SHORT-TERM PROCESS SCEHDULER

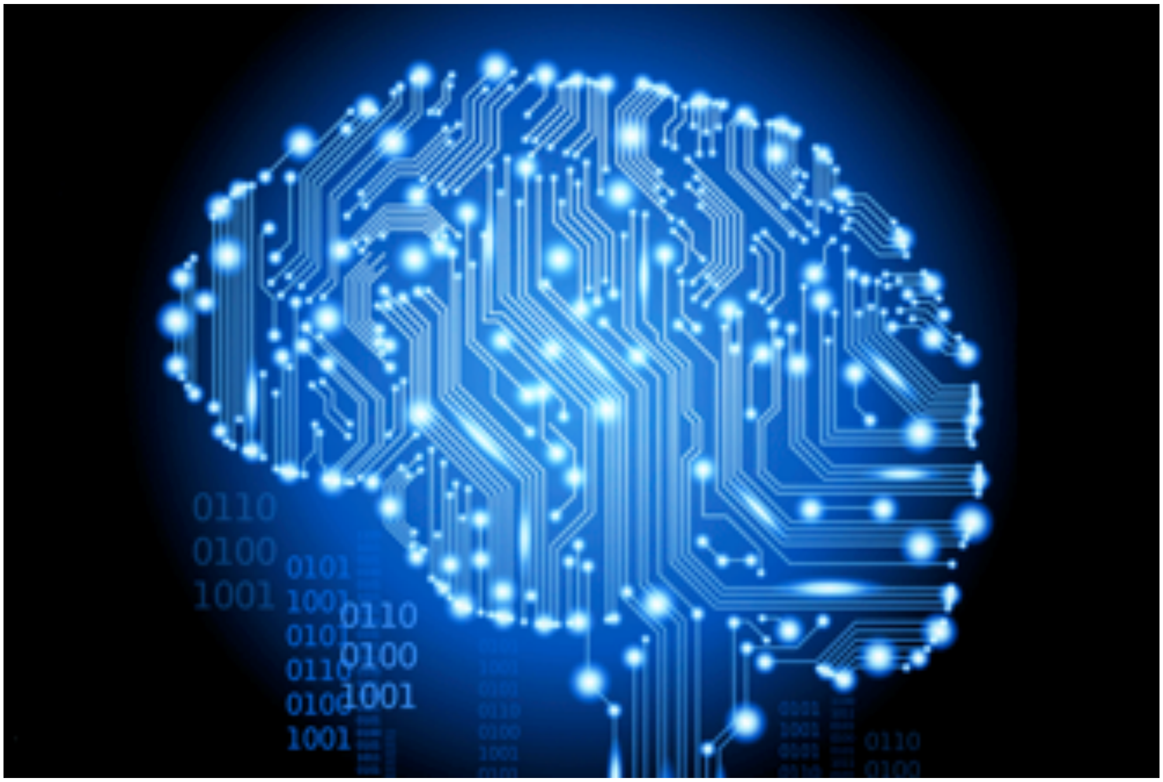


TABLE OF CONTENTS

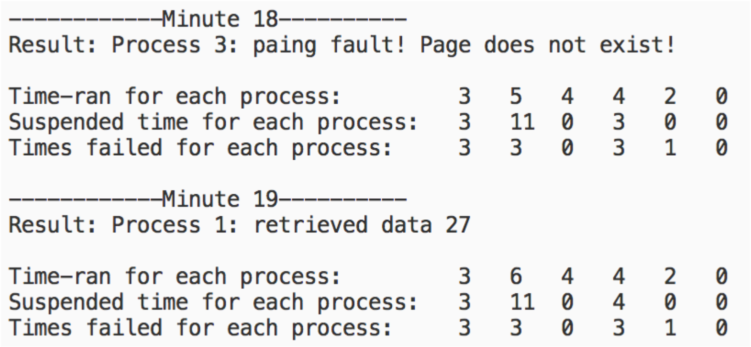
DESCRIPTION

This project extends the knowledge learned thus far, from managing the memory requirements of the processes, to scheduling processes for execution. To do so, this project will use a simulated process scheduler in order to demonstrate the Shortest Process Next scheduling algorithm.

