

COIS 3320H Winter 2021

Assignment 1

## 1- Based on Exercise 1.1.1 – Abstraction and Virtualization (3 points)

Justify or refute the statements below. Give supporting examples, if appropriate.

1. **Abstraction can be used without virtualization.**

Yes, Abstraction can be used without visualization. Even though abstraction and visualization are closely related, that does not mean that the two cannot work without the other. Abstraction is the act of removing unimportant details or attributes of objects in order to construct more general and less complex objects. However, this does not mean that any virtual object is created, that means not necessarily, visualization occurred. To give an example, when we are opening a file on device to read, the content is opened and read. We don’t see the entire operation of how the file was accessed and how the contents were read from it, there was no new virtual object created in the reading process.

1. **Virtualization can be used without abstraction.**

Yes, virtualization can be used without abstraction. Virtualization is the process of creating an illusion of having one or more objects with more desirable characteristics than the real object. However, this illusion can be similar in attributes to a physical object. That means, there was no removing of attributes done. Thus, a virtual object got created for an existing physical object. Hence, visualization done without abstraction. For instance, when trying to find a movie in ‘similar’ criteria, let’s say, *‘find movies similar to crawl’ (suggestions are appreciated).* All the movies are searched using the same genre that were in crawl. Even if a new movie has an extra genre, the movie is still shown because the operation was done, and the search was represented without any change in attribute. So, no characteristic was removed or changed. I am a little doubtful on this example, so for another example, let’s suppose I have a Bluetooth speaker, now the OS can create this illusion for various users that there are more than one speaker for each user but actually all these virtual speakers are similar to the one physical speaker. Here, we have visualization but no abstraction.

1. **Abstraction and virtualization can be used together.**

Yes, abstraction and visualization can be used together. The operation would be basically representing some virtual object but after removing some characteristics or attributes. For example, clicking on the *evergreen* “save” button (the hard disk, even after we are in 2021, anyways), the visual representation (visualization) is the virtual representation of the hard disk, and the abstraction is hiding the fact and process of how exactly the operation is taking place, not all users want to know how it got saved in the memory. So the user is oblivious to the interior execution.

## 2- Based on Exercise 1.1.2 – Multiprogramming & Time-Sharing (3 points)

**What do multiprogramming and time-sharing have in common? What are the differences between the two concepts?**

Multiprogramming, as the name suggests is a technique that where various programs are active in the memory at the same time and switches execution among the different programs to maximize the use of CPU and other resources. Time-sharing by definition is an extension of multitasking/ multiprogramming, where the CPU switches periodically between the active programs to guarantee the acceptable response times to each other.

Multiprogramming and timesharing have the common tasks of letting multiple programs stay active in memory at the same time and using a lot of CPU and resources. They both help in saving time and completing multi-programs faster than sequential execution.

The difference between the two are that multiprogramming let’s the other programs run and use CPU when the previous program enters the I/o-bound phase, where a little CPU is needed. However, in time-sharing, the programs are active together and utilizing the CPU at the same time periodically. A little of program 1 uses the CPU, then a little of program 2 uses the CPU. This alternate process is done on acceptable response times to each user. This means both the programs enter the I/O-bound phase at approximately the same time. Basically division into 50-50 processing in time-sharing.

## 3- Based on Exercise 1.2.1 – Interrupts & Traps (3 points)

**What do interrupts and traps have in common? What are the differences between the two concepts?**

Interrupt is an event that diverts the current execution of a program to a predefined location in the kernel in order to respond to the event. A trap is also an interrupt, triggered by currently executing instruction.

The common things between interrupt and trap is that they both stop or halt the current execution and then the focus and execution of some other event is handled and completed in the specified location in the kernel.

The difference between the two is that the request for interruption is caused by an event which is external to the current execution instruction. A trap is also an interrupt however the event that is doing the interruption is the current execution instruction.

