# Project 3: AUbatch- A Pthread-based Batch Scheduling System

Satyam P. Todkar

szt0064@auburn.edu

Department of Computer Science & Software Engineering

Auburn University

## 1. Introduction:

The report is about building a batch scheduling system based upon pthread library which can simulate the scheduling and execution of processes (also known as jobs) in the AUbatch system. The system makes extensive use of the pthread library and functions that provide communication & synchronization mechanism. There are two thread viz Scheduler and Dispatcher which are assigned the task of scheduling the jobs (based on some policy e.g. fcfs, sjf) and execution of the jobs respectively. For execution, the function execv is used instead of the system and the sample_job of matrix multiplication is considered. Though the problem can be treated as a simple producer consumer problem the challenge lies in synchronization of the two threads. The synchronization is achieved using pthread's mutexes and condition variables.

### 1.1 Design Document:

**The data flow diagram (DFD)** forms the basis of the design for the project. It shows how information (data) flows between different processes. However, there is no presence of decision-making elements that are found in a flowchart. The DFD for the AUbatch can be visualized in Figure 1.

Initially, the user issues a command to the command line parser. The module then decides the type of command issued (run, help, display etc.) and calls the respective modules to perform the computation. The flow of data is represented by text over the arrow and the data flows in the direction of the arrowhead. When a user issues a command to schedule a job the system calls the schedule module which takes two input parameters- job and the policy to schedule the job. It would then add the job to the Job Queue. As soon as there is a job in the queue the dispatching module consumes it and once consumed removes it from the job queue. A job has many attributes like ID, priority, CPU time, status, arrival time, completion time etc. all of which is discussed in the Implementation section. The DFD are used to get a clear understanding of the system and how different parts of the system interact with each other, they do not consider the intricate implementation details like synchronized communication that is needed in our AUbatch system.

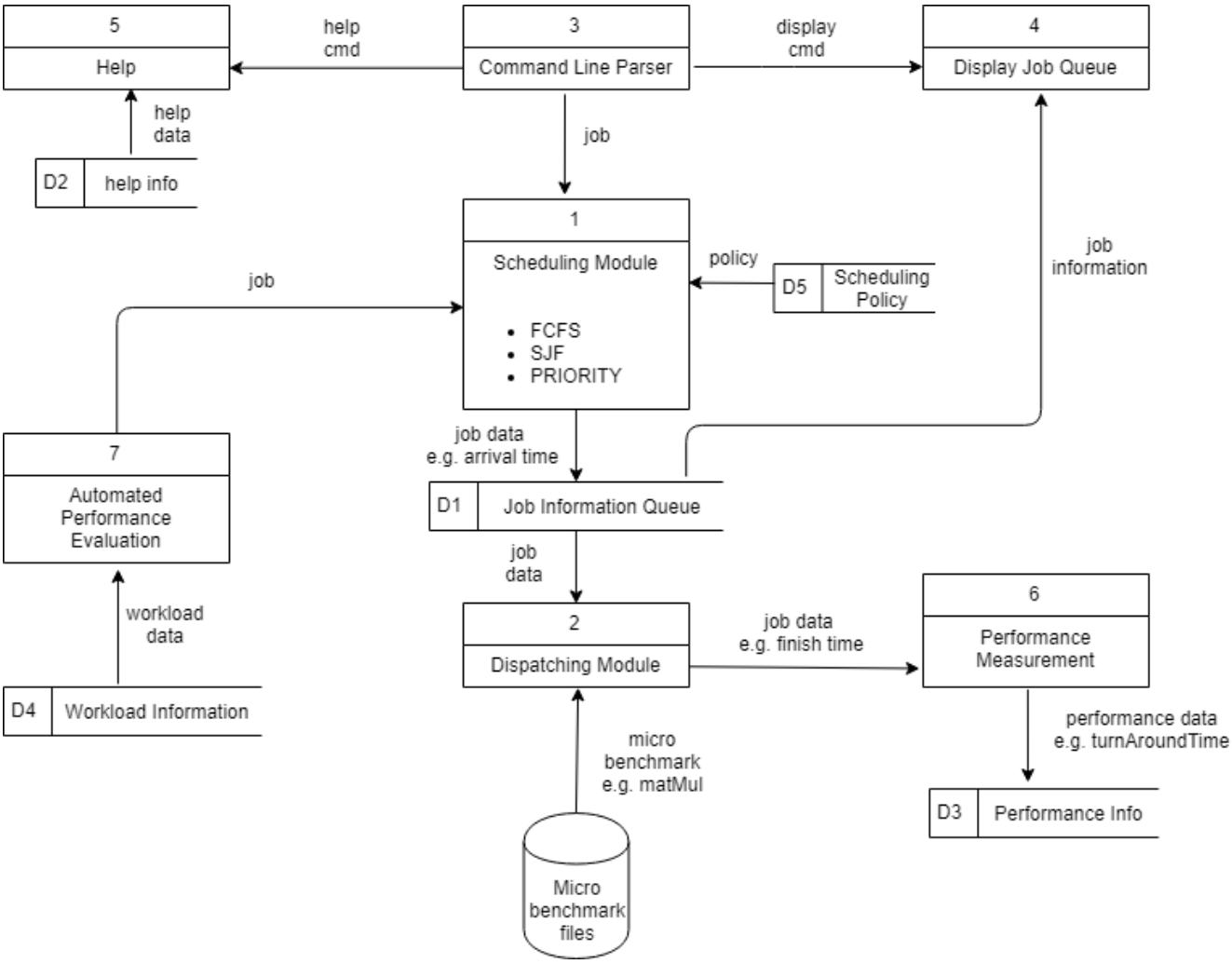


**Fig. 1. AUbatch Data Flow Diagram**

**The Program Structure** for the AUbatch system can be visualized from its DFD. Fig. 2 represents the program structure of the AUbatch system. It is used to systematically breakdown a system to its lowest manageable level. Here, the modules in the DFD participate in the program structure and are represented as rectangles along with their names. AUbatch is the system whereas modules numbered from 1 to 7 form the system. Another key aspect of the AUbatch system is the performance measurement and the performance evaluation modules that are used for comparing one scheduling policy to another under different load conditions such as arrival rate, load distribution, number of submitted jobs etc. A brief discussion about this is done in the following section.

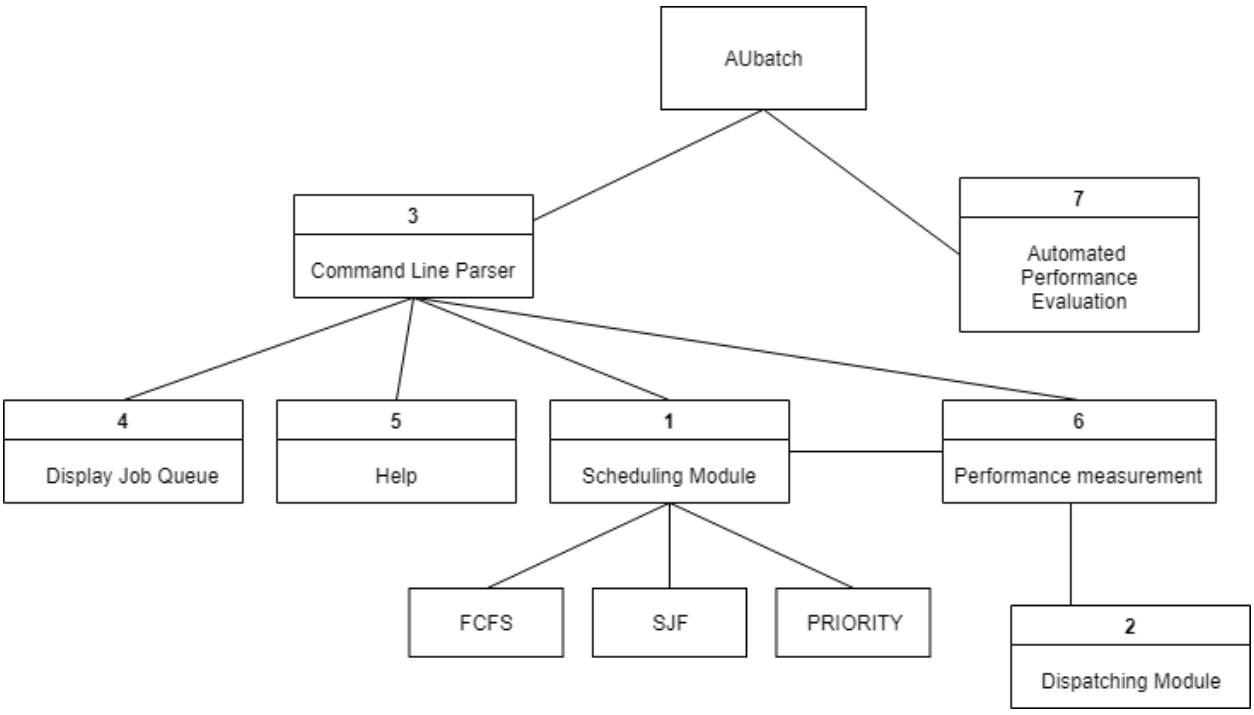


**Fig. 2. AUbatch Program Structure**

### 1.2 Performance Metrics and Workload:

Performance is one of the most important and crucial factors that decide the fate & usefulness of a software system. As per the design document, AUbatch has a separate dedicated performance measurement module that in coordination with the scheduling and the dispatching module measure different parameters like turn around time and waiting time for a job. How well a system performs is directly related to the performance of the system. The metrics like turn around time and waiting time need to be defined and should be taken as consistent measure throughout the experiment.

We define waiting time as the difference between the amount of time taken by the job to execute and the time at which the job first arrives in the system. We define turn around time as the time taken by the job from the moment it enters to the moment it exits the system. These measures need to be found per job which can be later used to find the average waiting time and the average turn around time to determine the throughput. The throughput is the actual measure of how well the system performs and is used as a tool for comparison of the performance of different scheduling algorithms.

Besides the Scheduling policy used the throughput of the system is sensitive to different workload conditions like Number of Submitted jobs, Arrival Rate, Load Distribution etc. The Automated Performance Evaluation system uses these workload conditions to determine the throughput of the system (under specific scheduling policy)

Sample Workload Condition for the Automatic Performance Evaluation as shown in Table 1. One should check the performance of the system for a specific condition like (Number of Submitted Jobs) keeping the other variables constant

| Workload Parameters | Default Values |
|---|---|
| Number of Submitted Jobs | 5, 10, 15, 20 |
| Arrival Rate | 1, 2 |

**Table 1: Workload Conditions**

### 1.3 Performance Comparison

As discussed earlier the average computation like average waiting time, average turn around time is more important than the individual waiting time, turn around time per job. The results achieved can be closely related to the theoretical implications such as Shortest Job First outperforms the First Come First Serve scheduling policy. This can be validated by the throughput achieved when we scheduled jobs firstly using FCFS, then using SJF and PRIORITY based scheduling. However, it should be evident that SJF and PRIORITY based scheduling starve the jobs that have a longer CPU time or Lower Priority respectively.

Consider the following run of jobs and the output it produced

**A. FCFS**

```
[stodkar@localhost project3]$ ./aubatch

Welcome to Satyams's batch job scheduler Version 1.0

Type 'help' to find more about AUBatch commands.


>run p 7 1

Job p was submitted.

Total number of jobs in the queue: 1

Expected waiting time: 0.000000 seconds

Scheduling Policy: FCFS

>run p 2 5

Job p was submitted.

Total number of jobs in the queue: 2

Expected waiting time: 0.000000 seconds

Scheduling Policy: FCFS
```

```
>run p 3 1
Job p was submitted.
Total number of jobs in the queue: 2
Expected waiting time: 5.000000 seconds
Scheduling Policy: FCFS
>list

Total number of jobs in the queue: 2
Scheduling Policy: FCFS
Name  CPU_Time    Pri   Arrival_time      Progress
p     2           5           23:54:47    Waiting
p     3           1           23:54:54    Waiting
>list

Total number of jobs in the queue: 0
Scheduling Policy: FCFS
Name  CPU_Time    Pri   Arrival_time      Progress
>quit

Total jobs submitted : 3
Average waiting time : 14.666667
Average turn around time : 27.666666
Throughput : 0.036145[stodkar@localhost project3]$
```

**B. SJF**

```
[stodkar@localhost project3]$ ./aubatch
Welcome to Satyams's batch job scheduler Version 1.0
Type 'help' to find more about AUBatch commands.

>sjf
```

```
Scheduling policy is switched to SJF. All the 0 waiting jobs have been
rescheduled.
>run p 7 1
Job p was submitted.
Total number of jobs in the queue: 1
Expected waiting time: 0.000000 seconds
Scheduling Policy: SJF
>run p 5 2
Job p was submitted.
Total number of jobs in the queue: 2
Expected waiting time: 0.000000 seconds
Scheduling Policy: SJF
>run p 3 1
Job p was submitted.
Total number of jobs in the queue: 2
Expected waiting time: 5.000000 seconds
Scheduling Policy: SJF
>quit

Total jobs submitted : 3
Average waiting time : 16.000000
Average turn around time : 30.333334
Throughput : 0.032967[stodkar@localhost project3]$
```

**C. PRIORITY**

```
[stodkar@localhost project3]$ ./aubatch
Welcome to Satyams's batch job scheduler Version 1.0
Type 'help' to find more about AUBatch commands.
```

```
>priority


Scheduling policy is switched to PRIORITY. All the 0 waiting jobs have been
rescheduled.
>run p 7 1
Job p was submitted.
Total number of jobs in the queue: 1
Expected waiting time: 0.000000 seconds
Scheduling Policy: PRIO
>run p 5 2
rJob p was submitted.
Total number of jobs in the queue: 1
Expected waiting time: 0.000000 seconds
Scheduling Policy: PRIO
>un p 3 1
Job p was submitted.
Total number of jobs in the queue: 2
Expected waiting time: 0.000000 seconds
Scheduling Policy: PRIO
>quit


Total jobs submitted : 3
Average waiting time : 10.666667
Average turn around time : 19.000000
Throughput : 0.052632[stodkar@localhost project3]$
```

**Observations & Conclusion**

- The Fig. 3 and Fig. 4 are the performance comparison for different scheduling policies and under different arrival rate and the number of jobs. From these figures it is evident that as the number of jobs increase the throughput decreases.
- One can easily verify that for the given load conditions in automated test throughput of SJF is greater than the throughput of PRIORITY which again is greater than the throughput of FCFS. However, the order may change when the number of jobs are small or the number of jobs with higher priority or with lower cpu time is more

  Considering the mentioned input/output for scheduling only 3 jobs
- Let the average waiting time is denoted by **Avg. WT**
  **(Avg. WT) PRIORITY < (Avg. WT) FCFS < (Avg. WT) SJF**
- Similarly, let average turnaround time is denoted by **Avg. TAT**
  **(Avg. TAT) PRIORITY < (Avg. TAT) FCFS < (Avg. TAT) SJF**
  However, SJF outperforms the two for increased load conditions. The achieved results are in accordance with the intuition one has for these scheduling algorithms and hence we have successfully conducted the experiment.

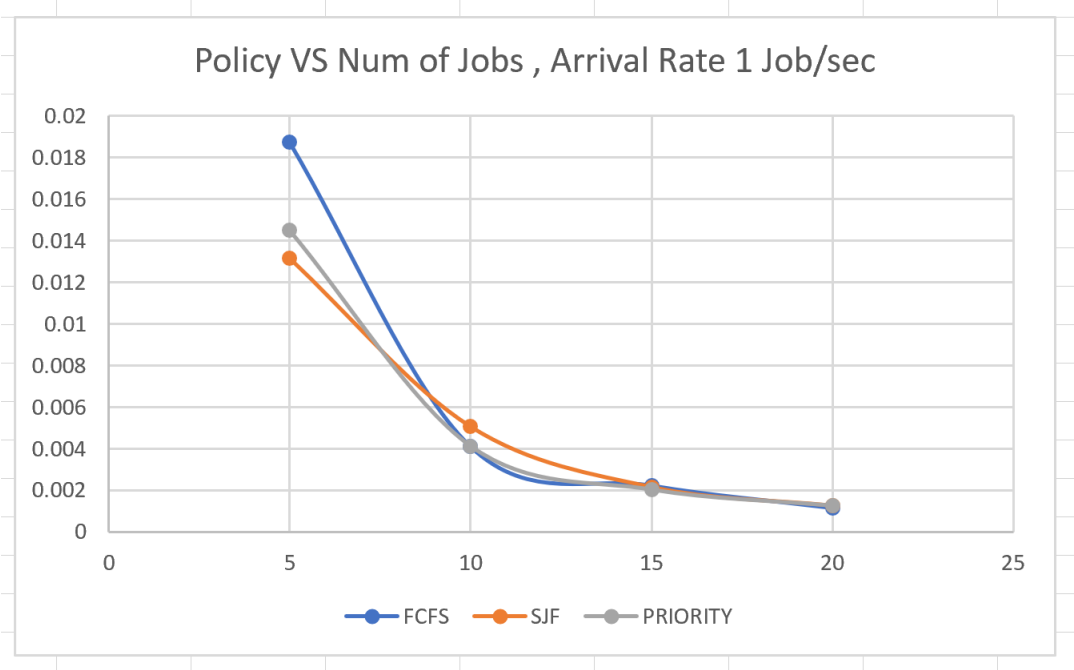


**Fig. 3. Policy Vs Number of Jobs (Arrival Rate 1 Job/sec)**

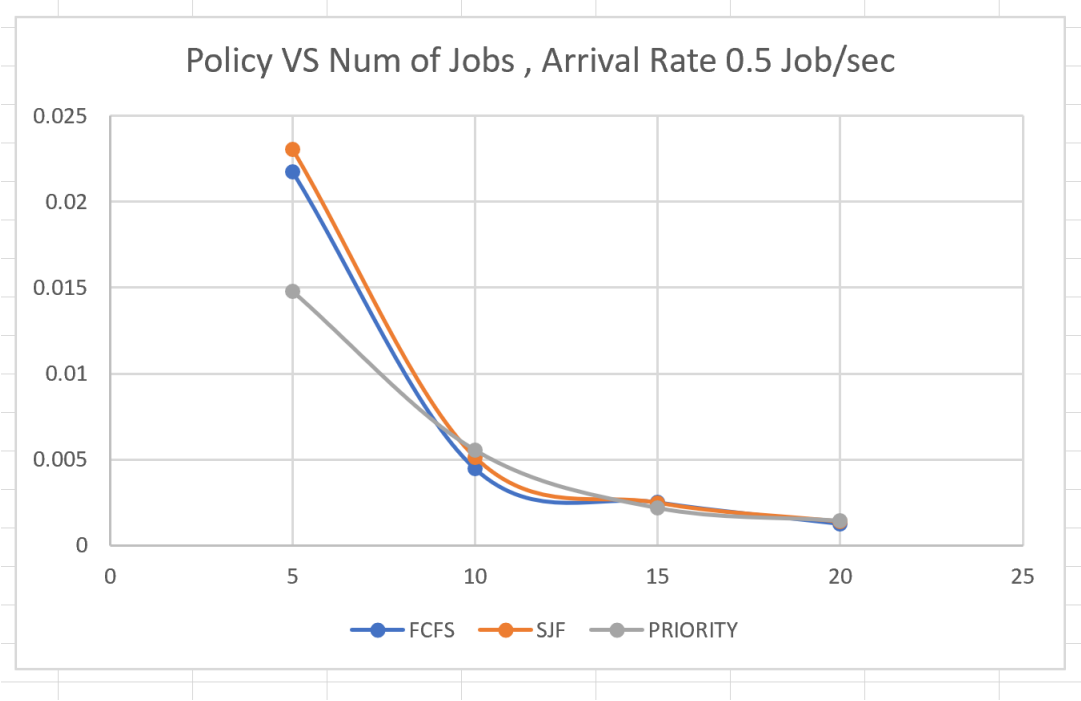


**Fig. 4. Policy Vs Number of Jobs (Arrival Rate 0.5 Job/sec)**

### 1.4 Lessons Learned

AUbatch- A Pthread based Batch Scheduling system formed an elegant yet sophisticated system that helped us understand the level of complexity that is present in a scheduling dispatching system. These scheduler and dispatchers form the basis of many systems in our day to day life and their malfunctioning may formidably cause erroneous situations. There are many take away from this project-

1. The synchronization mechanisms like mutexes, semaphores and condition variables form the heart of a scheduling dispatching application like AUbatch. If theses synchronization is not handled carefully it may lead to an undesirable condition called deadlock. The situation of deadlock arises when a process holds a resource that is needed by the other process and vice versa.
2. The two threads scheduler and dispatcher are pretty fast and if the job queue has no jobs schedule yet it may be possible that while the scheduler is trying to schedule a job the dispatcher has already assumed a lock on the resource (here mutex) and goes to sleep while the scheduler waits for the resource (lock) to be free. This may lead to starvation and loss of computing resources and may have some adverse effects like a Race condition. In order to solve this issue, we deliberately make the dispatcher sleep for an arbitrary time e.g. sleep(10) so that the producer i.e. the scheduler can assume a lock and schedule a job
3. We need to make the scheduler and dispatcher threads independent of each other's execution i.e. if the dispatcher is running a task scheduler can also run meanwhile and thus interact with the user. However, the program waits for the completion of execv function in

dispatcher and thus cutting off the user interaction with the scheduler thread. This was solved by spawning a child for the dispatching process which is entitled with the task of only performing the execv function. One should keen in mind that the location of the prompt does not necessarily suggest that the scheduler or dispatcher is accepting jobs. This is because the terminal is shared by both the threads and they output independent of each other. Thus, one can submit jobs while other thread is executing the job.

4. The automation test module is implemented by the test command. This command simulated the scheduling of jobs by automatically adding enlisted number of jobs in the command. It makes use of the pre created run function.

**Shortcomings:**

- Even though while scheduling the scheduler needs a lock over the resource (mutex lock on the queue) before creating an order it is seen that all the jobs are scheduled correctly and a correct order is found except sometimes the job at the last position is not ordered properly. The loop for ordering (policies) iterate from tail+1 to head-1 as tail holds the job in execution and head points to a location in the queue for insertion of new job.

**Functions Used:**

I. **difftime:** This function returns the difference of two times (time1 - time2) as a double value.

double difftime(time_t time1, time_t time2)

time1 − This is the time_t object for end time.
time2 − This is the time_t object for start time.

II. **sprintf:** sends formatted output to a string pointed to, by str.

int sprintf(char *str, const char *string,...)

str − This is the pointer to an array of char elements where the resulting C string is stored.
format − This is the String that contains the text to be written to buffer.

III. **sscanf:** reads formatted input from a string.

int sscanf(const char *str, const char *format, ...)

str − This is the C string that the function processes as its source to retrieve the data.
format- This specifies the format in which we need to read the string

IV. **strtok:** breaks string str into a series of tokens using the delimiter delim.

char *strtok(char *str, const char *delim)

str − The contents of this string are modified and broken into smaller strings (tokens).
delim − This is the C string containing the delimiters.

This function returns a pointer to the first token found in the string. A null pointer is returned if there are no tokens left to retrieve.

V. **strstr:** function finds the first occurrence of the substring

char *strstr(const char *string, const char *sub_string)

string − This is the main C string to be scanned.

sub_string − This is the small string to be searched with-in haystack string.

This function returns a pointer to the first occurrence in string of any of the entire sequence of characters specified in needle, or a null pointer if the sequence is not present in string.

VI. **strcmp**: compares the string pointed to, by str1 to the string pointed to by str2.

int strcmp(const char *str1, const char *str2)

str1 − This is the first string to be compared.
str2 − This is the second string to be compared.

if Return value < 0 then it indicates str1 is less than str2.
if Return value > 0 then it indicates str2 is less than str1.
if Return value = 0 then it indicates str1 is equal to str2.

**Running the Code**

- The program is written in aubatch.c file and needs to be compiled using **gcc aubatch.c -o aubatch -lpthread** The execution of which is simply done by issuing **./aubatch**
- The run command in the aubatch takes <job> as one of its input parameters and executes **./sample_job** which is the executable of a Matrix Multiplication code sample_job.c The compilation of which is **gcc -o sample_job sample_job.c**
- The output from the aubatch is stored in the scripts folder for every test run carried out.