DESCRIPTION OF METHODOLOGY/ APPROACH

The approach to this topic was a simple but effective implementation of a short-term process scheduler. The program starts out by creating an “OS”. The OS object basically keeps track of each of the process, their remaining time, and which process should run next. Also contained in the OS are a Disk object and a Memory object.

The scheduler is based on a shortest process next or SPN algorithm, where each of the processes have a priority or nice value of the initial estimated time it takes to complete the process. There is no p eemption, so a process will run to completion until it either finishes, or a fault occurs. At which point a fault occurs, the process will, instead of being terminated, have a context switch and be “killed” for one cycle. After that cycle, the process will be added back to the queue according to their original nice value. While the process is killed, if there is another process ready, the other process will begin to execute.



```
------------Minute 18----------
Result: Process 3: paing fault! Page does not exist!

Time-ran for each process:         3   5   4   4   2   0
Suspended time for each process:   3   11  0   3   0   0
Times failed for each process:     3   3   0   3   1   0

------------Minute 19----------
Result: Process 1: retrieved data 27

Time-ran for each process:         3   6   4   4   2   0
Suspended time for each process:   3   11  0   4   0   0
Times failed for each process:     3   3   0   3   1   0
```

Additional functionality of the OS includes the creation of a process, the retrieval of data for a specific process, and data handling.

Once the OS is initiated, processes being to be created according to predetermined hard-coded arrival times. We chose an initial six processes that will at some point be added, but the implementation does scale to higher amounts of processes. Following the creation of each process, we continue by repeatedly having each process retrieve data from a randomly generated address in memory. If when a new process is added, the OS determines its priority, or nice value, based on the time it takes to complete the process. The process scheduler uses a shortest process next algorithm which looks at the overall time it takes to finish the process.

Process objects contain a data size and a segmentation object which contains page tables.

ASSUMPTIONS

The assumptions we made include that addresses are decimal instead of binary, and the page replacement algorithm is random.

Example: address 25264 is broken down to [2|52|64] where [segment = 2|page = 52|offset=64]

Another assumption we made is that in this simulation the disk size in unlimited and the virtual memory contains 100 page frames. All pointers used are shared pointers which allows for automatic destruction of referenced object and pointer.

The page size is set to 100b which is the reason the offset is 2 decimal digits to represent 100 addressable bits.

Process arrival times are hard coded as well as the number of processes generated. Arrival times are the clock cycle at which a given process will join the execution queue as ready to execute

Process times for each process is already known and are hard coded in. Process times is how many clock cycle the process needs to run for before completion.

IMPLEMENTATION

PROJECT 2

SIZE OF PHYSICAL AND VIRTUAL MEMORY

As stated above, this simulation assumes that the disk size is unlimited for simplicity sake.

The virtual memory can contain 100 page frames or 10kb

PROCESS MEMORY

The amount of memory given to each process is hardcoded and are chosen so that they are not too big as to where there would be too many page tables, and not too small as to all fit in one page table. This correlates with the page size. Therefore the amount of memory is a 5-digit number.

PAGE SIZE

The page size is set to 100b which is chosen for simplicity and to allow for the offset to represent each byte. This also allows for the virtual memory to be easily understood.

PAGES PER PROCESS

The number of pages per process are calculated by the size of the process / page size.

PAGE TABLE IMPLEMENTATION

Each process keeps track of its page tables within its segmentation object as described earlier. The page table object contains a vector of shared pointers to pageTableEntries. When a process is initialized, each page table is created and added to the process segmentation. When a page table is created, pages are created, pushed to disk and referenced based on page table size and page size. The page size is 100b and the page table size is 100 pages or 10kb.

When a process attempts to retrieve data within a page table, first it will parse the address into segmentation number (first digit), the page number (next two digits), and the offset (last two digits). From there, the corresponding segment is retrieved, which leads to the page table. If the desired page table is not in the segmentation object, it will be grabbed from disk and replace a random page table.

MAIN MEMORY IMPLEMENTATION

Main memory contains a vector of pages and a counter for the availability of space. When a page is added to main memory, it will add to the page vector if it is not full. If main memory is full, we have implemented a random memory allocation algorithm to place the page randomly.

PAGE SWAP ALGORITHM

If a page is not in main memory, the location of the page in the disk is found by summing the disk offset by calculating the offset made by the previously loaded pages. Then the index location of the page is found be moving to the correct segmentation, adding the page number and then the offset previously calculated. The page is retrieved at that location and then added to memory.

