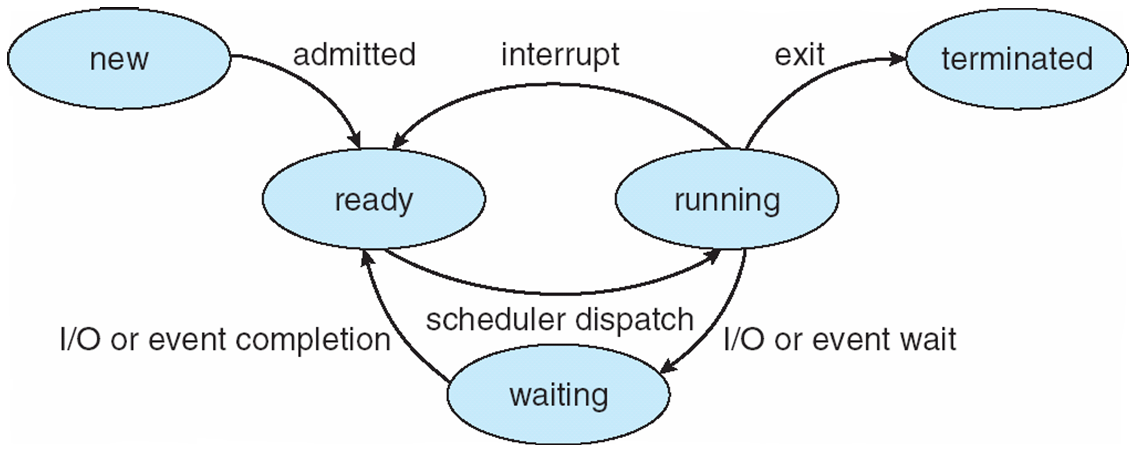
**EECS4287-5287: Principles of Cloud Computing, Fall 2020**

**On-ground Section: Homework #2: Assigned: 10/6/2020; Due 10/20/2020 @11:59:59 electronically into Brightspace.**

**2U Section: Homework #2: Assigned: 10/7/2020; Due Wed 10/21/2020 @11:59:59 electronically into 2U LMS.**

**Absolutely no collaboration is permitted on the homeworks. It is part of the individual assessment. Honor code applies. Please consult the OS refresher lecture slides.**

1. **Processes:** A typical operating system process lifecycle state diagram is shown in the figure below (taken from Silberschatz’s OS textbook slides). Now consider a scenario where you have opened a shell window, and also double clicked the VirtualBox icon on your desktop so that the VirtualBox program starts executing. At this moment, both the shell and VirtualBox are started from the root OS process. Now suppose that we invoke the “vagrant up” command in the shell to create and start an Ubuntu VM. At this stage (i.e., before vagrant up command exits but the Ubuntu VM is now up), how many total processes are running in the system? Draw a composite process lifecycle state diagrams for all the processes you have identified except the shell process and the OS root process (we don’t show it for these to keep the clutter low). Instead show these two as just a single opaque box each to avoid clutter. Since one process results in the creation or triggering (i.e., unblocking) of another process, show a dotted arrow originating from the appropriate state of the source (e.g., parent) process and ending in the appropriate state of the destination (e.g., child) process. Use similar dotted arrows emerging from the boxes for the shell and OS root process. To make it easy for you to do this digitally, **you can just make copies of the diagram below** to represent the process under consideration, name it (like how vagrant is shown) and draw the dotted arrows among them (but make sure that these dotted arrows start from the correct state of the source to the correct state of the destination process). As an example, the shell that creates the child process “vagrant” is shown below. **(10 points)**

vagrant

shell

**<ADD SPACE TO WRITE YOUR ANSWER>**

1. **Scheduling:** In this problem we are asked to ***determine the total response time and waiting time for the Ansible process*** shown in the table below that is executing inside a guest OS. The response time and waiting times **are to be computed based on the absolute time that is tracked by the host OS and not that of the guest OS**. The clock speeds of both the host and guest are assumed to be the same. However, we are going to assume that virtualization is emulated in software (i.e., no help from hardware). Consequently, although the guest OS schedules its processes according to its scheduling algorithm, ultimately that VM is subject to the scheduling policies of the host OS. As a result, the guest OS is able to track only its local progression of time which does not align with the absolute time tracked by the host OS because the guest OS can track its time only when its VM is scheduled on the host CPU. In our example, the host OS uses a round robin scheduling algorithm which allocates 2 units of CPU quantum per process after which the process must be preempted if it has additional burst left and is scheduled again when its turn comes next. We do not allow any empty slots, i.e., if a process has completed, its quanta do not go idle. So, when there is just one process left in the system, it can be scheduled in consecutive quanta. Our guest OS schedules processes according to a pre-emptive priority scheduling in which if a low priority job is currently executing, and a higher priority job enters the system, the lower priority job will be pre-empted and the higher priority job takes over until all its burst is done (unless an even higher priority job shows up). In case of a tie, use first-come-first-serve for processes of the same priority. In the data provided below, HP => host process, and GP => guest process. Time t = 0 is the start time when both the host OS and guest OS’s clocks are synchronized (later they will drift apart). Draw a Gantt chart for how processes are scheduled on the host and guest OS. For every turn of HP2 being scheduled on the host OS, show the nested level of scheduling inside HP2 where the guest is scheduling processes on its end. Arrival time of 0 for host processes => they all show at the same time and immediately get scheduled per round-robin with HP1 followed by HP2 followed by HP3. **(10 points)**

Table : Data for the Host OS (processes on the host are named with prefix H)

|  |  |  |
| --- | --- | --- |
| **Process** | **CPU Burst** | **Arrival Time** |
| HP1 | 10 | 0 |
| HP2 (our VM) | 19 | 0 |
| HP3 | 8 | 0 |

Table : Data for the Guest OS running in the VM process HP2

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **CPU Burst** | **Arrival Time (relative to guest clock; not absolute)** | **Priority** |
| GP1 (Ansible) | 10 | 0 | Lowest |
| GP2 | 3 | 4 | Highest |
| GP3 | 5 | 2 | Intermediate |

**<ADD SPACE TO WRITE YOUR ANSWER>**

1. **Virtual memory and demand paging:** A virtual machine, as we have seen, is simply a process on the host OS and hence will incur the same demand paging operations as that of other applications on that host. However, the VM has a guest OS, which has its own version of virtual memory and demand paging. As was the case in question 2, assume that virtualization is emulated completely in software. Moreover, assume that the guest OS has a much smaller addressable memory range than the host OS and that the block size of the page on the guest is ½ that of the host. In other words, two pages of the guest can fit in one page of the host. Both the host and guest OS divide their main memory into four frames each for the purpose of demand paging. Since we are emulating virtualization in software, the emulation logic tries to improve performance for the guest by considering the guest OS “swap space” (i.e., the part of disk on which dirty pages get written to or pages not in memory are obtained from) are memory-mapped. In other words, although the guest OS may think that it has experienced a page fault, the guest swap space is in memory. It is entirely possible, however, that when mapped to a sequence of page references on the host, actual disk I/O may happen due to the real demand paging on host OS. As in Problem #2, assume that HP2 is our VM while HP1 and HP3 are other applications running on the host OS, and that these processes are scheduled in a round robin manner. Thus, the string of requests for pages on the host OS will be an interleaving of pages referenced by the three processes.

|  |  |
| --- | --- |
| **Guest OS page** | **Host OS page** |
| 1 | 5 |
| 2 | 5 |
| 3 | 2 |
| 4 | 2 |
| 5 | 1 |
| 6 | 1 |
| 7 | 8 |

**Part (a)** You are provided with the following reference string for page references occurring on the guest OS. How many page faults occur in the guest OS if it uses the Least Recently Used (LRU) algorithm. Show your work.

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 5

**Part (b)** For each pair of references on the guest (because 2 pages of guest map to 1 page of host), the virtualization logic uses the above table to map into a corresponding single or double page reference on the host OS. Fill in the blanks with host-level page references obtained by mapping guest page references, and subsequently use LRU algorithm to compute the total number of page faults on the host for the scenario provided. A single blank does not mean only one page ref (there could be two depending on the mapping). Consecutive dashes implies other processes on the host had terminated by that time and only our VM process was alive. **(20 points)**

3, 4, \_, 6, 3, \_, 9, 6, \_, 7, 4, \_,9, 3, \_,4, 6, \_, 7, 9, \_, 3, 6, \_, \_, \_

**<ADD SPACE TO WRITE YOUR ANSWER>**

1. **Effective Access times:** In demand paging, pages must often be brought into memory from disk. Moreover, if the selected victim that is being evicted is “dirty” (i.e., modified), then it must be written back to the disk bit not otherwise. Using the final string of page references that you find out in problem #3 above and using the LRU algorithm, find the effective access time for all these operations to execute assuming that 40% of the page faults comprise victims that are “dirty”. You are given that memory access time is 10 milliseconds while disk access (i.e., both read and write each) time is 100 milliseconds individually for read and write. Ignore any overhead of interrupt handling (i.e., page fault). Assume there is no cache and hence pages are not cached in the translation lookaside buffer. **(10 points)**

**<ADD SPACE TO WRITE YOUR ANSWER>**