October 28*

- **All questions must be answered on an individual basis.**
- **Homework is to be submitted using gsubmit by 11:59pm on the due date.**
- **State all assumptions if something seems ambiguous.**
- **START EARLY! Do not wait until the due date.**

1. **Protection and Interrupt-Handling:**

   (a) Consider a dual-mode system in which the kernel is separated from user-level applications using two "rings of protection".

   - (**4 points**) Is it safe for the kernel to directly access user-level memory in any process address space? Briefly explain your answer.
     $\Longrightarrow$ *It is safe for the kernel to directly access user-level memory in any process address space. "Directly access" implies being able to read/write user-level memory. The kernel can do this as it copies to/from user-space all the time when e.g. doing call-by-reference parameter passing.*
   - (**6 points**) Is it safe for the kernel to directly execute *user-level code* in *any* process address space? Again, briefly explain your answer.
     $\Longrightarrow$ *It is not safe. "Directly execute" in this context means to call a function at user-level e.g. by jumping to an instruction address. The kernel should not place its trust in (untrusted) user level code. Side effects may violate integrity of system. However, returning to user-level is typically accepted by hardware supporting at least two rings of protection.*
   - (**8 points**) Systems such as UNIX allow processes to associate signal handlers with various events. Unlike interrupts, signals are not handled until the target process is active. Briefly explain one CPU, memory and I/O protection problem if we allowed a signal handler for process $p1$ to execute in the context of another process, $p2$. If we could address these protection problems, what would be the advantage of this scheme?
     $\Longrightarrow$
     CPU problem: the signal handler of p1 might execute for an extremely long time which will affect the progress of process p2.
     MEMORY problem: the signal handler of p1 might write to the memory address space of p2 so that it "pollutes" content in the memory of p2.
     I/O problem: a handler for P1 executing in the context of P2 may e.g. read from a file descriptor for P2, obtaining sensitive data.
     If we could address these protection problems, we can reduce latency between kernel event/signal occurrence and response after event/signal delivery and processing. Also, this solution is independent of the kernel scheduling policy, e.g. static priority scheduling may never schedule a low priority process with pending signals.

   (b) (**8 points**) Briefly describe *four* actions that a program should *not* be able to do in *user* mode (as opposed to *kernel* mode). Give reasons for each of your answers.
   (1) Execute privileged instructions (e.g., to manipulate the interrupt vector table), (2) directly access hardware devices without being granted such privilege by the trusted kernel, (3) dis-

able interrupts, (4) access arbitrary regions within the address space of the kernel or another process.

(c) **Linux Interrupt Handling:** Many systems split interrupt handling into two parts: a top half and a bottom half. For example, Linux top halves respond immediately to interrupts, while bottom halves are deferrable.

i. (**4 points**) Why are interrupts handled in two parts in systems such as Linux?
$\Longrightarrow$ *ISRs often run with interrupts disabled to avoid reentrancy problems. Splitting interrupts into a first part that is handled quickly at the time of the interrupt allows subsequent interrupts to be handled without loss, as would be the case if long-lived ISRs ran with interrupts turned off.*

ii. (**2 points**) In what sense is a bottom half deferrable?
$\Longrightarrow$ *It can be scheduled to execute at a later time.*

iii. (**6 points**) Explain how this approach allows a device (e.g., a Ethernet network interface card) to generate multiple interrupts before the first one from the device has been completely handled.
$\Longrightarrow$ *Subsequent interrupts can be posted as pending events for deferred handling, at a convenient time. In this case, the events are handled in threads that have their own machine context (including a private stack) and are schedulable. Note that only a brief part of the interrupt handling, known as the top-half, needs to be performed when the interrupt occurs.*