PAGE AND SEGMENTATION FAULT

The algorithm implemented to check for page faults checks if the page number from the address is greater than the corresponding page table size. The segmentation fault algorithm checks if the segmentation number is greater than process segmentation entries size.

FREEING RESOURCES

Resources are assumed to be freed at program termination for simplicity of implementation.

PROJECT 3

IMPLEMENTING THE QUEUE OF PROCESSES

The implementation that we chose to use consist of a vector to contain the all of the processes. There are also five other arrays, each tracking the state of each process, the number of cycles a process has run, the priorities or nice values of each process, the total time a process has been suspended, and how many times each process has failed due to a segmentation or page fault.

**The suspended counter for each process only starts if the process had started execution but a segmentation fault or page fault occurred. At which point the process is put back into the queue and has to wait until its turn again.

SHARES OF THE CPU

Shortest process next algorithm helps reduce bias in favor of CPU time for longer processes but is susceptible to starving longer processes.

MAXIMIZING EFFICIENCY

The implementation maximizes efficiency by allowing other processes to run when a page fault occurs, so that there is no down time if there are ready processes in the queue.

USER INPUT REACTION

The implementation does not implement user input for simplicity, but after input, the process asking for input could be allowed to continue execution until either fault or completion.

ENSURING REPEATABLE TIMES

Generally, this will be maintained in the algorithm but is susceptible to change based on load of processes, and the times of those processes. For example, shorter processes will be added to the top of the queue whereas longer process will be added to the end of the queue. So short processes will generally be executed quickly each time and longer processes will take longer. But this could change if there are a bunch of small processes or only large processes ready to execute.

MINIMIZING OVERHEAD

Shortest process next and the implementation reduces overhead by not recalculating a nice/priority value every cycle. It also does not have to constantly switch between different processes.

EFFICIENT RESOURCE UTILIZATION

The implementation allows for efficient resource utilization by reducing idle time of the CPU. When a process faults out, it is “switched” off for one cycle. Instead of the CPU being idle in that time, if another process is ready, it will begin to execute.

STARVATION

Shortest process next reduces starvation for shorter processes because of their higher priority but it is susceptible to starvation of longer processes if there is an abundance of shorter processes. The implementation does not have this problem too much because there is a finite number of processes, basically guaranteeing execution of longer processes.

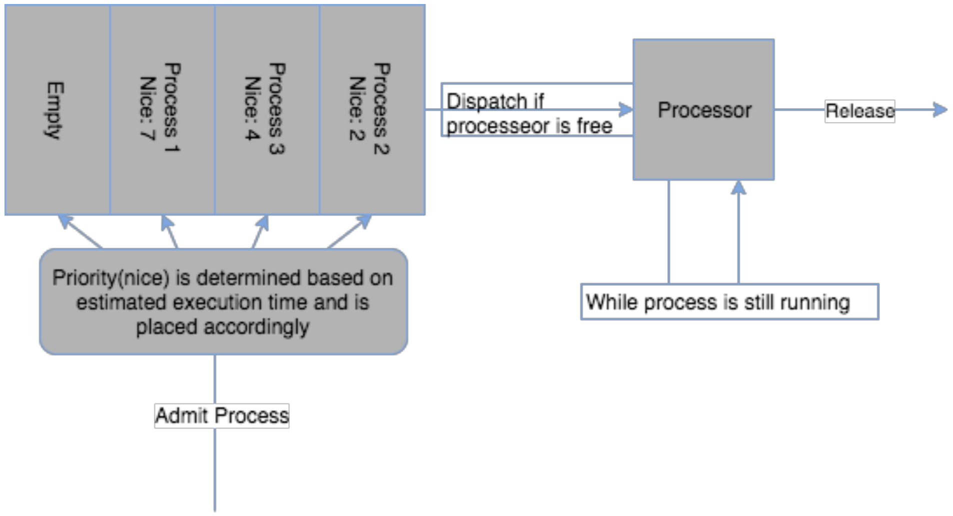
GRADUAL DEGREDATION

Ensuring performance degradation can be difficult but the shortest process next utilizes a continually updated process time estimation in order to improve performance of each process and be able to determine more accurately how long it will take.