## 4- Based on Exercise 1.2.2 Multiprogramming, Time-Sharing, and Interrupts. (3 points)

1. **Is multiprogramming possible without interrupts?**

At first thought, it may seem that since multiprogramming involves two executions running, there might be interrupts involved. However, multiprogramming does not need interrupts to work. This is because, the change in the executions, that is the context switch happens when the current execution event enters the I/O-phase. This means there was no interrupts.

1. **Is time-sharing possible without interrupts?**

Time-sharing is different than multiprogramming. There is a periodic context switching happening, which means, that there is a periodic interruption taking place. Hence, time-sharing does require interrupts periodically and hence cannot be possible without interruption.

## 5- Based on Exercise 2.1.1 – State Transitions (4 points)

Indicate whether each series of state transitions is valid or invalid. If you think a series of state transitions is invalid, please justify why.

1. suspended ➛ blocked ➛ suspended ➛ blocked ➛ ready ➛ running : ***Valid***
2. new ➛ ready ➛ suspended ➛ blocked ➛ suspended : ***Valid***

if ready to suspended then go back to ready and not blocked

1. new ➛ ready ➛ running ➛ ready ➛ new : ***Invalid***

A transaction cannot go from Ready state to new state. New state is just when there is a newly created process. Ready state means that the new process is ready to compete for the CPU, and once that has started the process does not go back to new state, only to running or suspended.

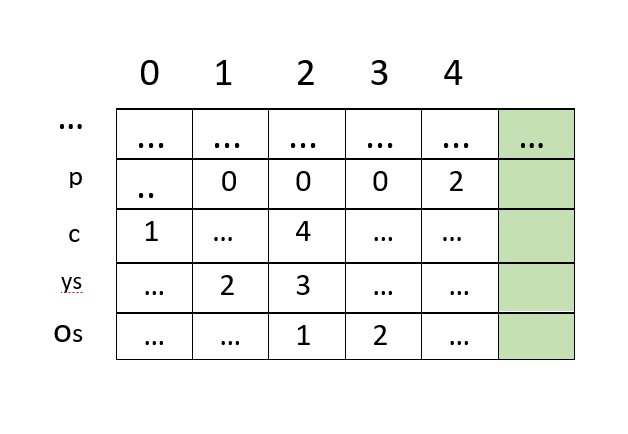
1. running ➛ blocked ➛ ready ➛ blocked ➛ suspended : ***Invalid***

Transition from Ready to Blocked state is not possible. A process in the ready state goes to running where it can compete for CPU. And then only from there can they request for resources and enter the blocked state. Once that is released, it goes to ready state. But, from ready state, there is no direct way to request for resources from CPU and hence it cannot go to blocked state.

## 6- Based on Exercise 2.3.1 – Creation Hierarchy Without Linked Lists (4 points)

Processes 0-4 are related as follows: 1, 2, 3 are children of 0, and 4 is a child of 2. PCBs are implemented as an array indexed by the process number. Each PCB has the links: parent (p), first child (c), younger sibling (ys), and older sibling (os).

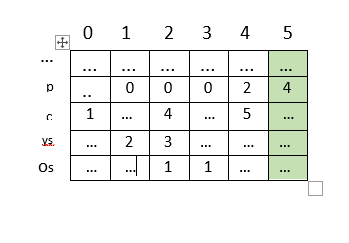
1. **Complete the PCB array to show the values of the 4 links (p, c, ys, os) for all processes, to reflect the parent-child hierarchy**



… -> means that there is no link.

1. **Modify the array to reflect the creation of a new child,** 5, of process 4.

