COMP 7500 Advanced Operating Systems

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

Frequently Asked Questions

REVISED: March 17, 2025
VERSION 3.7

1. Should we use fork() or pthread_create() to create two threads?

Answer: Please use pthread_create() to create two threads (i.e., the scheduler and disptacher). Please check the sample code here:
http://timmurphy.org/2010/05/04/pthreads-in-c-a-minimal-working-example

2. Are we executing actual programs (e.g., hello_world) or simulating them?

Answer: You must actually execute the programs (a.k.a., benchmarks).

3. Do we ignore the actual execution time and use the execution time given by the run command?

Answer: The execution times should be measured on the fly based on actual execution times rather than the estimated ones submitted by users.

4. If you add a job with a higher priority and change the policy to priority, the running job will still remain the same, right?

Answer: Once your change your policy, all the jobs waiting in the queue must be rescheduled based on the new policy. The only job that should be affected by the new policy is the current running job (i.e., the one sitting in the head of the job queue).

5. Can I use any 3rd party class library?

Answer: You don't have to develop all the functions from the ground up. Please feel free to use any third-party libraries dealing with basic functions like data/times, queues, lists, file I/Os. It is important to give credits to the original authors of these libraries.

6. Why users must specify a job's CPU time priori to the job submission?

Answer: All the existing batching systems require users to provide estimated the CPU time for each submitted job. Schedulers will make scheduling decisions based on required amount of resources (e.g., CPU time, memory). Estimated CPU times should be close to the real measured ones. Prior to job submissions, users (i.e., you in this project) must estimate the CPU times of jobs to be submitted to AUbatch.

7. Should "quit" exit immediately or after all jobs have completed?

Answer: Ideally, you should choose to let the parent process wait for the dispatcher thread to finish executing all the pending jobs. A simple solution is "immediate termination" using the exit system call, which is demonstrated in the sample source code - commandline_parser.c

8. Do we include the running job's expected execution time to compute expected waiting times?

Answer: Expected waiting times are derived from the running job's remaining time, which depends on its expected execution time. Because "list" can occur while a job is running, you must calculate the running job's remaining time to estimate the expected waiting times of all the other waiting jobs. After computing the expected time left (i.e., remaining_time) for the current running job, all the expected wait time for pending jobs can be easily evaluated. The remaining_time of the running job can be expressed as:

```
remaining_time = expected_exe_time - (current_time - start_time);
```

9. Can I test my program using built-in test command?

Answer: Yes. You may implement your performance evaluation module using the test command. Please feel free to change the command format when it comes to performance evaluation. You are encouraged to propose an improved user interface with respect to performance evaluation.

10. **Test Command.** For the Test command in Project 3, after entering the **test** command, should the user be able to interact with the program while the jobs are executing? Or, should the test run until it is complete, and then print out the report for the test? (Spring'20)

Answer: '`test`' is a command furnishing an automatic performance testing process; therefore, users don't interact with the AUBatch while jobs (a.k.a., microbenchmarks) are executing. All the jobs should be batching jobs rather than interactive jobs.

Users may issue "quit -i" to immediately terminate testing. Alternatively, the users may type "quit -d" to exit AUBatch after all the submitted jobs are completed.

11. **Test when the initial queue is non-empty.** If several jobs are already in the queue, should we still be able to run the test function knowing that the queue is not empty? (Spring'20)

Answer: When we issue the test function, we assume that the queue is initialized to be empty. If the queue is non-empty prior to running the test function, your system should report an error.

12. **Priority Settings.** In the test module do we assign priorities randomly to the jobs (if yes what distribution we use for randomly generating the priorities) or they all have to carry the same priorities? (Spring'20)

Answer: You may choose to assign priorities to the jobs or randomly assign priority levels based on a uniform distribution in a range.

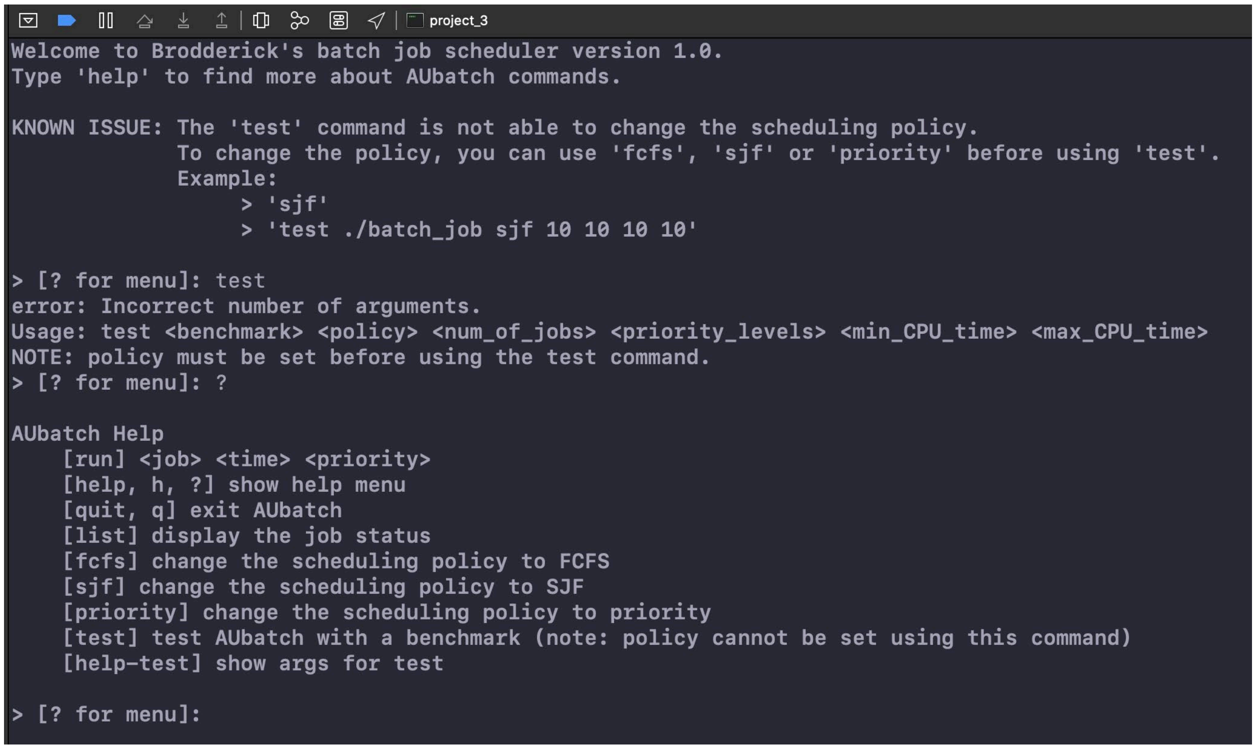
13. **Micro Benchmarks.** Do these micro-benchmarks actually have to use the CPU (by doing garbage computations) or can a call to "sleep" or "usleep" be used to wait the appropriate amount of time? (Spring'20)

Answer: These micro benchmarks should consume CPU utilization (e.g., dealing with garbage computations). Please don't use "sleep" or "usleep" in the micro benchmarks.

14. **Slight UI change?** I know the assignment document says not to deviate from the user interface shown. However, I am having an issue where the 'test' command cannot change the policy. To get around this, you can first change the policy using 'fcfs', 'sjf', or 'priority', then call 'test'. I cannot figure out why this is happening. The project is complete besides this one issue. I plan on submitting it as is and document the issue.

    I would like to add:
    (1) a "known issue" statement to the "welcome" printout.
    (2) add a note to the "correct use" of 'test'.
    (3) add a note to the "help" menu.

    Please let me know if this is OK. See attached screenshot. (Spring'20)



```
Welcome to Brodderick's batch job scheduler version 1.0.
Type 'help' to find more about AUbatch commands.

KNOWN ISSUE: The 'test' command is not able to change the scheduling policy.
             To change the policy, you can use 'fcfs', 'sjf' or 'priority' before using 'test'.
             Example:
                  > 'sjf'
                  > 'test ./batch_job sjf 10 10 10 10'

> [? for menu]: test
error: Incorrect number of arguments.
Usage: test <benchmark> <policy> <num_of_jobs> <priority_levels> <min_CPU_time> <max_CPU_time>
NOTE: policy must be set before using the test command.
> [? for menu]: ?

AUbatch Help
    [run] <job> <time> <priority>
    [help, h, ?] show help menu
    [quit, q] exit AUbatch
    [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] test AUbatch with a benchmark (note: policy cannot be set using this command)
    [help-test] show args for test

> [? for menu]:
```

Answer: Your proposed user interface was well designed and; therefore, I endorse the new user interface developed by you.

15. **Extra Work.** If we wanted to implement extra metrics and also print each metric per job in addition to overall metrics when quitting would it be permissible? Additionally I have it right now so when you ask to list the jobs it also lists the jobs that have finished, just to show the user a complete view of jobs running, jobs waiting and jobs that have finished. I just wanted to know if it was okay to do the extra work, as I do not want to get docked for not having my output exactly like the specifications show? (Spring'20)

Answer: You are encouraged to do the extra work. It is a brilliant idea to list finished jobs in addition to pending jobs.

16. **Output.** Is it safe to assume that our project4 output should match *exactly* to the provided sampleoutput.txt provided? (Spring'24)

Answer: You should try your best to match your output to the provided sample output. If there are minor differences, it should not be a big issue.

17. **File Names.** If have three files, aubatch.c (contain main and sets up the threads and all the condition variables), modules.c (contains the scheduler and dispatcher modules), and

commandline.c (contains code that interacts with the users). Additionally, I have a header file for both commandline.c (commandline.h) and for modules.c (modules.h) to allow aubatch.c to call them. There is a note in the specifications that says the files must adhere to the file names specified. Is it fine if I have these files? (Spring'20)

Answer: These three source code files look good to me. I recommend you to change modules.c into scheduler.c, because modules.c sounds too generic.

18. **Waiting time.** After the run command is given and the job is inserted, we need to display the total waiting time. Should this be the sum of the execution time of jobs waiting in the queue only or the remaining execution time of the currently running process also needs to be added? (Spring'20)

Answer: A simple way of computing the expected waiting time of a newly submitted job is the summation of the execution time of pending jobs that are scheduled ahead of this submitted job. It might be hard to calculate the remaining execution time of the currently running job, so you may count the entire execution time of the running job toward the waiting time of the submitted job.

19. **Commandline Parser and Scheduler Threads.** My understanding is that we have two threads - dispatcher thread and the scheduler thread. Dispatcher thread will take the jobs from queue (bufTail) and send it to execv/system. And the scheduler thread will be continuously checking for user input and working accordingly. Now here is my question. A new job arrives and we call `run` command when our job_queue is full, we would use `cv_wait' on the scheduler thread (as in sample code). I.e. our scheduler thread is sleeping and user can no longer enter any more commands (`list`, `help` etc). Is this how it is supposed to be? (Spring'20)

Answer: If the buffer is full, the scheduler will sleep. If the scheduler and commander line parser are embedded within the same thread, you will be unable to enter another command until your scheduler is waked up by the executor.

20. **Workload Parameters.** What do load distribution and arrival rate mean? (Spring'21)

Answer: Let's give examples to explain the definitions of load distribution and arrival rate. If load distribution is [1, 10], you randomly generate CPU time between 1 second to 10 seconds. If the arrival rate is 0.1 No./second, you submit a new program to AUBatch once every 10 seconds. The table in the project specification embraces the sample workload configuration. Please feel free to modify the workload parameters.

21. **Job Queue.** Can our job queue be an array (of jobs) of max size of 50? Or do we have to manually implement something like a linked list? (Spring'21)

    Answer: If you are unfamiliar with the singly-linked list implementation in C, the best data structure for you to implement the job queue will be a static array. A circular array is easier to debug than the linked-list counterpart.

22. **Run vs Scheduler.** What's the relationship between the "run" command function and the "scheduler" thread? (Spring'21)

    Answer: The "run" command should be implemented by the "scheduler" thread. When the run command is issued, the scheduler thread will insert the newly added job into the appropriate place in the job scheduling queue. In other words, the run command passes a new job data to the scheduler, which can add the new job to the queue based on the scheduling policy.

23. **A synchronization problem.** I had everything in place expect that when the user enters a run command it will not put the job in queue if a previous job is in the executor thread. I was super confused as I'm using 2 separate threads, one for submitter and the other for the executor. How to solve this problem? (Spring'22)

    Answer: I suggest that you should check if you are holding on to the lock too long in the executor method's, meaning that your critical section is spread apart more than it should. For example, if you put the unclock() behind wait(NULL), your scheduler thread may be blocked by the executor thread.

    Here is the basic idea about how to create a small critical section. Before the second executor thread removes a job from the head of the job queue, you should use the lock() operator to prevent the scheduler thread from accessing the shared queue. After the job was removed from the queue, the executing thread MUST immediately release the lock to keep a short critical section. Otherwise, the scheduler thread can't access the job queue for a very long time period.

24. **Run - Job Input.** Can a user input a job that requires user command line input? Or do we assume every job they submit requires no input from the user? If so, how do we account for a user input with run? (Spring'22)

    Answer: You can choice your own benchmark design. To simply the implementation, your benchmark may take no input at all. You may, nevertheless, experiment a case where benchmark jobs take input parameters. If this is your option, you will have to update the "test" command format stipulated in the project description. A sample format looks like:

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

25. **Project Report - Performance Metrics and Workload Conditions**. What are you looking for in the project report for the performance metrics and workload conditions? Are you just wanting to see sample input and output from the test function with multiple different tests being ran? Will we use the same tests in that section in the performance evaluation section? (Spring'22)

    Answer: The Metrics should include throughput, the mean of response time/ turnaround time/ CPU time/ waiting time, and the GTA needs to check your output with respect to these metrics. Apart from the test stipulated in the performance evaluation section, you are free to design your own test workloads.

26. **Benchmark doesn't exist.** Let's say I enter a run into aubatch with it saying:

    ```
    run my_job1 8 6
    ```

    The program passes the name "my_job1" to the execv and if that file exists in the directory, you run that file process if it is compiled. What should the program do if my_job1 doesn't exist?

    Answer: You should implement an error-checking function to verify if my_job1 exists in the directory. If the answer is no, your program should return an error message to users.

    Before running "my_job1" using execv, your program checks if "my_job1" exists or not. A sample code that checks the availability of a file is given below, where fname is set to "my_job1".

    ```
    if (access(fname, F_OK) == 0) {
        // file exists
    } else {
        // file doesn't exist
    }
    ```

    Reference: https://stackoverflow.com/questions/230062/whats-the-best-way-to-check-if-a-file-exists-in-c

27. **Priority Scheduling**. In Priority Scheduling, if a new process has the same priority as existing processes in the queue, should it be placed at the end of the queue? (Spring'24)

Answer: If two processes are sharing the same priority, you may use their arrival times to break the tie -- an earlier arrival should be executed first. In this scenario, this strategy falls back to the FCFS policy for a group of processes with the equal priority.

28. **Test Benchmark**. Am I supposed to create real jobs that I add to the queue and actually let run to get the statistics, or is it supposed to be simulated and be able to complete super fast? If it is real jobs, I assume we want a test function to have the mutex the entire time so nothing can interrupt it? (Spring'24)

    Answer: The statistic measures should be based on the actual running of submitted jobs: you should not use any simulated run time to complete this project. You don't need the mutex to stop the running job from being interrupted. Rather, you may make use of the `system()` function to invoke an executable file, and the dispatcher will await the completion of the invoked executable program before dispatching the next job in the queue. Below is a sample of using the system() function.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // for sleep()

int main() {
    printf("Master process started\n");

    /*
     * Run another program (replace "program_name" with the actual
     * executable file name)
     */
    system("./program_name");

    printf("Master process resumed after child program completion\n");

    return 0;
}
```

    Ideally, you should rely on the execv() system call rather than the system() function to implement your dispatcher, which will launch the running job by creating a process. You may follow the sample code below to implement your dispatcher.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    printf("Master process started\n");

    pid_t pid = fork(); // Fork a new process
    if (pid == -1) {
        perror("fork");
        return 1;
    } else if (pid == 0) { // Child process
        char *args[] = {"./program_name", NULL}; // Arguments for the
executable
        execv(args[0], args); // Execute the program
        perror("execv"); // execv returns only on error
        exit(1);
    } else { // Parent process
        int status;
        waitpid(pid, &status, 0); // Wait for the child process to finish
        printf("Master process resumed after child program completion\n");
    }

    return 0;
}
```

29. **Header Files**. Do we need to make an AUbatch.h file for our AUbatch.c file to access the global variables from? And the other files included as well, like cmd_parser.c and cmd_parser.h? (Spring'24)

    Answer: The data structures and function prototypes of aubatch.c should be listed in aubatch.h. Similarly, the data structures and function prototypes of the command-line parser module should be specified in cmd_parser.h. You may place all the global variables in aubatch.h

30. **Running Job**. Is the running job the one in the head of the queue? Or is the running job the one that has already been taken away by the dispatcher and put on the CPU? (Spring'25)

    Answer: There are two approaches. In the first approach, a job can be removed from the scheduling queue after the job is dispatched. Alternatively, if you would like to implement a function to check which job is a currently running, you need to keep it in the head of the queue until the job is finished.

31. **Preemption or Non-preemption**. Is there a case when this running job would get preempted? (Spring'25)

    Answer: To simplify your implementation, we assume that AUbatch is a non-preemptive scheduling system where no preemption is allowed.

32. **Benchmark Implementation.** How do I ensure that job1 runs for 3 secs without using the sleep() system call? (Spring'25)

    Answer: In the following sample code, a busy wait loop continuously checks the current time until 3 seconds have elapsed. You may, of course, implement this requirement using other ideas.

```
#include <stdio.h>
```

```
#include <time.h>

void job1() {
    printf("Job1 starting...\n");
    // Your job1 code here
    printf("Job1 completing...\n");
}

int main() {
    clock_t start_time, current_time;
    double elapsed_seconds;

    // Get starting time
    start_time = clock();

    // Run job1
    job1();

    // Busy wait until 3 seconds have passed
    do {
        current_time = clock();
        elapsed_seconds = (double)(current_time - start_time) /
                           CLOCKS_PER_SEC;
    } while (elapsed_seconds < 3.0);

    printf("3 seconds elapsed, job1 complete\n");
    return 0;
}
```

33. **Endless Loop Problem.** Let me explain how I'm trying to get it to work: Program starts and user enter a job or jobs in this format <jobname> <execution time><priority>

- Jobs are saved to a submit queue.
- user chooses which scheduling policy to apply
- jobs are moved from submit queue to scheduled queue
  scheduled queue performs execution based on scheduling policy selected
  performance is printed
  User is asked rerun with different scheduling policy (yes/no)
- If yes, scheduled queue reruns jobs using new selected scheduling policy performance is printed.
- If no jobs are sent back to submit queue to add more jobs
- My problem is that the jobs are looping in the scheduling queue and not prompting the rerun question.

How do I get my program to recognize that the jobs have completed in the scheduling queue and wait on Boolean input from the user before running again?

Answer: Here is a shortcut solution to your problem.

- You should estimate the CPU time of your jobs. Let's assume the CPU time is 5 seconds.

- Once your jobs are being dispatched, your program will automatically "recognize" the completion of these jobs as their finish time is predictable -- 5 seconds after the start time.

34. **How to Pass Multiple Arguments into a Function as a Thread**. Does the `pthread_create()` function allows multiple arguments into function it uses? If so, then how can it be done? (Spring'25)

    Answer: Please follow the example below, in which there are four arguments to be passed to the thread function, to implement your PThread code.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Structure to hold multiple arguments
typedef struct {
    int job_id;
    int execution_time;
    char* job_name;
    int priority;
    // Add any other arguments you need
} ThreadArgs;

// Thread function
void* scheduler_thread(void* arg) {
    // Cast the void pointer back to our structure type
    ThreadArgs* args = (ThreadArgs*)arg;

    // Now we can access all the arguments
    printf("Thread started for job: %s\n", args->job_name);
    printf("Job ID: %d\n", args->job_id);
    printf("Execution time: %d\n", args->execution_time);
    printf("Priority: %d\n", args->priority);

    // Do the actual work here

    // Free the memory if we allocated it
    // (Be careful about memory management across threads)

    pthread_exit(NULL);
}

int main() {
    pthread_t thread;

    // Allocate and initialize the arguments structure
    ThreadArgs* args = malloc(sizeof(ThreadArgs));
    args->job_id = 123;
    args->execution_time = 500;
    args->job_name = "BatchJob1";
    args->priority = 2;
    // Create the thread and pass the structure pointer as the argument
    int result = pthread_create(&thread, NULL, scheduler_thread,
(void*)args);

    if (result != 0) {
        printf("Thread creation failed: %d\n", result);
        return 1;
    }

    // Wait for the thread to finish
    pthread_join(thread, NULL);

    // Clean up
    free(args);

    return 0;
}
```

35. **Arrival Rate**. The example usage statement in Section 4.1 does not include <arrival rate> as a parameter for the test command. However, the body of Section 4.6 talks about it as being a parameter but the usage example in the same section excludes it. Should <arrival rate> be included as a test parameter? (Spring'25)

    Answer: It depends on how you want to implement the project.
    - Option 1: If you pass the arrival rate to AUBatch and let it to automatically create jobs, then you must include <arrival_rate> as an input parameter.
    - Option 2: If you implement a module creating all jobs according to the arrival rate outside AUBatch -- and your job creator is an external entity of AUBatch, then <arrival_rate> should be excluded from AUBatch's parameter list.

36. **Insert a Job into the Queue**. I implement the queue as an array of jobs, which are structures. How to write a function to add a job into the queue? (Spring'25)

    Answer: Please review and follow the sample code below.

```
typedef struct {
    int id;
    char description[100];
    int priority;
    // other job fields as needed
} Job;

#define MAX_QUEUE_SIZE 100
Job jobQueue[MAX_QUEUE_SIZE];
int queueSize = 0;  // Current number of jobs in queue

/* Function to add a job to the queue */
int addJobToQueue(Job newJob) {
    // Check if queue is full
    if (queueSize >= MAX_QUEUE_SIZE) {
        return -1;  // Queue is full, cannot add more jobs
    }

    // Add the job to the end of the queue
    jobQueue[queueSize] = newJob;
    queueSize++;

    return 0;  // Success
}
```

This is a simple implementation where jobs are added to the end of the queue. The function:

- Checks if there's space available in the queue
- If space is available, adds the job to the end of the queue
- Increments the queue size counter
- Returns 0 for success or -1 if the queue is full