iv. (**6 points**) Using sources of information on the Internet (citing your references) explain what is meant by "receiver livelock" due to device interrupts. How could this be resolved using bottom half interrupt handling?
$\Longrightarrow$ *Receiver livelock occurs when a target machine is effectively spending all its time processing interrupts or handling service requests from a device such as a NIC and is not able to make progress with other activities. The term livelock, here, refers to some entity that is being starved of service, which is typically other processes on the target host.*

v. (**6 points**) Explain one problem with Linux bottom half interrupt handling, and provide a solution to the problem.
$\Longrightarrow$ *One problem is that interrupts are given immediate precedency over threads/processes currently running on the CPU. The interrupt may, however, be the result of a prior IO request from a low priority (perhaps, now blocked) thread. Hence, we can end up with a form of priority inversion. To guard against this, and to allow conventional threads to make progress without waiting for a long stream of interrupts, Linux defers interrupt events arriving above a certain threshold to lowest priority threads. This means that interrupt handling effectively switches from highest to lowest priority, and is nonetheless decoupled from the priority of other threads. A better solution would be to determine the priority of an interrupt and integrate its scheduling with that of conventional threads.*

2. **Process Creation and Execution:**

(a) (**8 points**) In pseudo-code, explain how a *vfork()* system call might be implemented, in which case a child process is created and executed ahead of the waiting parent, until either the child calls exit() or exec's a new program image in its address space. Explain what is happening in terms of resource allocation and control over which process runs when.
$\Longrightarrow$ *Create child process control block with new process ID for child*

```
// Child does not get a copy of the parent address space,
```

```
// but instead shares the address space of the parent.
waitpid(child); // Parent is suspended on progress of child
schedule(child); // Call scheduler to execute child

//In child
Continue execution or call exec()
exit();

//In kernel, on either exec() or exit() from child
wakeup(parent); // Move parent to ready queue for possible
                   scheduling/execution
```

(b) (**10 points**) Draw a diagram of the relationship between physical and virtual memory for the parent and child processes when the parent has just invoked *vfork()*. NOTE: you do not need to worry about paging, just the general relationship. Show pseudo-code for an example where program correctness (e.g., a race condition occurs) if the vfork() call did not enforce the child went first but instead allowed the parent and child processes to interleave their execution.

$\Longrightarrow$ *Both parent and child virtual address spaces should map to the same physical memory.*

$\Longrightarrow$ *An example race condition could be if both the parent and child wish to modify a variable, e.g., x++. If the parent reads the value of x into a register and then is preempted by the child, the child may go ahead and similarly try to update x. Should the child succeed and write back to memory, on return to the parent, the changes performed by the child will be overwritten and hence lost. If the parent and child really had separate physical memories for their address spaces, the variable x would not be shared and this would not be a problem.*

3. **Process/Thread Scheduling:** Consider the following set of processes, with all times in milliseconds.

Process	Arrival Time	Burst/Computation Time	Priority (1=highest)	Deadline
P1	3	2	1	7
P2	0	2	3	5
P3	2	3	4	15
P4	6	4	2	11
P5	1	5	5	20

(a) (**15 points**) Draw the Gantt charts illustrating the execution of these processes using the following scheduling algorithms:

- Static Priority with preemption.
  $\Longrightarrow$*Result is in Fig. 1*
- Round-Robin, no priorities, quantum = 1.
  $\Longrightarrow$*Result is in Fig. 2*
- Earliest Deadline first, no preemption.
  $\Longrightarrow$*Result is in Fig. 3*

(b) (**6 points**) What are the *average waiting* and *average turnaround* times for each of the processes, using the scheduling algorithms above?