This means -> 5 is the child of 4



## 7- Based on Exercise 2.3.2 – Ready List without Linked Lists (5)

**The RL can be implemented without dynamically managed linked lists by creating a new field, next, in each PCB, which points to the next PCB on the same list. Each entry of the RL then points to the first PCB on the list.**

**Assume that RL contains 3 processes at level 5 and 1 process at level 0. Draw a diagram showing the RL and the modified PCBs.**

**Running**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

n

5

0

Pointers to PCBs

Assuming that level 5 is not the highest priority, and there is a higher priority at level n, otherwise simply level 5 will be at the very top. The RL contains processes at level with priority 0 to n. The process with highest priority at n is running. There are 3 process at level 5 and 1 process at level 0. If level 5 was indeed the highest priority then the first process will run first and continue with the other processes. If any new process with lower priority than the running process priority is added, the process running continues running. Else, the process running turns to ready state and then new process starts running.

## 8- Based on Exercise 2.4.1 – The Suspend & Activate Functions (4)

The following pseudo code implements the suspend() and activate() functions. Two new states are introduced, suspended\_ready and suspended\_blocked, to keep track of the state in which a process was suspended. That is, a ready process moves to the suspended\_ready state by the suspend function. Similarly, a blocked process moves to the suspended\_blocked state by the suspend function. The activate function reverses the transitions.



suspend(p) {

if (p.process\_state == blocked)

p.process\_state = suspended\_blocked

else p.process\_state = suspended\_ready

}

activate(p) {

if (p.process\_state == suspended\_ready)

p.process\_state = ready

else p.process\_state = blocked

scheduler()

}

1. **What changes must be made to the scheduler or other functions to make suspend/activate work correctly?**

* To get suspend and activate functions to work correctly, there might be certain conditions we need to be careful about. The scheduler function must know the definition of the new process state variables. So that it knows that whether the processes are to be moved from suspended to ready or blocked. This will also help the scheduler in knowing how to prioritize them. Also, let’s say a process goes from suspended to blocked, then other functions (maybe for blocked functions), it will be good to know that this was from suspended and the scheduler must prioritize and give it the resources before the process that is coming from running.
* It should also be ensured that the processes being suspended should be linked with the list they were in prior to suspension. Even though the process states are being names accordingly, does not mean that either the ready or waiting list should not have link with these processes. Whether via any pointers or state field or whatever, there should be an established link.
* Maybe, take care of the children of that process and if a suspended\_ready process got activated then make its children also activated. For blocked however, this might not be true, since a process would have already blocked its children before being transitioned as suspended.

1. **Why is the scheduler called only in activate but not in suspend?**

Schedular is not called in suspend but only in activate. This is because in suspend, the processes that have been suspended can not work and are out of the schedular lists. In other words, these process can be ‘ignored’ while other process run as scheduled. For instance, let’s say there is a process A and next scheduled is process B. Now when process A gets *suspended*, process B, even though it was *scheduled* by the OS after process A, it can still run. However, if process A was blocked then process B can only run when process A completes *running*. That’s why, in suspended, there is no need to call *scheduler()* since there isn’t any changes except ignoring the process with *process\_state* value *suspended*. In activate though, the process’s *process\_state* is changed back to either ready or running and hence, *scheduler()* needs to be called since these process are now on the scheduler list.

1. **A process must be prevented from calling suspend() or activate() on itself. Why?**

If a process can call *suspend()* or *activate()* functions on itself, then there might be kind of a havoc for OS. A process might suspend itself and then the scheduler needs to decide which process to run next. This can prove to be difficult if the scheduler does not know about the suspension. There may be a delay and slow in the changing of processes. Also, if processes can suspend themselves then too many processes can be suspended, and the Scheduler will not be able to prioritize or give resources to processes correctly. If any process can activate itself on its own, then scheduler might face trouble prioritizing and there can be mess up in the ready list/ prioritizing. This way the process recently entering ready state might run before the process whose priority should have been higher.