COOPERATIVE SCHEDULER

Shortest process next is cooperative in the sense that even if a process is added to the queue that has a higher priority than that of the current running process, it will still wait until the running process either finishes or faults out before execution.

QUEUING DIAGRAM


Empty
Process 1
Nice: 7
Process 3
Nice: 4
Process 2
Nice: 2
Dispatch if processeor is free
Processor
Release
Priority(nice) is determined based on estimated execution time and is placed accordingly
While process is still running
Admit Process

DESCRIPTION/FLOWCHART OF PROCESS SCHEDULING ALGORITHM

DESCRIPTION

The Shortest Process Next or SPN algorithm is implemented in such as way as to avoid giving bias to longer processes. SPN does this by ordering the next process queue by the shortest initial work time of each process. The biggest difficulty when implementing this model into a real operating system, is the need to know the time each process takes before it starts executing. The easiest way to do this is to take an average of the times it took to execute previous similar processes. This can be calculated in the following manner:

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i$$

$$T_i = processor\ exectuion\ time\ for\ the\ ith\ instance\ of\ the\ process$$

$$S_i = predicted\ exectuion\ time\ for\ the\ ith\ instance\ of\ the\ process$$

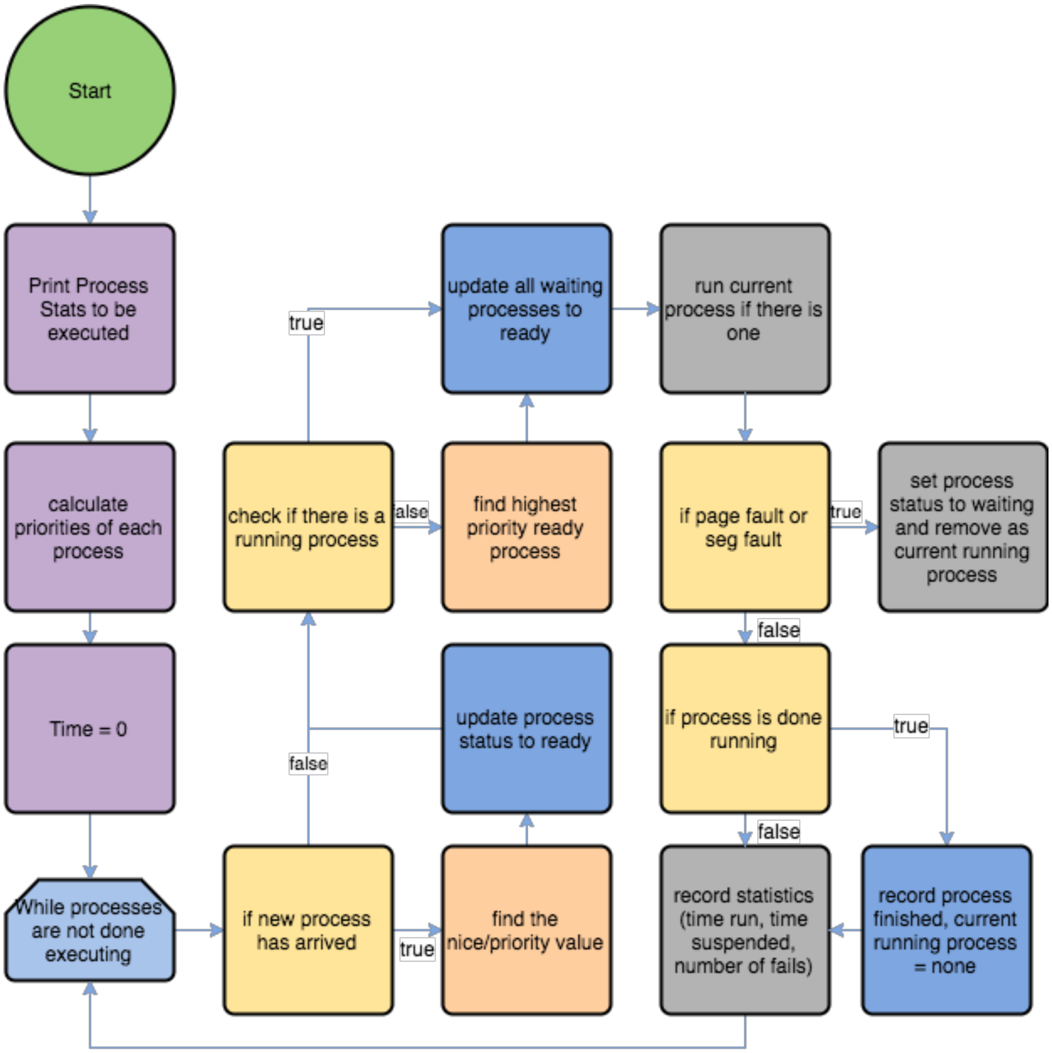
$$S_1 = predicted\ value\ for\ the\ 1st\ instance, not\ calculated$$