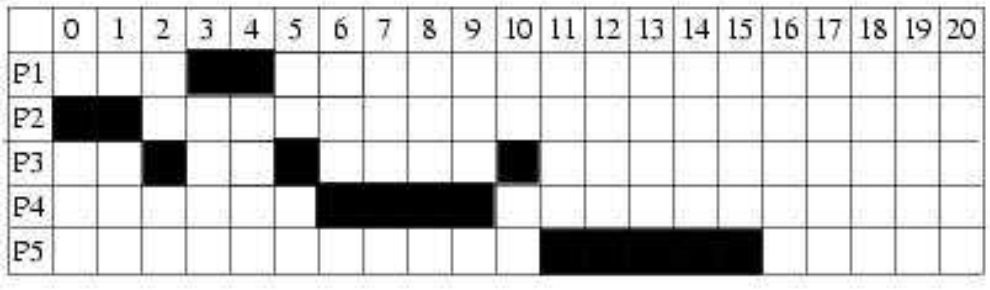

Figure 1: Static Priority

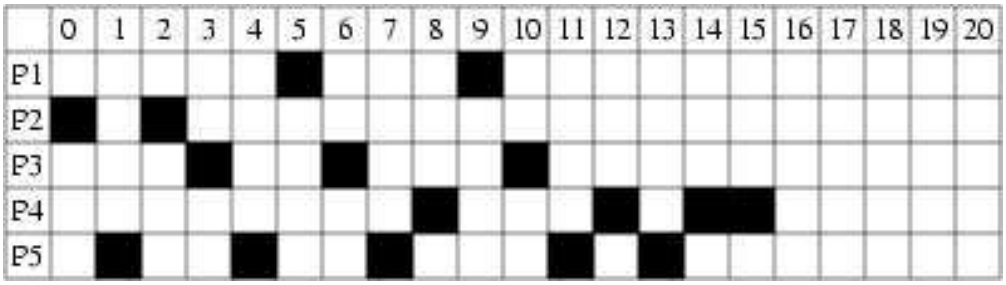

Figure 2: Round Robin

- *Static Priority:*
  Average waiting time = $\frac{0+0+6+0+10}{5} = 3.2$
  Average turnaround time = $\frac{2+2+9+4+15}{5} = 6.4$
- *Round-Robin*
  Average waiting time = $\frac{5+1+6+6+8}{5} = 5.2$
  Average turnaround time = $\frac{7+3+9+10+13}{5} = 8.4$
- *EDF*
  Average waiting time = $\frac{2+0+0+1+10}{5} = 2.6$
  Average turnaround time = $\frac{4+2+3+5+15}{5} = 5.8$

(c) Rate Monotonic Scheduling (RMS) is a well-known, *preemptive* scheduling algorithm for scheduling real-time, *periodic* processes. A periodic process, P, has a request period, T, and a computation (or burst) time, C, such that P is ready to start execution at time t, $t + T \cdots t + nT \mid n = 1, 2, \cdots$ etc. Moreover, P must execute for exactly C time units every request period T, for a *feasible* schedule to exist. That is, if P starts execution at time t, it must complete its execution of C time units by time $t + T$. This requirement continues for all subsequent request periods.

RMS selects the highest priority process for execution, where the highest priority process is the one with the smallest request period.

i. (**6 points**) If the following three periodic processes all require scheduling for the first time, at time t, can a feasible schedule be constructed with RMS? What happens if we use earliest deadline first scheduling, assuming deadlines are at the ends of request periods? Briefly explain your answer.
$\Longrightarrow$From Fig. 4, RMS can not construct a feasible schedule. P3 fails to meet it's deadline. But from Fig. 5, EDF can give a feasible schedule.

ii. (**3 points**) Can RMS yield a feasible schedule for the following periodic processes? Briefly

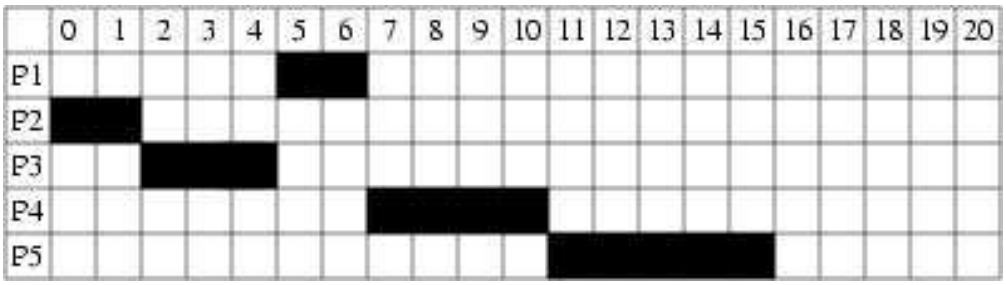

Figure 3: EDF

Process	Computation Time, C	Request Period, T
P1	2	8
P2	3	10
P3	9	20

explain your answer.
$\Longrightarrow$*From Fig. 6, we see that RMS can yield a feasible schedule.*

iii. (**6 points**) What is the highest utilization by a set of processes that yields a feasible schedule with RMS, if all processes have *harmonically-related* request periods? Prove your answer. NOTE: harmonically-related request periods implies each period is a multiple of all smaller periods.
$\Longrightarrow$
The highest utilization by a set of processes that yields a feasible schedule with RMS is 1.0*.*

proof:
Suppose there are in total N *requests/tasks* $\{\tau_i | 1 \leq i \leq N\}$ *starting at time 0, with the request period* $\{T_i | 1 \leq i \leq N\}$*. We sort tasks in increasing order:* $T_1 \leq T_2 \leq T_3 \leq \ldots \leq T_N$*. The corresponding execution time for each task in one period is* $\{C_i | 1 \leq i \leq N\}$*. Given* $\sum_{i=1}^{N} \frac{C_i}{T_i} = 1$*, we want to prove that we can feasibly schedule for each task before it's deadline* T_i*. We give the proof by induction.*

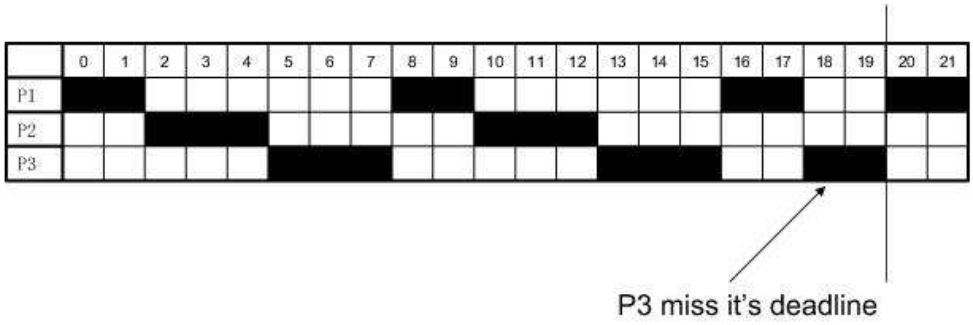

Figure 4: RMS none-harmonically-related

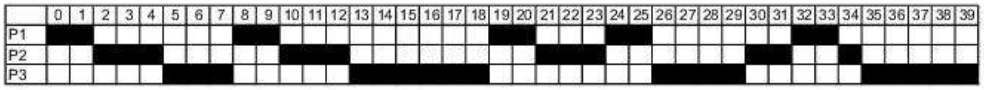

Figure 5: EDF none-harmonically-related

Process	Computation Time, C	Request Period, T
P1	3	6
P2	3	12
P3	6	24

The worst finish time W_i for τ_i in one of the request periods T_i should be:

$$W_i = \begin{cases} C_1 & \text{for } i = 1 \\ C_i + \sum_{j=1}^{i-1} \frac{T_i}{T_j} C_j & \text{for } 2 \leq i \leq N \end{cases}$$

Intuitively, the equation can be explained as follows: For τ_1, with the highest priority, it will finish at time C_1. For τ_i, if we can feasibly schedule tasks from τ_1 to τ_{i-1}. The worst finish time should be that we let all the tasks with higher priorities be served first, thereby satisfying all their quota for this whole period. The time is $\sum_{j=1}^{i-1} \frac{T_i}{T_j} C_j$. Since this is a preemptive algorithm, we can run τ_i in the time slots left by higher tasks, so for the finish time W_i of τ_i, we can just add it's execution time C_i.

Now, we prove that for every i, $W_i \leq T_i$. Given that $\sum_{j=1}^{N} \frac{C_j}{T_j} = 1$, we get:

$$\sum_{j=1}^{i} \frac{C_j}{T_j} \leq \sum_{j=1}^{N} \frac{C_j}{T_j} \leq 1 \text{ for } 1 \leq i \leq N \tag{1}$$

So

$$\begin{aligned} \frac{W_i}{T_i} &= \frac{C_i}{T_i} + \frac{\sum_{j=1}^{i-1} \frac{T_i}{T_j} C_j}{T_i} \\ &= \frac{C_i}{T_i} + \sum_{j=1}^{i-1} \frac{T_i}{T_i} \frac{C_j}{T_j} \\ &= \sum_{j=1}^{i} \frac{C_j}{T_j} \\ &\leq 1 \end{aligned}$$

So

$$W_i \leq T_i \; \forall i \in [2, N]$$

Here, we get that given $\sum_{j=1}^{N} \frac{C_j}{T_j} = 1$, $\forall \tau_i$, it's finish time before it's deadline, so it can be feasibly scheduled. So we can achieve full system utilization.

(d) (**16 points**)

EEVDF (Earliest Eligible Virtual Deadline First) is a proportional share scheduling policy. Given three tasks with weights of 1, 3, and 4, draw a timeline schedule on a uniprocessor for

these tasks using EEVDF. Extend your timeline between t=0 to t=24. Assume that the tasks execute without blocking in quanta of size 1 and continue to execute indefinitely. Draw the same scheduling timeline for this task set on a dual processor. In each case, show the virtual deadlines of each task at every scheduling point.

See Fig. 7

For Dualprocessor,

Here, it should be noted that deadline scheduling on two or more processor is NP-hard. What we are looking for is scheduler that assigns EEVDF-ordered tasks to each CPU as soon as its free. See Fig. 8

4. **Virtualization:**

For this question you will need to study extra sources of information, such as technical papers and web pages. Please cite any sources of information with your answers.

- (**12 points**) Before the introduction of hardware virtualization support, the x86 architecture was not considered virtualizable in the "trap and emulate" sense. This was because of approximately 17 sensitive, unprivileged instructions. Briefly describe what is meant by a "sensitive instruction" in this case and give an example of how it is potentially problematic when constructing a virtual machine. How then can the x86 support VMs when not all instructions are virtualizable?

  "sensitive instruction" in this case means that instructions are sensitive if (1) they read or change sensitive registers and/or memory locations such as a clock register and interrupt register; (2) they reference the storage protection system, memory or address relocation system. See details in (J. Scott Robin and Cynthia Irvine, "Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor", USENIX Security Symposium, 2000)

  Normally, these "sensitive instructions" should be privileged instructions so that when a virtual machine runs at user-level and issues such instructions, it will cause an exception and invoke the exception handlers in a virtual machine monitor (VMM), and exception or signal handlers in the VMM will EMULATE the execution of this sensitive instruction.

  If these instructions are not privileged, they won't be trapped by the OS and may pollute machine states.

  The ways for the x86 to support VMs include:

  1. *Pure Emulation, which interprets every instruction of the guest OS. This will have a significant performance degradation.*
  2. *Dynamic translation and interpretation. That is, search binary code before execution and replace all sensitive non-privileged instructions with code that will trap to the VMM indicating the cause of trap, where it can be emulated.*

- (**10 points**) Explain how one could virtualize system calls issued by P_{guest} so that they are serviced by kernel code of the guest OS (mapped into the address space of P_{guest}). How do you differentiate and control the switching between application and kernel stacks in P_{guest}? In your answer, include a diagram that shows how control is redirected between various parts of memory, to handle virtualized system calls.

  Refer to Fig. 9.

  When an application running on an OS that wants to perform a system call, it executes a trap instruction with the arguments properly placed on the stack. Here, it will trap into VMM. VMM will have its trap handler which will perform upcall to guestOS with arguments of system call

placed on a stack. VMM has information about the guestOS trap handler because when the guestOS tried to install its own trap handlers, it was trying to execute priviledged code, and therefore trapped into the VMM. At that time, the VMM would have stored the information about guestOS trap handler in memory. The guestOS execute this upcall as it came from user appplication. When guestOS has finished execution of trap handler, it will try to return from system call which will again trap into VMM. VMM finds that OS tried to return from system call, so it exectutes return from trap with return value placed on stack and return back to user application.

Notice that here we need context switch from guestOS to hypervisor. Hardware with VT support could enable the syscall from application process to be intercepted and directed to guestOS without needing intervention of the hypervisior.

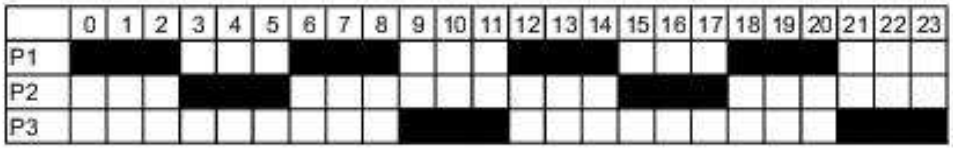

Figure 6: RMS harmonically-related

Sheet1

	Task 1				Task 2				Task 3				Schedule
t	W1	Ve	Vd	r	W2	Ve	Vd	r	W3	Ve	Vd	r	CPU1
0	1	0	1	1	3	0	0.33	1	4	0	0.25	1	3
1	1	0	1	1	3	0	0.33	1	4	0.25	0.5	1	2
2	1	0	1	1	3	0.33	0.67	1	4	0.25	0.5	1	3
3	1	0	1	1	3	0.33	0.67	1	4	0.5	0.75	1	2
4	1	0	1	1	3	0.67	1	1	4	0.5	0.75	1	3
5	1	0	1	1	3	0.67	1	1	4	0.75	1	1	1
6	1	1	2	1	3	0.67	1	1	4	0.75	1	1	2
7	1	1	2	1	3	1	1.33	1	4	0.75	1	1	3
8	1	1	2	1	3	1	1.33	1	4	1	1.25	1	3
9	1	1	2	1	3	1	1.33	1	4	1.25	1.5	1	2
10	1	1	2	1	3	1.33	1.67	1	4	1.25	1.5	1	3
11	1	1	2	1	3	1.33	1.67	1	4	1.5	1.75	1	2
12	1	1	2	1	3	1.67	2	1	4	1.5	1.75	1	3
13	1	1	2	1	3	1.67	2	1	4	1.75	2	1	1
14	1	2	3	1	3	1.67	2	1	4	1.75	2	1	2
15	1	2	3	1	3	2	2.33	1	4	1.75	2	1	3
16	1	2	3	1	3	2	2.33	1	4	2	2.25	1	3
17	1	2	3	1	3	2	2.33	1	4	2.25	2.5	1	2
18	1	2	3	1	3	2.33	2.67	1	4	2.25	2.5	1	3
19	1	2	3	1	3	2.33	2.67	1	4	2.5	2.75	1	2
20	1	2	3	1	3	2.67	3	1	4	2.5	2.75	1	3
21	1	2	3	1	3	2.67	3	1	4	2.75	3	1	1
22	1	3	4	1	3	2.67	3	1	4	2.75	3	1	2
23	1	3	4	1	3	3	3.33	1	4	2.75	3	1	3
24	1	3	4	1	3	3	3.33	1	4	3	3.25	1	3

Figure 7: EEVDF Uni-processor Schedule

Sheet1

	Task 1				Task 2				Task 3				Schedule	
t	W1	Ve	Vd	r	W2	Ve	Vd	r	W3	Ve	Vd	r	CPU1	CPU2
0	1	0	1	1	3	0	0.33	1	4	0	0.25	1	3	2
1	1	0	1	1	3	0.33	0.67	1	4	0.25	0.5	1	3	2
2	1	0	1	1	3	0.67	1	1	4	0.5	0.75	1	3	1
3	1	1	2	1	3	0.67	1	1	4	0.75	1	1	3	2
4	1	1	2	1	3	1	1.33	1	4	1	1.25	1	3	2
5	1	1	2	1	3	1.33	1.67	1	4	1.25	1.5	1	3	2
6	1	1	2	1	3	1.67	2	1	4	1.5	1.75	1	3	1
7	1	2	3	1	3	1.67	2	1	4	1.75	2	1	3	2
8	1	2	3	1	3	2	2.33	1	4	2	2.25	1	3	2
9	1	2	3	1	3	2.33	2.67	1	4	2.25	2.5	1	3	2
10	1	2	3	1	3	2.67	3	1	4	2.5	2.75	1	3	1
11	1	3	4	1	3	2.67	3	1	4	2.75	3	1	3	2
12	1	3	4	1	3	3	3.33	1	4	3	3.25	1	3	2
13	1	3	4	1	3	3.33	3.67	1	4	3.25	3.5	1	3	2
14	1	3	4	1	3	3.67	4	1	4	3.5	3.75	1	3	1
15	1	4	5	1	3	3.67	4	1	4	3.75	4	1	3	2
16	1	4	5	1	3	4	4.33	1	4	4	4.25	1	3	2
17	1	4	5	1	3	4.33	4.67	1	4	4.25	4.5	1	3	2
18	1	4	5	1	3	4.67	5	1	4	4.5	4.75	1	3	1
19	1	5	6	1	3	4.67	5	1	4	4.75	5	1	3	2
20	1	5	6	1	3	5	5.33	1	4	5	5.25	1	3	2
21	1	5	6	1	3	5.33	5.67	1	4	5.25	5.5	1	3	2
22	1	5	6	1	3	5.67	6	1	4	5.5	5.75	1	3	1
23	1	6	7	1	3	5.67	6	1	4	5.75	6	1	3	2
24	1	6	7	1	3	6	6.33	1	4	6	6.25	1	3	2

Figure 8: EEVDF Dual-processor Schedule

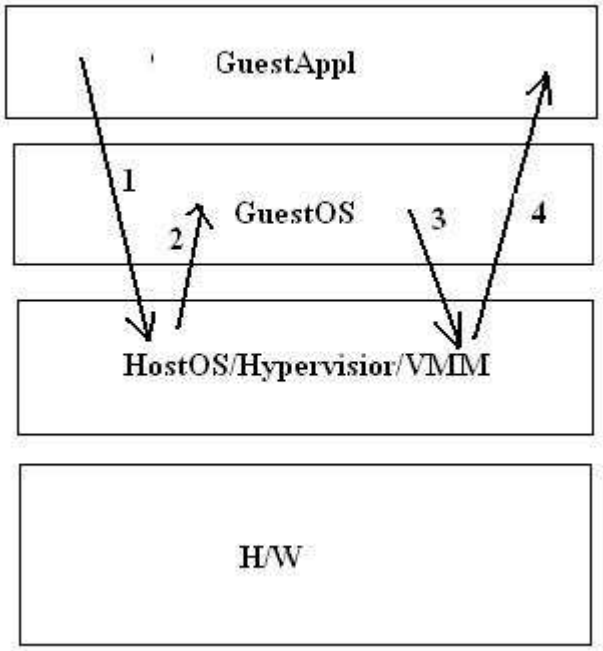

Figure 9: