COMP 7500/7506 Advanced Operating Systems

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

Document History: Created on Feb. 11, 2018. Revised on Feb. 18, 2020. **Version 2.2**
Points Possible: **100**
Submission via **Canvas**

This is an individual assignment; no collaboration among students. Students shouldn't share any project code with any other student. Collaborations among students in any form will be treated as a serious violation of the University's academic integrity code.

Learning Objectives:

- To design a batch scheduling system
- To evaluate three scheduling policies/algorithms
- To implement a scheduler where two threads are synchronized
- To learn POSIX Threads Programming
- To use the `pthread` ligrary and `execv()` functions
- To study and apply the condition variables using the `Pthread` library
- To address synchronization issues in the scheduler
- To develop micro batch-job benchmarks
- To design of performance metrics
- To assess various workload conditions
- Use the GDB tool to debug your C program in Linux
- To strengthen your debugging skill
- To improve your software development skills
- To boost your operating systems research skills

1. Project Overview

The goal of this project is to design and implement a batch scheduling system called *AUbatch* using the C programming language and the `Pthread` library. AUbatch is comprised of two distinctive and collaborating threads, namely, the *scheduling* thread and the *dispatching* thread. The scheduling thread enforces scheduling policies, whereas the dispatching thread has submitted jobs executed by the `execv()` function. The two threads are created by the `pthread_create ()` function (see also [2] for an example).

The *synchronization* of these two threads must be implemented by *condition variables*. To address the synchronization issues in AUbatch, you may envision the scheduling and dispatching modules as a producer and a consumer, respectively. In addition to

condition variables, mutex must be adopted to solve the critical section problem in AUbatch.
The three scheduling policies to be implemented in your AUbatch are FCFS (a.k.a., First Come, First Served), SJF (a.k.a., Shortest Job First), and Priority (a.k.a., Priority-based scheduling). Please refer to [3] for details on these three scheduling algorithms.

After implementing three scheduling policies (i.e., FCFS, SJF, and priority based), you are required to compare the performance of these three scheduling policies under various workload conditions.

2. AUbatch: A Batch Scheduling System

2.1 System Architecture

Fig. 1 depicts the system architecture of the *AUbatch* scheduling system, which consists of a *scheduling* module and a *dispatching* module. The scheduling module is in charge of the following two tasks:

(1) accepting submitted jobs from users and
(2) enforcing a chosen scheduling policy.

The dispatching module have two responsibilities listed below:

(1) making use of the `execv()` function to run jobs sorted in the job queue and
(2) measuring the execution time and response time (a.k.a., turn-around time) of each finished job.

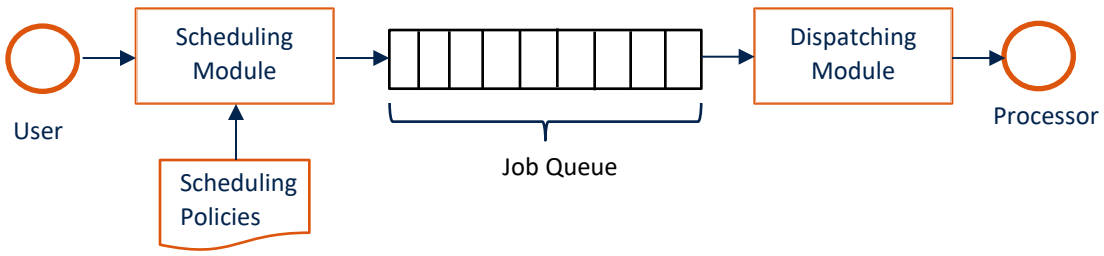

Fig. 1 The system architecture of the AUbatch scheduler

2.2 System Design

A data flow diagram or DFD illustrates how data is processed by your AUbatch system in terms of inputs and outputs. In this project, you will be required to draft a data flow diagram for your AUbatch system. Fig. 1 outlined in Section 2.1 should be an inspiration when you design the data flow diagram, which is a driving force behind the design of AUbatch's important data structures and function prototypes. In one of the lectures pertaining to project 3, Dr. Qin will offer you feedback on your DFD designs.

Please keep in mind that during the system design phase, you shouldn't be concerned

with the implementation details like how to use the `execv()` function and how to apply the `pthread` library. You should be capable of crafting a simple yet elegant system design without prior knowledge of *Pthread* and *multithreaded* programming.

Please refer to [4] for commonly used data flow diagrams symbols. A few sample data flow diagrams also can be found in [4].

2.3 Two Optional Modules for Job Submissions and Performance Measurements

It is worth noting that if the job submission portion in the scheduling module becomes complicated, it is prudent to separate job submission from the scheduling module. Thus, you should create another module called the job submission module.

Similarly, in case the performance measuring part of the dispatching module is non-trivial, it is a wise decision to implement a dedicated module for performance measurement and comparison.

2.4 Critical Sections and Synchronizations

The scheduling module and the dispatching module are launched as two concurrent threads created by the `pthread_create ()` function (see also [2] for an example). The two threads share a job queue, meaning that you must incorporate the mutex mechanism in AUbatch to protect the shared job queue residing in critical sections. You may identify a few shared variables other than the job queue, please apply mutex available in the *PThread* library to implement all the critical sections.

Apart from critical sections, the *synchronization* between the scheduling module and the dispatching module should be addressed using *condition variables,* the functions of which are available in the PThread library [1].

The synchronization design pattern of AUbatch is the conventional producer-consumer problem [5]. To tackle the synchronization problem in AUbatch, you may simple treat the scheduling module as a producer and the dispatching modules as a consumer. More specifically, the scheduling module issues (i.e., produces) newly submitted jobs into the job queue, enforcing a scheduling policy; the dispatching module extracts (i.e., consumes) a job from the job queue to run the job.

Important! You are required to take the full advantage of the PThread library to make use of condition variables and mutexes to solve the aforementioned critical section and synchronization problems in AUbatch. Please refer to [1] for the detailed information on the PThread library. Dr. Qin will introduce PThread programming in one of the project 3 lectures. A list of important PThread functions to be employed in this project is given below.

- `pthread_create (thread,attr,start_routine,arg)`

The mutex functions:

- `pthread_mutex_lock (mutex)`
- `pthread_mutex_unlock (mutex)`

The condition-variable functions:

- `pthread_cond_init (condition,attr)`
- `pthread_cond_wait (condition,mutex)`
- `pthread_cond_signal (condition)`

Sample Code In this project-specification package, you may find an example (see `condvar.c`) of using condition variables. This example source code demonstrates the use of a few commonly used Pthread condition variable functions. In `condvar.c`, the main function creates three threads, two of which perform work and update a "`count`" variable. The third thread waits until the `count` variable reaches a specified value.

2.5 Three Scheduling Policies/Algorithms

In this project, you must implement three scheduling policies in AUbatch. Throughout this specification document, we use terms policies and algorithms interchangeably. The three algorithms to be investigated in the project are:

- FCFS: First Come, First Served,
- SJF: Shortest Job First, and
- Priority: Priority-based scheduling.

Important! You may refer to [3] for the detailed descriptions on these three scheduling algorithms. The three algorithms should be implemented in the scheduling module of AUbatch (see also Fig. 1 in Section 2.1).

3. Performance Evaluation

3.1 Performance Metrics

After implementing the three scheduling policies (i.e., FCFS, SJF, and Priority), you will be in a position to compare the performance of these algorithms under various workload conditions (see also Section 3.3). The first step toward evaluating and comparing the performance of the three scheduling algorithms is to design performance metrics (see the list below) and workload conditions (see Section 3.3). Performance metrics measure your AUbatch's behaviors and performance. The two suggested performance metrics are:

- Average Response Time
- Throughput

You may also consider the following optional performance metrics:

- Maximum Response Time
- Minimum Response Time
- Response Time Standard Deviation

Important! If you are a doctoral student who intends to improve your research skills, you should evaluate the algorithms using all the above performance metrics. Master's students with strong programming skills are recommended to test the algorithms using all the possible metrics.

3.2 Micro Batch-Job Benchmarks

After the scheduling module makes scheduling decisions, the dispatching module launches a job from the head of the job queue with the `execv()` function. Jobs submitted to the scheduling module can be batch jobs, which may be either large batch-job benchmarks or micro batchmarks. You are suggested to implement a micro batch-job benchmark, which takes running-time as an input parameter. For example, let's consider a micro benchmark called "`batch_job`", the format of which is "`batch_job <exe_time>`". You may run `batch_job` from the Linux terminal as,

```
$./batch_job 5
```

This micro benchmark will run for approximately 5 seconds.

Important! Your dispatching module should be capable of launching your micro benchmarks. Because your micro benchmarks are in the batch mode, there shouldn't be any output displayed in the Linux terminal. In case your benchmarks must output any data, the data should be saved in output files.

3.3 Workload Conditions

We model a workload condition using three parameters, namely, the number of submitted jobs, arrival rate, and load distribution.

- The number of submitted jobs: the number of job submitted in each experiment.
- Arrival rate: the number of job submitted per time unit (e.g., 1 second).
- Load Distribution: distribution of batch jobs' execution times.

The sample workload-condition setup is summarized in the following table. You may use any setup to conduct your experiments.

Workload Parameters	Default Values
Number of Submitted Jobs	5, 10, 15, 20, 25
Arrival Rate	0.1, 0.2, 0.5, 1.0 No./Second

Load Distribution (Uniform)	[0.1, 0.5], [0.1, 1], [0.5, 1], [1, 10]

The load distribution can be either uniform or normal distribution. The implementation of the uniform distribution is, of course, easier than that of the normal distribution.

3.4 Automated Performance Evaluation

Important! It is impractical to evaluate performance using AUbatch's command line. Please propose a way of automatically evaluating performance of the three scheduling algorithms. You are suggested to implement a performance evaluation module, where all the designed workload conditions are implemented through hard coding.

4. User Interface

The commonly used commands in AUbatch are (1) help, (2) run, (3) list, and (4) change scheduling policies. In what follows, we show the sample dialog for each command.

Important! Your program's output should match the style of the sample output. You will lose points if you don't follow the user interface specified in this Section.

4.1 Help Information

The name of your batch-job scheduler is `aubatch`, which takes no input parameter. Here is a sample dialog (where the user input is depicted as **Bold**, but you do not need to display user input in bold.). In the sample usage below, please replace "Dr. Qin" with your name.

```
$./aubatch
Welcome to Dr. Qin's batch job scheduler Version 1.0
Type 'help' to find more about AUbatch commands.
>help
run <job> <time> <pri>: submit a job named <job>,
                        execution time is <time>,
                        priority is <pri>.
list: display the job status.
fcfs: change the scheduling policy to FCFS.
sjf: change the scheduling policy to SJF.
priority: change the scheduling policy to priority.
test <benchmark> <policy> <num_of_jobs> <priority_levels>
     <min_CPU_time> <max_CPU_time>
quit: exit AUbatch
>
```

4.2 Submit a Job

A batch job is submitted using the "`run`" command in AUbatch. A sample dialog of job submission is given below.

```
$./aubatch
Welcome to Dr. Qin's batch job scheduler Version 1.0
Type 'help' to find more about AUbatch commands.
>run sample_job 10
Job sample_job was submitted.
Total number of jobs in the queue: 4
Expected waiting time: 113 seconds
Scheduling Policy: FCFS.
>
```

After a batch job is submitted, AUbatch displays important information including (1) total number of submitted jobs, (2) excepted waiting time, and (3) scheduling policy.

4.3 Display the Job List

All the jobs submitted to the job queue can be displayed using the "`list`" command in AUbatch. A sample dialog of list submitted jobs is outlined as follows.

```
$./aubatch
Welcome to Dr. Qin's batch job scheduler Version 1.0
Type 'help' to find more about AUbatch commands.

>list
Total number of jobs in the queue: 4
Scheduling Policy: FCFS.
Name CPU_Time Pri Arrival_time Progress
job1 59       3   07:12:13       Run
job2 127      1   07:12:35
job3 8        2   07:13:26
>
```

When a job display command (i.e., list) is issued, the total number of queued jobs and the scheduling policy should be displayed prior to the list of jobs. For each job managed in the job queue, the following information should be shown (1) job name, (2) execution time (i.e., CPU_time), (3) priority, (4) arrival time, and (5) status (e.g., Run).

4.4 Switch Scheduling Policy

The scheduling policy can be switched by one of the three commands, namely, "`fcfs`", "`sjf`", and "`priority`". A sample dialog of changing the scheduling policy from FCFS to SJF is depicted below. Please note that the dialogs of the "`fcfs`" and "`priority`" commands are similar to that of the "`sjf`" command.

```
$./aubatch
Welcome to Dr. Qin's batch job scheduler Version 1.0
Type 'help' to find more about AUbatch commands.
>sjf
Scheduling policy is switched to SJF. All the 3 waiting jobs
have been rescheduled.
>fcfs
Scheduling policy is switched to FCFS. All the 2 waiting jobs
have been rescheduled.
>
```

When you switch the scheduling algorithm, AUbatch should following the following two principles:

- The running job can't be preempted.
- All the submitted jobs except the running job must be reordered in the queue according to the new scheduling policy.

4.5 Quit AUbatch

Users can exit AUbatch using the `quit` command. Before exiting AUbatch, performance evaluation information should be displayed. A sample dialog of the `quit` command list submitted jobs is outlined as follows.

```
$./aubatch
Welcome to Dr. Qin's batch job scheduler Version 1.0
Type 'help' to find more about AUbatch commands.
...
>quit
Total number of job submitted: 5
Average turnaround time:  32.12 seconds
Average CPU time:         15.43 seconds
Average waiting time:     16.69 seconds
Throughput:               0.031 No./second
$
```

4.6 Automated Performance Evaluation

Users may conduct automated performance evaluation using the `test` command. The fomat of this command is:

```
test <benchmark> <policy> <num_of_jobs> <priority_levels>
     <min_CPU_time> <max_CPU_time>
```

where `<benchmark>` is the benchmark name, `<policy>` specifies the scheduling policy to be evaluated, `<num_of_jobs>` is the number of issued jobs, `<priority_levels>` is the number of available priority levels, `<min_CPU_time>` and `<max_CPU_time>` indicate the minimum and maximum CPU times randomly generated by the automated performance evaluation module. A sample dialog of the `test` command is depicted below. In this example, three scheduling policies (i.e., FCFS, SJF, and Priority) are evlauted under the same workload condition.

Ideally, we should add an input parameter to specify job arrival rate. The updated format should be:

```
test <benchmark> <policy> <num_of_jobs> <arrival_rate>
<priority_levels> <min_CPU_time> <max_CPU_time>
```

```
$./aubatch
Welcome to Dr. Qin's batch job scheduler Version 1.0
Type 'help' to find more about AUbatch commands.
...
>help -test
test <benchmark> <policy> <num_of_jobs> <priority_levels>
     <min_CPU_time> <max_CPU_time>

>test mybenchmark fcfs 5 3 10 20
Total number of job submitted: 5
Average turnaround time:  32.12 seconds
Average CPU time:         15.43 seconds
Average waiting time:     16.69 seconds
Throughput:               0.031 No./second

>test mybenchmark sjf 5 3 10 20
Total number of job submitted: 5
Average turnaround time:  28.95 seconds
Average CPU time:         15.43 seconds
Average waiting time:     13.52 seconds
Throughput:               0.035 No./second
> test mybenchmark priority 5 3 10 20
Total number of job submitted: 5
... ...
```

5. Project Development

The intent of this section is to demonstrate the systematic use of resources, knowledge and practices to design and implement AUbatch under specific requirements.

5.1 Step 1: Creating two child processes in AUbatch

The first step is understanding how to create two concurrent processes using the `pthread_create()` system call. The first child process handles the scheduling module, whereas the second child process performs the dispatching module. Please refer to [6] for a sample code of `pthread_create()`. You may also recycle your code from project 2.

5.2 Step 2: Running new program by the dispatching module

You should learn how to run programs by the dispatching module of your batch scheduling system. This functionality can be easily implemented by the `execv()` system call. Before implementing the dispatching module, please try `execv()` by writing a simple program. A good sample program can be found in [7].

5.3 Step 3: Tackling synchronization problems using conditional variables

Before proceeding to the next step, you must be familiar with solving synchronization problems using conditional variables and mutexes. In addition, you should write a *PThread* program that implements the conventional producer-consumer problem. Please refer to [5] for the Producer–consumer problem and [1] for the PThread programming. A producer may push a string into an array of strings, whereas a consumer will retrieve a string from the shared array. In a later step (i.e., Step 4), this string should be replaced by a job coupled with a few parameters (e.g., priority and execution time).

5.4 Step 4: Coordinating the scheduling and dispatching modules

Now you should synchronize the behaviors of the scheduling and dispatching modules in AUbatch. You are advised to follow the design pattern of the producer-consumer PThread program developed in Step 3. In this step, jobs are submitted by hard coding rather than user input. You should integrate the outcomes of Steps 2 and 3 together. In this step, you are expected to write a program to achieve the following goal:

- A hard-coded job can be submitted to the job queue by the producer thread (a.k.a., scheduling module). You should refer to Step 3 to achieve this goal.
- The consumer thread (a.k.a., dispatching module) retrieve the job and run the job using execv() (see Step 2). You should refer to Steps 2 and 3 to achieve this goal.

5.5 Step 5: Implementing the user interface

In this step, you are in a position to implement the user interface (i.e., see command line usages specified in Section 4). You are suggested to develop the command line as an independent module, meaning that you shouldn't integrate the user interface with the scheduling module. Such an integration should be achieved in the next step (i.e., Step 6).

5.6 Step 6: Integrating the user interface with the scheduling module

After accomplished step 4, your prototype can submit and run hard-coded jobs through two concurrent and synchronized threads. Step 5 allows you to submit jobs through a command line. The goal of step 6 is to seamlessly integrate the outcomes of Steps 4 and 5.

5.7 Step 7: Implementing the SJB and priority-based algorithms

A default scheduling algorithm implemented in Step 6 is FCFS (i.e., first come, first served). Step 7 is focused on the development of the SJB and priority-based algorithms. You should implement the two algorithms as two separate functions. After passing the unit testing of the algorithms, you may integrate these two algorithms into the

prototype accomplished in Step 6.

5.8 Step 8: Micro Batch-job Benchmarking

In this step, you must implement micro batch-job benchmarks. Please refer to Section 3.2 for instructions on the development of micro benchmarks.

5.9 Step 9: Putting It All Together

Combining Steps 7 and 8, you will be able to submit micro benchmarks in the command line inside AUbatch.

5.10 Step 10: Evaluating Performance

The final step is to conduct extensive experiments. Please refer to Section 3 for the instructions on performance evaluation. The instructions on automated Performance Evaluation can be found in Section 3.4 on page 6.

5.11 Don't Procrastinate

Important! If you desire to be a project 3 survivor, please don't procrastinate on project 3. The estimated number of hours spent on this project is anywhere between 20 to 30 hours, depending your multithreaded programming skills. In the worst case in which you are unfamiliar with PThread and synchronizations, it is likely to consume you at least 10 hours to grasp the basic programming knowledge for project 3. As such, this project isn't the kind of thing you can complete three days before the deadline. You are strongly recommended to embark on this project on the first day when the specification is released.

6. Programming Requirements

6.1 Programming Environment

You must implement your *AUbatch* system in C. Please compile and run your system using the `gcc` compiler on a Linux box (either in Tux machines, computer labs in Shelby, your home Linux machine, a Linux box on a virtual machine, or using an emulator like Cygwin).

6.2 Function-Oriented Approach

You are *strongly suggested* to use a structure-oriented (a.k.a., function-oriented) approach for this project. In other words, you will need to write function definitions and use those functions; you can't just throw everything in the `main()` function. A well-

done implementation will produce a number of robust functions, many of which may be useful for future programs in this project and beyond.

Remember good design practices include:

- A function should do one thing, and do it well
- Functions should NOT be highly coupled

6.3 File Names and Comment Blocks

Important! You will lose points if you do not use the specific program file name, or do not have a comment block on **EVERY** program you hand in.

6.4 Usability Concerns and Error-Checking

You should appropriately prompt your user and assume that they only have basic knowledge of the tool. You should provide enough error-checking that a moderately informed user will not crash your system. This should be discovered through your unit-testing. Your prompts should still inform the user of what is expected of them, even if you have error-checking in place (see an example in Section 4.2).

6.5 Make Your Code Readable

It is very important for you to write well-documented and readable code in this project. The reason for making your code clear and readable is three-fold. First, you should strive allow Dr. Qin to read and understand your code. Second, there is a likelihood that you will read and understand code written by yourselves in the future. Last, but not least, it will be a whole lot easier for the COMP7500/7506 teaching assistant to grade your programming projects if you provide well-commented code.

Since there are a variety of ways to organize and document your code, you are allowed to make use of any particular coding style for this programming project. It is believed that reading other people's code is a way of learning how to write readable code. In particular, reading the source code of some freely available operating system provides a capability for you to learn good coding styles. Importantly, when you write code, please pay attention to comments which are used to explain what is going on in your AUbatch system.

Some general tips for writing good code are summarized as below:

- A little time spent thinking up better names for variables can make debugging a lot easier. Use descriptive names for variables and procedures.
- Group related items together, whether they are variable declarations, lines of code, or functions.
- Watch out for uninitialized variables.

- Split large functions that span multiple pages. Break large functions down! Keep functions simple.
- Always prefer legibility over elegance or conciseness. Note that brevity is often the enemy of legibility.
- Code that is sparsely commented is hard to maintain. Comments should describe the programmer's intent, not the actual mechanics of the code. A comment which says, "Find a free disk block" is much more informative than one that says "Find first non-zero element of array."
- Backing up your code as you work is difficult to remember to do sometimes. As soon as your code works, back it up. You always should be able to revert to working code if you accidentally paint yourself into a corner during a "bad day."

7. Separate Compilation (Required Requirement)

Unlike project 2, this project must be organized and compiled using separate compilation. You are required to write a makefile for project 3. If you are unfamiliar with separate compilation, you should follow the instructions in this Section to carry out separate compilation by creating the makefile for your AUbatch.

7.1 What is Make?

Make is a program that looks for a file called "makefile" or "Makefile", within the makefile are variables and things called dependencies. There are many things you can do with makefiles, if all you've ever done with makefiles is compile C or C++ then you are missing out. Pretty much anything that needs to be compiled (postscript, java, Fortran), can utilize makefiles.

7.2 Format of Makefiles: Variables

First, let's talk about the simpler of the two ideas in makefiles, variables. Variable definitions are in the following format:

```
VARNAME = Value
```

So, let's say I want to use a variable to set what compiler I'm going to use. This is helpful b/c you may want to switch from cc to gcc or to g++. We would have the following line in our makefile

```
CC = gcc
```

This assigns the variable CC to the string "gcc". To expand variables, use the following form:

```
${VARNAME}
```

So to expand our CC variable we would say:

```
${CC}
```

7.3 Format of Makefiles: Dependencies

Dependencies are the heart of makefiles. Without them nothing would work. Dependencies have the following form:

```
dependecy1: dependencyA dependencyB ... dependencyN
      command for dependency1
```

Check out the following links for more information on makefiles:
http://oucsace.cs.ohiou.edu/~bhumphre/makefile.html

8. Project Report

Write a project report that explains (see also Section 9-1 Grading Criteria):

- The design and implementation of your AUbatch.
- Performance metrics and workload conditions.
- The performance evaluation of the three scheduling algorithms.
- Lessons learned.

Important! Your report is worth 30 points. Your project report should provide sample input and output from your implemented program. The sample input/output data should be collected using the `script` utility program (see Project 1).

9. Deliverables

9.1 Final Submission

Your final submission should include:

- Your project report (see also Section 8).
- A copy of the complete source code of your developed AUbatch system.
- A makefile that will run and compile your AUbatch

Important! You must submit a single compressed file (see also Section 9.2) as a `.tgz` file, which includes both a project report and source code.

9.2 A Single Compressed File

Please submit your tarred and compressed file named `proejct3.tgz` through Canvas. You must submit your single compressed file through Canvas. No e-mail submission is accepted.

9.3 What happens if you can't complete the project?

If you are unable to complete this project for any reason, please describe in your report the work that remains to be finished. It is important to present an honest assessment of any incomplete components.

10. Project Assessment

10.1 Grading Criteria

The approximate marks allocation will be:

1)	Project Report:			30%
	1.1)	Design Document	10%	
	1.2)	Performance Metrics and Workload	5%	
	1.3)	Performance Comparison	10%	
	1.4)	Lessons learned	5%	
2)	Implementation (i.e., Source Code):			50%
3)	Separate Compilation and Makefile:			10%
4)	Clarity and attention to details:			10%
	Total (Items 1-4):			**100%**

10.2 Late Submission Penalty

Ten percent (10%) penalty per day for late submission. For example, an assignment submitted after the deadline but up to 1 day (24 hours) late can achieve a maximum of 90% of points allocated for the assignment. An assignment submitted after the deadline but up to 2 days (48 hours) late can achieve a maximum of 80% of points allocated for the assignment.

Important! Project assignments submitted more than 3 days (i.e., 72 hours) after the deadline will not be graded. In this case, the grade of your project 3 will be 0.

10.3 Rebuttal Period

You will be given a period a week (7 days) to read and respond to the comments and grades of your homework or project assignment. The TA and Dr. Qin may use this opportunity to address any concern and question you have. The TA and Dr. Qin also may ask for additional information from you regarding your project.

References

[1] POSIX Threads Programming. https://computing.llnl.gov/tutorials/pthreads/

[2] An example using fork, execv and wait. http://www.cs.ecu.edu/karl/4630/spr01/example1.html

[3] Operating System Scheduling algorithms. https://www.tutorialspoint.com/operating_system/os_process_scheduling_algorithms.htm

[4] Data Flow Diagram. https://www.smartdraw.com/data-flow-diagram/

[5] Producer–consumer problem. https://en.wikipedia.org/wiki/Producer–consumer_problem

[6] Using pthread_create – a minimum working example
http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example/
[7] execv -- Overlay Calling Process and Run New Program
https://support.sas.com/documentation/onlinedoc/sasc/doc/lr2/execv.htm