There are other ways to predict the value of process execution time such as weighting more recent times more heavily as they may represent more accurate data.

Shortest process next is non-preemptive because it allows processes to finish before the next process starts from the queue. The queue is ordered based on the calculated times for each process. The implementation allows for some preemption in the sense that if a process has a segmentation fault or page fault, instead of quitting the process, another process is allowed to run and the original process faults out for one iteration and then is placed back on the queue.

With SPN, there is a risk of starvation when it comes to longer processes. The implementation isn't effected too much by this since there is a short finite amount of processes, but a real processor might have issues since there is also a lack of preemption.

FLOWCHART


Start
Print Process Stats to be executed
calculate priorities of each process
Time = 0
While processes are not done executing
if new process has arrived
true
false
find the nice/priority value
update process status to ready
check if there is a running process
true
false
find highest priority ready process
update all waiting processes to ready
run current process if there is one
if page fault or seg fault
true
set process status to waiting and remove as current running process
false
if process is done running
true
record process finished, current running process = none
false
record statistics (time run, time suspended, number of fails)

SEGMENTATION AND PAGING FAULTS

ALGORITHMS

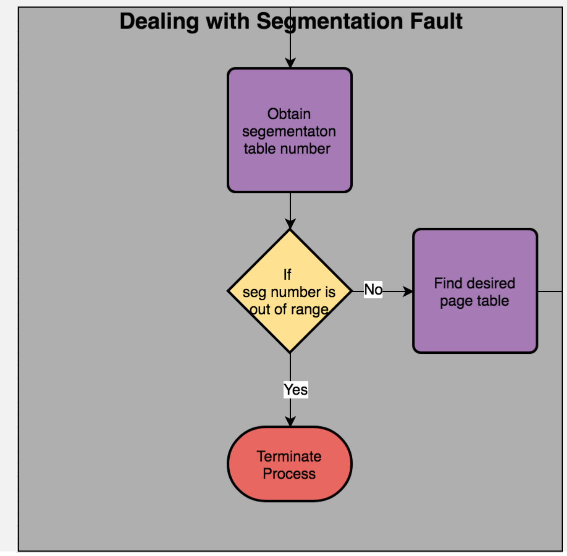
Segmentation faults can be detected by checking the segmentation address to see if it is attempting to access an illegal or invalid address. This can happen when using an invalid pointer or reference outside of an allocated range.

Major page faults happen when the page referenced is not in the TLB, or main memory. This just means the OS has to have the CPU retrieve the page from disk and add it to main memory. This was described above. A minor page fault happens when the page is loaded into memory but not allocated to that process. Main memory can then just make a copy of the page and load it into a table for the requesting process instead of going all the way to disk.

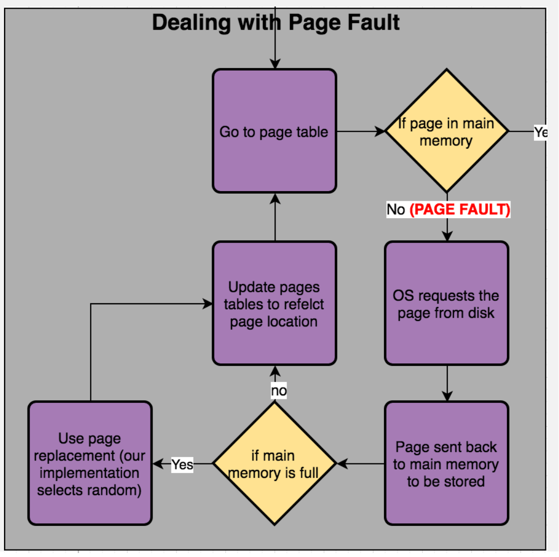
With the shortest process next, if there is a fault, the process is not terminated but instead killed by causing it to fail for one clock and then it is added back to the queue with the original nice value.

