Dynamic Resource Allocation and Scheduling

Team Members:-
Amit Vyas - 16
Anshul Mankani - 43
Saksham Pandey - 35
Suyash Singh - 36
Yogesh Choudhary - 37

Introduction

Some of the main functions that an operating system performs are resource allocation for processes and optimal scheduling of the processes. An operating system (OS) requires that a process must be given the resources it requires at the earliest, in order to complete its execution. However, it might not always be possible for the OS to allow a particular process to use all resources it has asked for. A process might be asked to wait for the resources because they are being used by another process. There might not be enough instances, of a particular resource, for all processes to use it at the same time. There are devised algorithms for this kind of resource allocation, one of them being Banker's algorithm.

A scenario might occur where none of the currently running processes are able to continue their execution because they're waiting for resources. Such a situation is known as a deadlock situation. An OS should use any deadlock avoidance or deadlock prevention techniques to ensure that such situations do not take place.

Another responsibility of the operating system is to incorporate a scheduling technique. It is common to have multiple processes running at the same time on the same processor, irrespective of a uniprocessor or a multiprocessor system. But not all processes can execute at the same instance on a single processor. Thus, we need a scheduling technique to manage the execution times of all running processes. Some of the common scheduling techniques include FCFS, Shortest Job First (SJF) scheduling, Priority scheduling, Round Robin (RR) scheduling, etc.

In this project, we have implemented dynamic resource allocation – a new process can be added to the ready queue at any point of time. Also, any running process can either request a new resource, release a resource or abort the entire process dynamically. All the requests from any process are evaluated after a set time frame in the program and will be granted, if the system seems to be safe with the new request. If not, the request is kept on hold and the requesting process is made to wait until it's safe to grant that request. The scheduling technique used in our program is Round Robin scheduling. We have kept some parameters fixed throughout the program for simplicity i.e. maximum no. of resources, time quantum for RR scheduling.

We shall discuss further each of these functionalities and how they've been implemented.

Methodology

While the entire program for our project has been developed in C language, the simulation graphs for the same have been implemented with Python. We shall discuss the methodology in three main parts:

1. Dynamic Resource Allocation
2. Round Robin Scheduling
3. Simulation Graphs

Dynamic Resource Allocation

As mentioned earlier, we have involved functionalities which allow the user to add a new process, request or release a resource for a running process or abort a running process dynamically. A structure for every process and every request allows us to use linked lists effectively and keep track of everything. A ready queue holds all processes which are currently running and the process at the front is the process currently executing. The processes keep getting pushed to the end of the queue, as a part of the scheduling technique which we discuss later. A 2D linked list has been implemented to accommodate multiple requests from multiple processes. At any point of time, all modifications are being concurrently made. For ex, if a process is dynamically aborted, all resources allocated to the process are freed and all its pending requests are also deleted. Then, the process is removed from the ready queue.
A log of all requests is kept and all pending requests of the currently executing process are checked to see if they can be granted. In case the request is for releasing a resource, it is granted conveniently without the need to check anything else. However, if any process requests a new resource, it is first checked if the system would be safe if the request is granted. If there aren't enough instances of that resource then that process is pushed to the back of the queue and is made to wait till its request can be granted. The inputs from the user are validated and only then accepted to ensure no discrepancies with respect to resource allocation. In case of a deadlock, the program terminates after aborting all processes and freeing their resources, thus, preventing a deadlock. We plan on shifting to a more efficient deadlock avoidance strategy later.

Round Robin Scheduling

This scheduling technique involves the use of a fixed time quantum i.e. the amount of CPU time given to a process before it is preempted by another process. All the process held in the ready queue are given an equal chance to execute, thus, considerably reducing the overall wait time and turnaround time for each process. After every time quantum, the process which was at the front is pushed back to the end of the queue. This ensures that a single process does not have control over the CPU for a longer duration of time while another process is kept waiting throughout. Simultaneously, we make a function call to grantRequests( ) which takes care of all the pending requests of each process. While one process is under execution, the wait time for all other currently running processes increases.

Simulation Graphs

We have simulated the wait times and the resource allocation for each process. Our C code includes creating a file with all the required data for the graphs and this file is updated regularly. Simultaneously, our python code fetches the data from this file and plots the graphs for this data, which also keeps changing dynamically. The python code also makes use of multiple libraries to solve our purpose, as shown in the code later.

Result


Figure 2
Resource Allocation Graph
Resources
ProcessID
1000
1001
1002
Wait Time Graph
Wait Time
ProcessID
Resource 0
Resource 1
Resource 2
Resource 3
Resource 4
Wait Time
OS-Project — asd ◂ python MainFile.py — 52...
6.Print Log
7.Continue Execution
8.Exit
7
Running: Process 1000
Remaining Burst Times:
Process ID - 1000 Remaining time - 6
Process ID - 1002 Remaining time - 23
Process ID - 1001 Remaining time - 7
1.Enter a new process.
2.Request a new resource.
3.Release a resource.
4.Abort a process
5.Show process status
6.Print Log
7.Continue Execution
8.Exit
5
Process Status:
Running process -> Process ID 1002.
ID - 1002
Resource 0 - 0
Resource 1 - 0
Resource 2 - 0
Resource 3 - 4
Resource 4 - 0
ID - 1001
Resource 0 - 7
Resource 1 - 7
Resource 2 - 7
Resource 3 - 3
Resource 4 - 7
ID - 1000
Resource 0 - 3
Resource 1 - 3
Resource 2 - 3
Resource 3 - 3
Resource 4 - 3
1.Enter a new process.
2.Request a new resource.
3.Release a resource.
4.Abort a process
5.Show process status
6.Print Log
7.Continue Execution
8.Exit


Process Status:
Running process -> Process ID 1003.
ID - 1003
Resource 0 - 7
Resource 1 - 4
Resource 2 - 5
Resource 3 - 3
Resource 4 - 4
ID - 1002
Resource 0 - 0
Resource 1 - 0
Resource 2 - 0
Resource 3 - 4
Resource 4 - 0

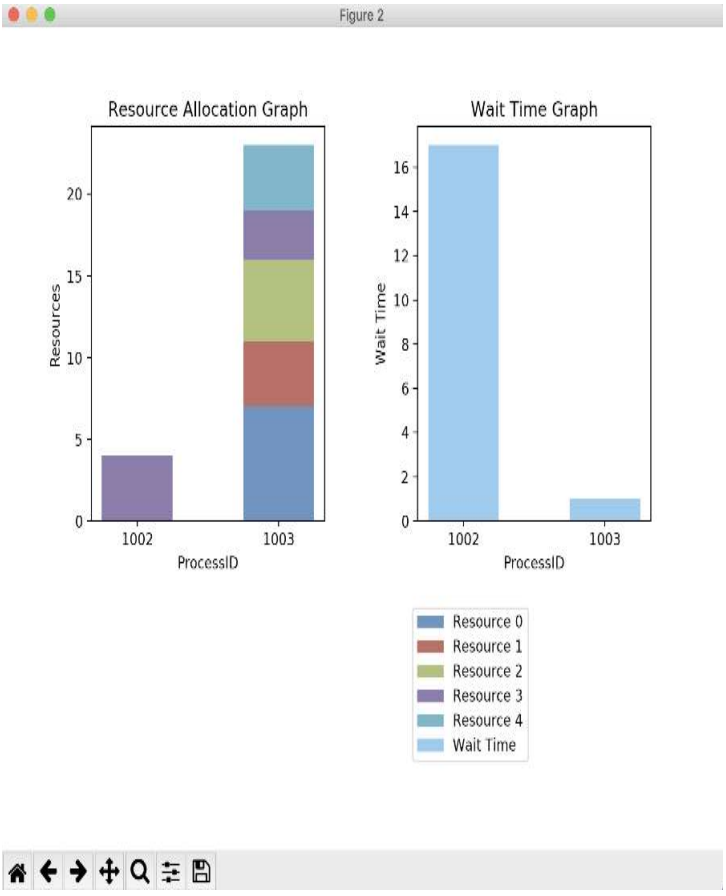
Figure 2
Resource Allocation Graph
Resources
ProcessID
1002
1003
Wait Time Graph
Wait Time
ProcessID
Resource 0
Resource 1
Resource 2
Resource 3
Resource 4
Wait Time

Conclusion

We have successfully merged the concepts of resource allocation and scheduling for an operating system while accommodating the dynamic user requests. A simulation of the system shows that the wait times of the processes is considerably low compared to other scheduling techniques and it also shows the pattern of the resources being used. This data could be very helpful to figure out what kind of resources are more used, how long are they held by any process, etc. We have clearly shown how Round Robin scheduling would be beneficial for our purpose. Several OS concepts have been implemented in this project. The rigorous use of queues and linked lists in our project has given a clear understanding of the data structures and algorithms concepts.
Our future work involves a better method of deadlock avoidance and also to completely remove a starvation condition for any process. A priority based scheduling could be implemented for this purpose.