CS552 Assignment 1

BY HANWEN WU

hwwu@bu.edu

Table of contents

1 Protection and Interrupt-Handling

Question. *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.

(I am not sure about the meaning of "directly". In my anwser, I assume it to be "unrestricted" or "unchecked".)

No. If kernel has the ability to do so, users can ask kernel to access other address space directly without any restriction. This violates the principle of isolating processes.

Also, in a practical design, the kernel part is trusted, but the userland is not. Whenever the kernel wants to access untrusted userland, there must be some kind of checks or restrictions.

Question. *(6 points) Is it safe for the kernel to directly execute user-level code in any process address space? Again, briefly explain your answer.*

No. If the kernel execute user-level code in some process address space directly, the code will be run in the context of kernel-level, which has ring 0 privilege. If the user supports some malicious code to the kernel, it could probably do anything harmful.

Question. *(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 p_1 to execute in the context of another process p_2. If we could address these protection problems, what would be the advantage of this scheme?*

Signals are handled right before the process returns to user space. Kernel will do a complex setup for both kernel stack and user stack. In short, it sets up kernel stack so that it can return to the handler code, instead of original process resume point. It sets up user stack so that the handler has its own stack frame, holds original user context, and a return address that points to a special syscall function to return to kernel. And when the signal handler returns to the kernel, it then copies back those original user context to return to the very beginning where the process blocks.

If it is currently running in p_2, and a signal has occured for p_1, then kernel simply sets corresponding data in p_1's task control block to say that some signal arrives, and possibly turns p_1 into ready queue. When next schedule point comes, p_1 may get scheduled and the signal handler is called.

This means, it is impossible to address p_1's handler code in p_2's address space, since the code is in p_1's user code segment, not p_2's. And since the kernel has to setup p_1's kernel stack and user stack in order to force the handler to run, it has to be in the context of p_1.

Question. *(8 points) Consider the implementation of a micro-kernel on an architecture such as the Intel x86. Suppose this OS structure allows device drivers to be implemented at user-level. On the x86 architecture a general protection fault occurs if we attempt to trap to a ring of protection less privileged than the current protection domain. Briefly explain how we can invoke user-level device drivers from the kernel (by essentially performing the opposite of a system call). Show pseudo-code or a diagram, if necessary, to help with your explanation.*

This question is nearly the same with the VMM question, since the VMM needs to transfer control from ring 0 to ring 1 guest kernel. The details are presented in that answer with the help of Appendix A explaining hardware context switch and GDT, IDT.

In short, after segmentations are setup, the kernel can perform a long jump or a long call, specifying a segment selector, to switch run level. It can switch run level because of x86 hardware switch machanism. Please refer to the question asking about guest syscall, and Appendix A.

Also, signals and event queues can be used to invoke user-level drivers, or the kernel can allow a user thread to wait for an interrupt by reading a special file [LCFG+05].

When the driver wants to handle some interrupts, it registers itself to the kernel. Kernel can map the interrupt to some signals or events, so that the driver gets executed. How signal handler can be invoked in user-level is explained in the last question. Device drivers can also open special files, like `/proc/irq/...` to install itself as a handler. When an interrupt occurs, the in-kernel handler disables the interrupt and increments a per-interrupt semaphore. When the user process does a `read()` on the file, the value of the semaphore, if non-zero, is returned. Otherwise the interrupt is enabled, and the process sleeps on the semaphore. Once an interrupt is received, the driver does whatever is necessary to acknowledge the interrupt on the card.

Question. *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.*

(4 points) Why are interrupts handled in two parts in systems such as Linux?

Mordern systems are going to handle so many interrupts in a short time. In order to response to as much interrupts as possible, the system needs to execute handlers as quickly as possible. Thus, a big handler could be decomposite into top half and bottom half. Those must-done and time-sensitive codes are in the top half, and they should be quick. Those more complex but not time-sensitive logics are encapsulated into bottom half, and they are relatively time comsuming. If we do not do so, some of the interrupts will not be servered, and may be simply ignored because the system is executing some other time comsuming handlers.

Question. *(2 points) In what sense is a bottom half deferrable?*

If it is not time-sensitive, it is more or less deferrable. Take the device driver as an example. When interrupt comes, the time-sensitive tasks could be saving device data into memory buffer, and they should be in the top half because when the next interrupt comes, those device data may be over-written. However, processing those data, awaking waiting processes, could be deferrable, because it is not likely to be affected by the next interrupt or other time related issues. "Deferrable" doesn't mean not important, they should also be executed timely. But it is deferrable compared to those time-sensitive codes in the top half.

Question. *(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.*

1. The first interrupt comes, and TopHalf 1 (TH1) is executed, to copy device data to driver buffer. BottomHalf 1 (BH1) gets queued.
2. The second interrupt comes, and TH2 is executed, device data is appended to the driver buffer. BH2 gets queued.
3. ...
4. (Suppose that the system is not busy handling NIC interrupts anymore, then) BH1 is executed, and the data in the buffer is processed and reported to its user.
5. BH2 is executed, and more data is processed and reported to its user.
6. ...

Such schema could happen because bottom half is deferrable. Top halves are of higher priority. When interrupts come frequently, the system will try to response them all with the help of quick top halves. And the heavy jobs in the bottom halves will be deferred until the system is not busy.

Actually, I believe that it is possible to implement a top half that only register the bottom half once (if there isn't a queued bottom half), and the bottom half could handle all the data in the buffer generated by all previous top halves.

Question. *(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?*

According to [Var04], "receiver livelock" refers to the phenomenon that under high load, the system spends all its time processing incoming interrupts, and only to discard them later because the applications who consume them never run. Such receiver livelock is more likely to happen on network devices. It is called livelock because it is locked, but not deadlocked.

According to [Sha04], the problem could be solved using the following schema.

1. The first interrupt comes, and the top half gets run.
2. The top half registers a bottom half, and prevent further interrupts from generating. (But the data still comes in.)
3. Top half finishes. And bottom half gets run. It will process all the data in the buffer. If they have all been processed, it will re-enable interrupts.
4. Bottom half finishes, and other bottom halves get run. Note that the same bottom half will be activated at most once in a round. If there is still the same bottom half needs to be run, it will run in Step 6.
5. All bottom halves are finished. User processes gets run.
6. When schedule point comes, some bottom halves metioned in Step 4 are activated. If all the data is processed, they re-enable their corresponding interrupts. And jump to Step 1.

The benifit of this schema is that, even if the data comes in like flood, there will be only one interrupt for the first piece of data. As a result, the top half is guaranteed to finish, and the bottom half is guaranteed to be activated (Step 1 to Step 3). Also, the bottom half will be activated only once before the user processes, so that the user processes are guaranteed to be executed (Step 4 to Step 5). And if the data is processed thoroughly, interrupt is re-enabled. If it is not, further data will be sliently dropped.

If we do not have a bottom half, there is no way to enable interrupt for the first piece of data, while disable the interrupts for further data until data in the buffer are processed.

Question. *(6 points) Explain one problem with Linux bottom half interrupt handling, and provide a solution to the problem.*

It has fairness problem.

Currently, Linux bottom half is executed just before returning to user space. It will pick one queued bottom halves from each IRQ, process all of them, and back to user space. This will happen whenever there is a kernel/user switch. However, it is not fair for the current process.

Assume that p_1 is currently doing CPU bound tasks, p_2 has required a lot of I/O preiously. Now, all the I/O request have finished and a lot of bottom halves are going to be executed before returning from p_1 kernel space back to p_1 user space. Alghouth there is no more "receiver livelock" any more for p_1, it has been still delayed due to p_2's bottom halves. That is not fair for p_1.

A better solution is to execute corresponding bottom halves before a switch, instead of executing all bottom halves before every switch. It means, a bottom half is delayed until it is in the context of the requester. This is reasonable. Although it seems that bottom halves are delayed, it is indeed not. In Linux, bottom halves are executed sooner, but the output of bottom halves will not be processed by the requester until it gains CPU time! According to the principle of delaying execution until the last minite, as seen in Copy-On-Write, we can just delay the execution of bottom halves until it is really necessary, that is right before switching to the requester user space.