DIAGRAM

SEGMENTATION FAULT


Dealing with Segmentation Fault
Obtain segementaton table number
If seg number is out of range
No
Find desired page table
Yes
Terminate Process

PAGE FAULT


Dealing with Page Fault
Go to page table
If page in main memory
Ye
No (PAGE FAULT)
OS requests the page from disk
Page sent back to main memory to be stored
if main memory is full
Yes
Use page replacement (our implementation selects random)
no
Update pages tables to refelct page location

KEY CODE-RELATED EXPLANATION OF IMPORTANT ASPECTS

An important aspect of the approach is encapsulation within OOP. Each class and object reflect a logical ownership and encapsulation to better represent the overall objective of a virtual memory manager. Also, this allows for the implementation to be scaled to more processes.

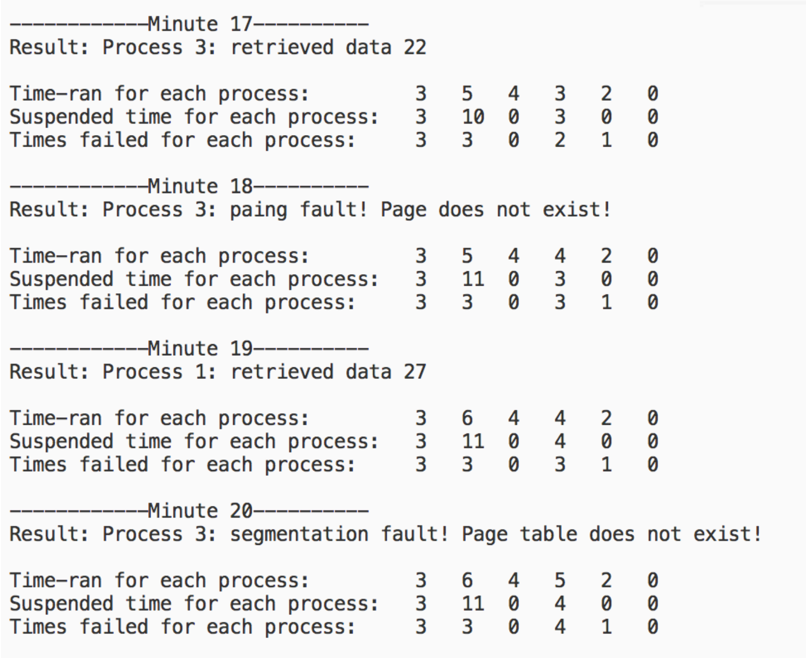
Virtual memory is randomly generated, and the random number generator are global variables.

The value of the data is the sum of all digits in the virtual memory of the data.

Segmentation and page faults result in a context switch, causing the process to fail and be suspended for one cycle. This could be changed to kill the process but for the model, we allowed the process to be added after one cycle.

Nice values or priority values are set to the estimated time it takes to complete the process as entering a ready state.

CODE EXECUTION



```
------------Minute 17----------
Result: Process 3: retrieved data 22

Time-ran for each process:         3   5   4   3   2   0
Suspended time for each process:   3   10  0   3   0   0
Times failed for each process:     3   3   0   2   1   0

------------Minute 18----------
Result: Process 3: paing fault! Page does not exist!

Time-ran for each process:         3   5   4   4   2   0
Suspended time for each process:   3   11  0   3   0   0
Times failed for each process:     3   3   0   3   1   0

------------Minute 19----------
Result: Process 1: retrieved data 27

Time-ran for each process:         3   6   4   4   2   0
Suspended time for each process:   3   11  0   4   0   0
Times failed for each process:     3   3   0   3   1   0

------------Minute 20----------
Result: Process 3: segmentation fault! Page table does not exist!

Time-ran for each process:         3   6   4   5   2   0
Suspended time for each process:   3   11  0   4   0   0
Times failed for each process:     3   3   0   4   1   0
```

RESOURCES

BIT BUCKET

https://bitbucket.org/shaun86wang/cst-315_wang_vandyke/src/0e64961c553e2a259e6156be4832ea6d8eb72684/Project3/?at=master

REFERENCES

Stallings, W. (2015). Operating Systems: Internals and Design Principles . Harlow: Pearson.