1. **A process must be prevented from calling suspend() on an already suspended process or calling activate() on a currently active (ready) process. Why?**

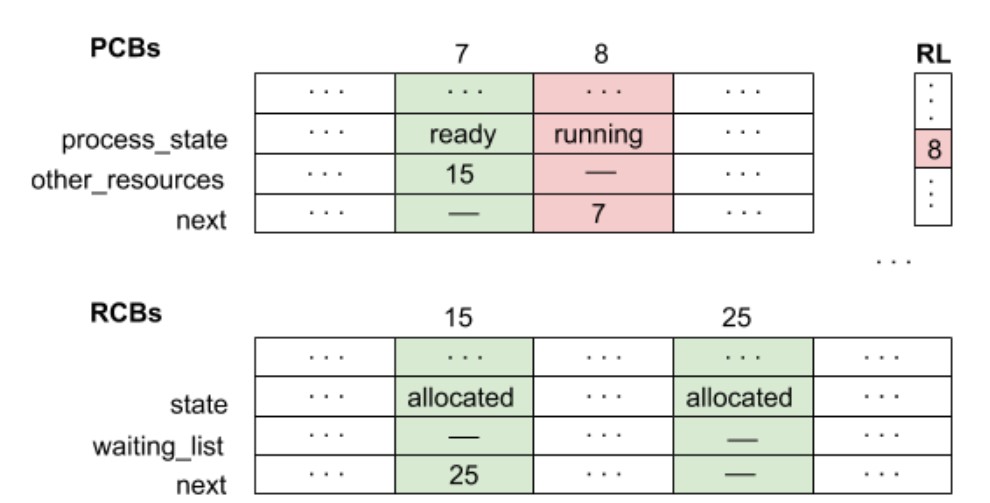
I will give this answer based on the code provided. According to the function, there is an if-else statement used. If the process on whom suspend() is called, is blocked then the process state’s value is overridden as *suspended\_blocked,* and for any other case it is *suspended\_ready* (this includes any running process on whom suspend() is called. Now, if we call suspend on already suspended function, the process state value is going to be either “*suspended\_blocked”* or “*suspended\_ready*” , that means in the if-else, the if condition is not meant since process state is not blocked, and hence the new process state value will become *suspended\_ready.* This means, that a *suspended\_blocked* will now be *suspended\_ready,* and instead of being sent back to blocked on activate() it will be sent to ready where it might still not get the resources it went to blocked for.

Similarly, for activate function, any process that has already been activated will have process state as either ready or blocked, and hence even a ready process can have process state blocked. And since scheduler is called after that, the priorities will change leading to an already higher priority process to run late.

## 9- Based on Exercise 2.5.1 – Request & Release Functions in Action (6 points)

Two processes (7 and 8) are currently on RL. Process 8 is at the head of the list and is running. The next field of process 8 points to process 7, which is ready.

Process 7 is currently holding two resources (15 and 25). The other\_resources field of process 7 points to resource 15, which in turn points to the next resource, 25.



1. **Show all changes to the data structures after process 8 requests resource 15.**

Resource 15 is currently held by process 7. Process 7 is ready.

Process 8 requests resource 15.

**Process 8** changes to **blocked**.

Process 7 changes to **running**.

Process 8’s resources list **is empty**.

Process 7’s resource list **contains** **other\_resource** **15 then next 25.**

Resource 15 is **allocated.**

Resource 15’s waiting list **contains process 8**

Resource 25 is **allocated.**

Resource 25’s waiting list **contains process 7.**

RL contains the **process 7**.

These are all the changes that happen when process 8 requests resource 15.

1. **Show all changes to the data structures after process 7 releases resource 15.**

**Assume that processes are entered into RL at the end of the list.**

Process 7 releases resource 15.

**Process 8** changes to **ready.**

Process 7 is **running**

Process 7’s resource list **contains 25.**

Process 8’s resource list **contains resource 15.**

Resource 15 is **allocated.**

Resource 15’s waiting list **is empty.**

Resource 25 is **allocated.**

Resource 25’s waiting list **is empty.**

RL contains the processes  **7, 8**

These are the changes happening once process 7 releases resource 15.