2 Process Creation and Execution

Question. *(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.*

In the following, I assume that every function call will succeed, and I will ignore error handling since I am going to focus on the "normal" logic. This pseudo code is largely referenced from the text description in [BC05].

```
vfork (parent)
{
   pid = allocate_pid ();
   tcb = allocate_tcb ();

   copy_proc_descriptor (tcb, parent->tcb);
   init_proc_pid (tcb, pid);
   init_proc_state (tcb); //reset related states, counters, schedule info, etc.
   share_open_files (parent->tcb);  //just share pointers, do not copy
   share_file_system (parent->tcb); //just share ...
   share_sig_handler (parent->tcb); //just share ...
   share_memory_map (parent->tcb);  //just share ...
   share_user_stack (parent->tcb);  //just share ...

   copy_thrd_descriptor (tcb, parent->tcb); //especially kernel stack info
   init_kernel_stack (tcb); //reset eax(child return value), esp, return address, etc.

   init_exit_signal (tcb, SIG_CHILD); //child issue SIG_CHILD when exit() or exec()
   block_parent (parent, wait_queue, SIG_CHILD, pid); //block parent to wait

   register_tcb (system_tcb_queue, tcb); //make new tcb visible in the system
   schedule_child (pid); //schedule it

   return pid;
```

```
}
```

When the parent calls `vfork()`, the kernel will allocate new kernel data structures to identify the newly created process. But, not everything is copied into the new `tcb` from the parent. Instead, most of the resources are shared with the parent, including user stack and memory map. This means, the child shares nearly every important resource with its parent, especially the address space. If the child modifies some data, the parent will be affected (No COW happends here). However, kernel stack is private. The kernel will setup a new kernel stack for the child, and initialize it with new values so that the kernel could return to the user space (of the child) from the kernel space correctly.

Next, the `vfork()` will put the parent into a wait queue, waiting the child to terminate. When the child terminates, it will generate a signal, to wake up its parent. This guarantees that the child runs before its parent. If the child `exec()`, it will replace the address space reference to its parent with a newly created one, and then unblock its parent.

Question. *(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.*

In Figure 1, the virtual address space and the physical memory are shown. Parent process and child process share the same virtual address space, which contains kernel space and user space. The TCBs are stored in the kernel space. They have different TCBs, and different kernel stacks. (But some fields of the TCBs may point to the same address.) They are in different threads, but they share the same user stack. (They are in different threads because they use different kernel stacks, in which store hardware context including program counter.) Since they are using the same virtual address space, and the same memory mapping, they are actually using the same physical memory, too.

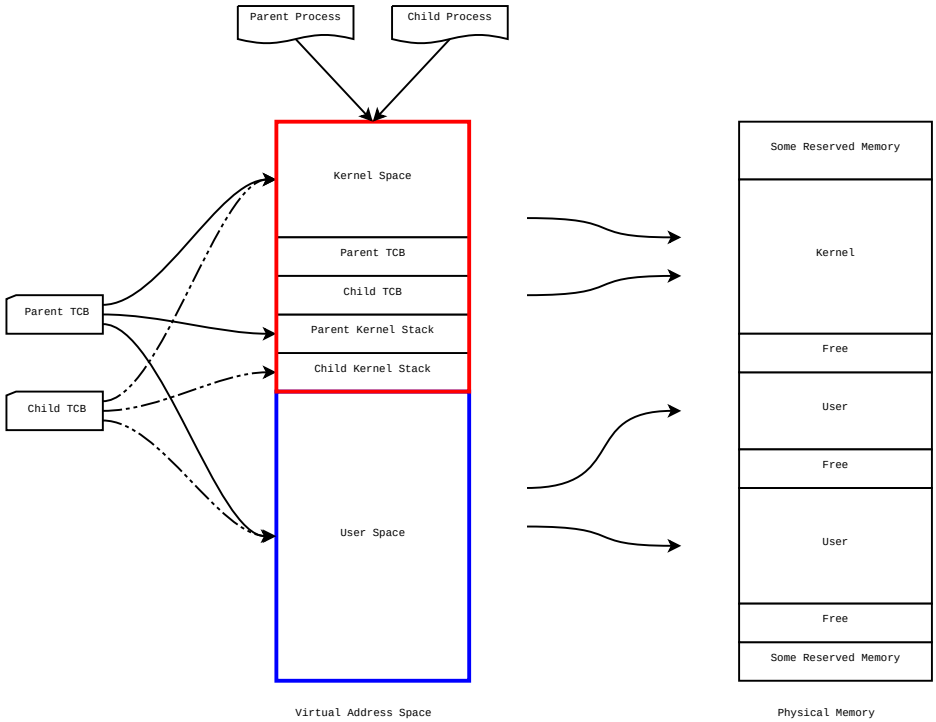

Figure 1. Address Space after `vfork()`

(I am not quite clear about the question itself. I assume that you are asking to show the race condition problems caused by not enforcing the child to run before its parent.)

The following code shows if the child is not forced to run first, then the variable value at the end is unpredictable. This is because the child and the parent is sharing the same user space, including the stack frame. Thus, the same symbol `counter` is simply the same memory address. One modifies, the other gets affected.

```
main ()
{
   count = 0;
   pid = vfork ();
   if (pid = 0)     //child
      count++;
      exit();
   else             //parent
      count++;

   print (count);   //unpredictable, either 2 (child first) or 1 (parent first)
}
```

3 Virtualization

Question. *(12 points) Before the introduction of hardware virtualization support, 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. What is meant by the term "trap and emulate"? How then can the x86 support VMs when not all instructions are virtualizable?*

According to [PG74], "sensitive instruction" means it attempts to change the configuration of resources in the system (control sensitive), like CPU states, or they behave differently due to different resource configurations (behaviour sensitive). "Unprivileged" means those instructions could run in user mode without trap.

For example, `popf` instruction behaves differently in ring 0 and in other privilege levels. In ring 0, nearly all flags could be set using supplied values. But in other privilege levels, flags like IOPL or interrupt will be left unchanged without generating any protection fault. A guest OS may try such instruction to modify IOPL or disable interrupt of its virtual CPU. This action should be trapped into VMM so that it can emulate `popf` on the guest virtual CPU. But it doesn't, which results in an unwanted state.

"Trap and emulate" is a classical way of implementing virtualization on architectures that are virtualizable. An architecture is virtualizable if sensitive instructions are a subset of priviledged instructions. Whenever guest OS attempts to execute sensitive instructions, it will trap (because it is also privileged instruction) into VMM, and VMM will emulate the effect on the guest.

VMWare introduces Binary Translation technology to force those sensitive unprivileged instructions to trap, so that it can emulate [VMw07]. The VMM will translate kernel code to replace nonvirtualizable instructions with new sequences of instructions that have intended effect on the virtual hardware.

For example [KWC07], `popf` may be translated into `int $99`, and the VMM implements a handler to handle to the interrupt.

Question. *(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. HINT: Early versions of User-Mode Linux used a "tracing thread" technique similar to this.*

Generally speaking, there are Type-1 VMM, Type-2 VMM and Hybird VMM [Hei09]. And they use different (but kind of similar) ways to handle guest system calls. In the following, I will try to explain Type-1 VMM syscall handlling. Also, this is a possible implementation. And there are other implmentations that may be slightly different to what I am saying.

> **Type-1 VMM.** The VMM runs on hardware in ring 0. Guest kernels run on VMM in ring 1 (typically). Guest applications run on guest kernels in ring 3.
>
> **Type-2 VMM.** The host OS runs on hardware in ring 0. The VMM runs on host OS in ring 3. Guest kernels run on VMM in virtual ring 0 (and physical ring 3), guest applications run in virtual and physical ring 3.
>
> **Hybird VMM.** The VMM is a hybird of Type-1 and Type-2, and it involves many different implementations.

The following is a brief execution path of a guest syscall [Gel08].

1. Guest application makes a syscall (Ring 3).
2. Traps into VMM (Ring 0). VMM decode the syscall, and transfer to guest kernel (Ring 1).
3. Guest kernel returns from syscall (Ring 1) and again traps into VMM (Ring 0).
4. VMM decodes it and does a real return to guest application (Ring 3).

And here is some notes about it.

- During guest boot, guest kernel will try to register a lot of handlers to the hardware. All these attempts will trap into ring 0, where VMM exists. Thus, VMM has the chance to record all these imformations. When the guest makes a syscall later, VMM knows exactly where the guest syscall handler is.
- When the guest kernel tries to return from syscall, it executes a privileged instruction in an insufficient privilege level, ring 1, which will still trap into ring 0.

The virtual memory address should be protected carefully so that guest user address, guest kernel address, and VMM address (can be regarded as host kernel address) won't trouble each other. Some implementation is avaliable [KDC03]. In general, the idea of [KDC03] is to put VMM address in kernel segment, and put guest kernel and guest user in user segment. But, guest kernel memory pages will be setup to be protected in the page table. Finally, VMM pages could be accessed in ring 0, guest kernel pages in ring 1, and guest user pages in ring 3.

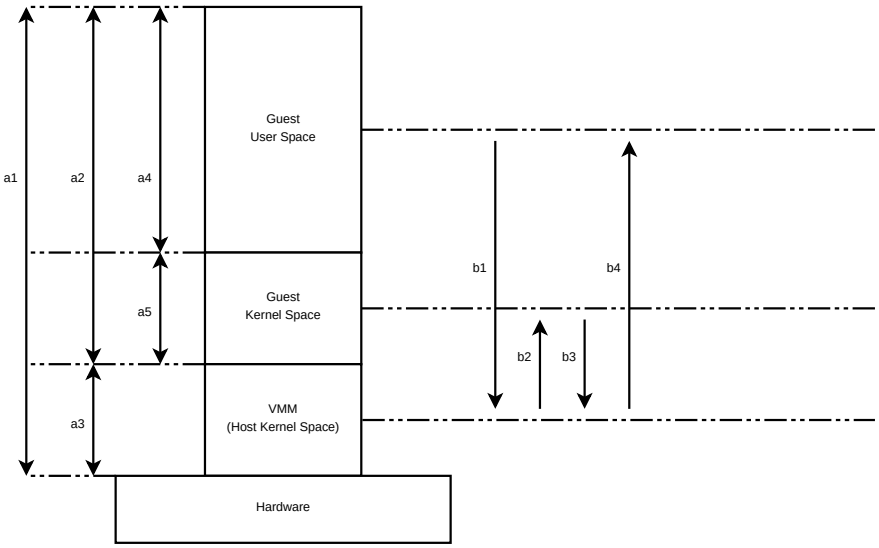

Figure 2. Virtual Address Space and Syscall Flow for Type-1 VMM

In Figure 2, a_1 is the whole virtual address space layout (lower position is higher memory address). a_3 is in kernel segment, a_2 is in user segment. This segmentaion protects VMM memory pages from being attacked because any unintended access will be rejected by CPU. a_5 is protected by marking them supervised in the page table, so that they could only be run in ring 1. a_4 is in traditional user mode.

The syscall flow is denoted with b_1 to b_4. When guest user makes a syscall, it will trap into VMM. (There are several ways of entering ring 0. I take interrupt as an example.) This is done by hardware. The hardware control unit will load a new CS:EIP from GDT specified by an entry in IDT, and a new SS:ESP from TSS specified by TR, and thus change the CPL according to DPL by using some hard-coded rules. Since it is a little bit complex, I made a digram for the whole process of hardware context switch. Please refer to Figure 11 in Appendix A.

When in VMM, it will dectect that it is a syscall by reading the information on the ring 0 kernel stack pushed by hardware control unit. And by using the information it collects during guest kernel boot, VMM can then do a far call or a far jump to transfer control to the guest kernel, and switch to guest kernel stack, guest code segment, guest data segment, and ring 1. This is denoted as b_2.

When finished, guest kernel perform `iret` or `sysexit`. It is not allowed to run in ring 1. So a protection fault generates, and thus trap into VMM again. VMM then restore guest application information from ring 0 stack and switch back to ring 3.

4 Ptrace Interposition

(Programming Assignment.)

5 Process and Thread Scheduling

Question. *Consider the following set of processes, with all times in milliseconds.*

Process	*Arrival Time*	*Burst/Computation Time*	*Priority (1=highest)*	*Deadline*
P_1	*3*	*2*	*1*	*7*
P_2	*0*	*2*	*3*	*5*
P_3	*2*	*3*	*4*	*15*
P_4	*6*	*4*	*2*	*11*
P_5	*1*	*5*	*5*	*20*

Table 1.

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

- *Static Priority with Preemption*
- *Round-Robin, No Priorities, Quantum = 1*
- *Earlist Deadline First, No Preemption*

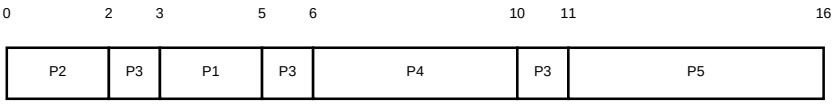

Figure 3. Static Priority with Preemption

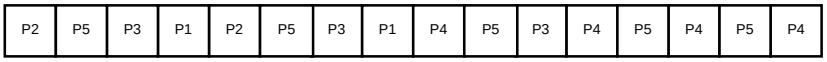

Figure 4. Round-Robin without Priority, Quantum 1

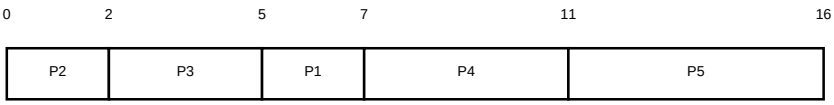

Figure 5. Earilist Deadline First without Preemption

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

Algorithm	Average Waiting Time	Average Turnaronud Time
Static Priority (Preemption)	3.2	4.4
RR (Preemption)	5.4	8.2
EDF (Non-Preemption)	2.6	3.2

Table 2.

Question. *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, ..., t + nT|n = 1, 2, ... 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.*

(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.

Process	*Computation Time C*	*Request Period T*
P_1	*2*	*8*
P_2	*3*	*10*
P_3	*9*	*20*

Table 3.

No. Although $C_1/T_1 + C_2/T_2 + C_3/T_3 = 1 > n\left(2^{1/n} - 1\right) \approx 0.780$, we can not say it is un-schedulable. But when P_1, P_2, P_3 come together, they can not be scheduled since in the first 20 units P_3 misses the deadline. This means they cannot be scheduled [LL73]. The detailed reason is, when they come together, the moment t becomes a critical instant for all the tasks according to [LL73]. Under such situation P_3 cannot be scheduled, then the whole set cannot be scheduled. See Figure 6.

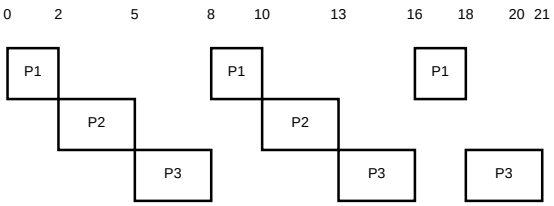

Figure 6. P_3 fails to meet the deadline

However, EDF can handle it since $C_1/T_1 + C_2/T_2 + C_3/T_3 = 1 \leqslant 1$ [Wik].

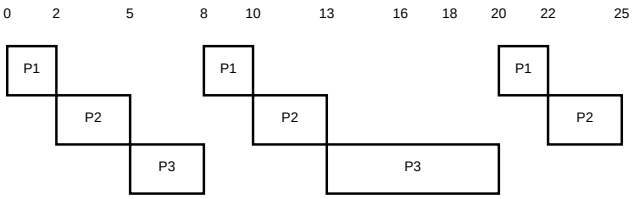

Figure 7. EDF can handle it

Question. *(3 points) Can RMS yield a feasible schedule for the following periodic processes? Briefly explain your answer.*

Process	*Computation Time C*	*Request Period T*
P_1	*3*	*6*
P_2	*3*	*12*
P_3	*6*	*24*

Table 4.

Yes. The reason is similar to the last question, that is they are schedulable in critical instance. See Figure 8 for the critical instance.

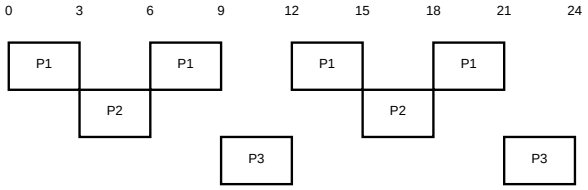

Figure 8. RMS Schedulable

Question. *(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.*

Answer. The maximum utilization $\max(U) = 1$.

Proof. Suppose we have n tasks, and they have the following properties,

$$\begin{aligned} \text{Task}_1 &: C_1, T_1 = k_1 \\ \text{Task}_2 &: C_2, T_2 = k_2 T_1 \\ &\vdots \\ \text{Task}_n &: C_n, T_n = k_n T_{n-1} T_{n-2} \cdots T_1 \end{aligned}$$

From [LL73] we know that if all tasks are schedulable under critical instances, then they can be scheduled at any time. So, I am going to prove that they can be scheduled under critical instances.

Denote U as the utilization. Then,

$$\begin{aligned} U &= \frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \\ &= \frac{\frac{T_1 \cdots T_n}{T_1} C_1 + \cdots + \frac{T_1 \cdots T_n}{T_n} C_n}{T_1 T_2 \cdots T_n} \end{aligned}$$

The critical instance for Task_n means, all tasks with higher priority comes together, and get served before Task_n. In our setting, $k_1, k_2, \cdots, k_n > 1$, thus $\text{priority}(\text{Task}_i) > \text{priority}(\text{Task}_j)$ if $i > j$.

Then, if they can be scheduled under critical instances, the following holds,

$$\begin{aligned} C_1 &\leqslant T_1 \\ \frac{T_2}{T_1}C_1 + C_2 &\leqslant T_2 \\ &\vdots \\ \frac{T_n}{T_1}C_1 + \frac{T_n}{T_2}C_2 + \cdots + \frac{T_n}{T_{n-1}}C_{n-1} + C_n &\leqslant T_n \end{aligned}$$

The last formula can be transformed into

$$\frac{T_1 \cdots T_n}{T_1}C_1 + \cdots + \frac{T_1 \cdots T_n}{T_n}C_n \leqslant T_1 \cdots T_n$$

Thus, if $\frac{T_1 \cdots T_n}{T_1}C_1 + \cdots + \frac{T_1 \cdots T_n}{T_n}C_n \to T_1 \cdots T_n$, then $U \to 1$. So, the problem becomes to prove the linear equation of the following has a root.

$$\begin{aligned} C_1 &= T_1 \\ \frac{T_2}{T_1}C_1 + C_2 &= T_2 \\ &\vdots \\ \frac{T_n}{T_1}C_1 + \frac{T_n}{T_2}C_2 + \cdots + \frac{T_n}{T_{n-1}}C_{n-1} + C_n &= T_n \end{aligned}$$

This is not difficult. By subtract formula $m-1$ from formula m, we get

$$\begin{aligned} C_n &= \frac{T_n}{T_{n-1}}C_{n-1} = \frac{T_n}{T_1}C_1 \\ C_{n-1} &= \frac{T_{n-1}}{T_{n-2}}C_{n-2} = \frac{T_{n-1}}{T_1}C_1 \\ &\vdots \\ C_2 &= \frac{T_2}{T_1}C_1 \end{aligned}$$

Since C_1 is a known value, we can conclude that the linear equation has a root. Thus, $\lim U = 1$. Since U must less than or equal to 1 so that they are schedulable, then $\max(U) = 1$. □

Question. *(18 points) EEVDF (Earliest Eligible Virtual Deadline First) is a proportional share scheduling policy. Given three tasks with weights of 1, 2, and 3, 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.*

(I assume that all requests generated by these tasks are of length 1, and all of them become active at time 0. To make the figure clean and easy to read, I use 0.3 for $\frac{1}{3}$, 0.7 for $\frac{2}{3}$, etc.)

According to [SAo], when the virtual deadlines are the same, scheduler could continue the current task to save context switch overhead (if the current task's virtual deadline are the same with others'), or to pick a task with bigger weight (if the current task's virtual deadline are bigger than others').

A recursion formula provided in [SAo] could be adopted in my solotion. Please refer to the paper for actual meanings of the symbols.

$$\begin{aligned} ve^{(1)} &= 0 \\ vd^{(k)} &= ve^{(k)} + \frac{1}{w_i} \\ ve^{(k+1)} &= vd^{(k)} \end{aligned}$$

The number on each segment denotes the virtual deadline of the task at that particular moment.

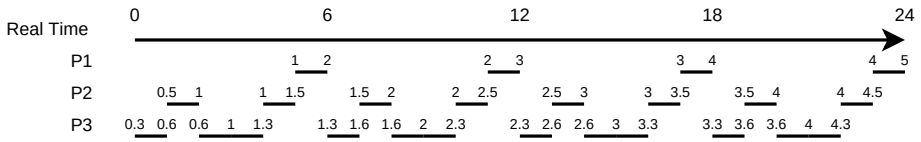

Figure 9. EEVDF on Single Processor

(For dual processor, I assume that there is only one global run queue. When they encounter the same virtual deadline, use the first processor. Whenever possible and legal, continue the same task on the same processor as long as possible.)

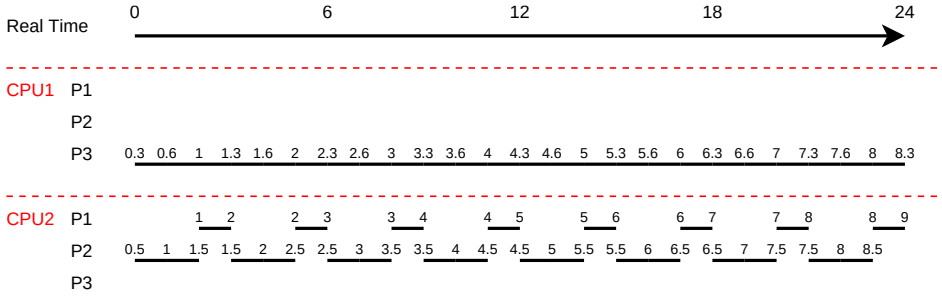

Figure 10. EEVDF on Dual Processor

Appendix A GDT, IDT, TSS, and Hardware Switch

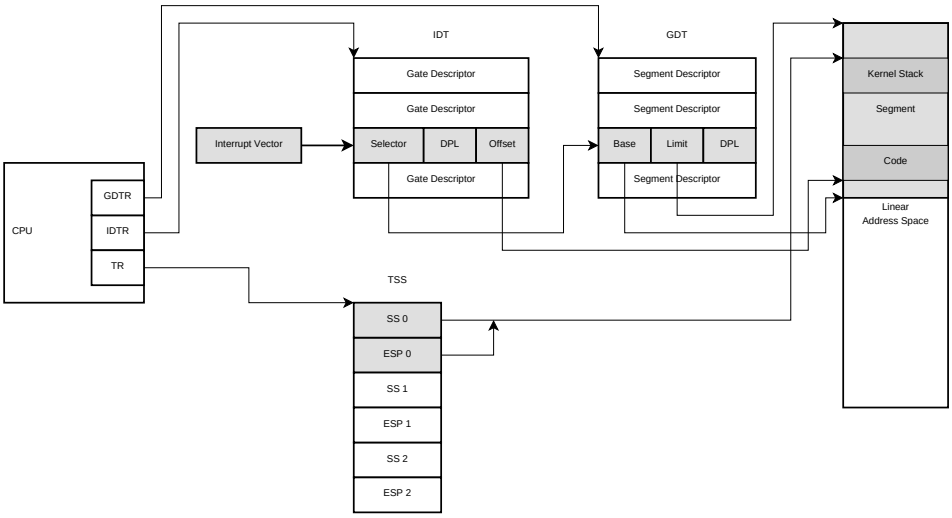

Figure 11. Relations between CPU, IDT, GDT, TSS, and Linear Address Space on Intel x86

Please note that, in Figure 11, IDT, GDT and TSS resist in the memory, and most likely, in the kernel data segment. GDT has one or more entries that describe TSSes, and TR points to the current TSS. GDTR and IDTR point to GDT and IDT repectively.

TSS is a defined hardware-dependent data structure, and contains stack selectors and stack pointers for ring 0, ring 1, and ring 2.

GDT entries are segment descriptors, which contain segment base, segment limit, descriptor privilege level, and more.

IDT entries are gate descriptors, which contain segment selector, segment offset, descriptor privilege level, and more.

CS, DS, SS registers contain current selectors to select descriptors from GDT. There are hidden (unprogrammable) registers contain current selected descriptors corrosponding to CS, DS, and SS.

Interrupt vector is the value used in the `int` instruction. For example, `int 0x80` points to the 128th entry in the IDT.

Another way to switch run level is to do a `far call` or a `far jmp`. The `far` means that segment selectors are encoded in the instruction. When the segment descriptor privilege is different from CPL, hardware control unit may switch CPL and stack, or generate a protection fault.

Bibliography

[BC05] D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Series. O'Reilly Media, Incorporated, 2005.

[Gel08] Johan De Gelas. Hardware virtualization: the nuts and bolts. 2008. Http://www.anandtech.com/show/2480.

[Hei09] Gernot Heiser. Virtualization. 2009. Http://www.ok-labs.com/blog/entry/some-get-it-some-dont/.

[KDC03] Samuel T. King, George W. Dunlap and Peter M. Chen. Operating system support for virtual machines. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '03, pages 6–6. Berkeley, CA, USA, 2003. USENIX Association.

[KWC07] Mike Kasick, Glenn Willen and Mike Cui. Virtualization. 2007. Http://www.cs.cmu.edu/~410-s07/lectures/L38_Virtualization.pdf.

[LCFG+05] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y.T. Shen, K. Elphinstone and G. Heiser. User-level device drivers: achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, 2005.

[LL73] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, jul 1974.

[SAo] I. Stoica, H. Abdel-Wahab et al. Earliest eligible virtual deadline first: a flexible and accurate mechanism for proportional share resource allocation. .

[Sha04] Asim Shankar. *Linux Networking*. 2004. Http://geocitiessites.com/asimshankar/notes/linux-networking-napi.html.

[Var04] G. Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. The Morgan Kaufmann Series in Networking. Elsevier Science, 2004.

[VMw07] VMware. Understanding full virtualization, paravirtualization and hardware assist. 2007.

[Wik] Wikipedia. Earliest deadline first scheduling. Http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling.