CS39002 Operating Systems Laboratory
Spring 2026

Lab Assignment: 7
Date posted: 13-Mar-2026

Programming Using the Pthreads API

FooBar Wagen GmbH manufactures two types of cars: a luxury car called foocar, and a sports car called barcar. The company receives orders for f foocar's and b barcar's. The production factory can build only one car at a time, so the productions of the two types of cars are alternately carried out. When all productions of one type finish, the productions of the remaining type are done one after another. For example, if $f = 4$ and $b = 2$, then the car production will run in the sequence f, b, f, b, f, f.

The production of each type of car is broken down into parts (or modules). The parts are numbered 0, 1, 2, … , $N-1$. Certain parts cannot be prepared if some other parts are not completed. The dependency of the parts for each type of car is specified in a text file. If part q depends on part p (that is, if p is a prerequisite for q), then the part numbering satisfies $p < q$. To sum up, the dependency relationship is a DAG (directed acyclic graph) with one topological ordering given by $0 < 1 < 2 < \ldots < N-1$.

The production of each type of car is carried out by M workers (mechanics) numbered 0, 1, 2, … , $M-1$. The workers are specialized in doing certain parts of the car. So the company management assigns one particular worker to each part. The list of parts assigned to a given worker is called the to-do list for that worker. The list is sorted by the part numbers. Each worker carries out the work in its to-do list in the given sorted sequence. However, because of dependency, a worker may have to wait for the completion of other parts, before starting work on the next part in the to-do list. The working of the worker w is summarized in the pseudocode below.

```
Worker ( w )
{
    While (there is/are more car(s) to produce) {
        For each part p in the to-do list of w (in the sorted order, for that type of car), do {
            Check if all the prerequisites for p are already completed.
            If not, wait until all the prerequisites of p are completed.
            Complete work on part p, and mark p as done.
            For each part q which depends on p (that is, for each edge p → q in the dependency graph), do {
                If not all prerequisites of q are done, do nothing.
                Otherwise, do {
                    Let v be the worker to which part q is assigned.
                    If v is currently waiting for starting part q, wake v up.
                }
            }
        }
    }
}
```

The topological ordering of the dependency graph guarantees that the production of each car eventually terminates. If the values of M for the two types of cars are different, take the larger of the two values as M. If a worker number w is not applicable for one type of car, assume that no parts are assigned to w for that type of car.

Implementation details

Use the POSIX threads (pthreads) API to implement the work of the company. The main thread works as the manager. It creates M worker threads, each implementing the work of a worker as specified above. The manager schedules the manufacturing of each car as per the order it received from the customer(s). After that, the workers work together to run the production to the end. When all the cars in the reveived order are produced, all the worker threads and the main thread exit.

The main threads starts by reading the two text files storing the specifications of the two types of cars. For each type, the corresponding file stores the following information: (1) the number of parts, (2) the number of workers, (3) the dependency links, and (4) the assignment of workers to the parts. The format of the text file is illustrated in the Sample Output at the end.

The main thread additionally prepares the prerequisite graph (the graph obtained by reversing the edges of the dependency graph). It also prepares the (sorted) to-do lists for the workers (some to-do lists may be empty). These data reside in global data structures (implement adjacency lists in arrays or linked lists). In addition, the main thread prepares other global arrays. An array PSTAT stores the status of each part (PENDING or DONE). Likewise, the status of each worker (START, WORKING, WAITING, DONE) is stored in another global array WSTAT. Finally, if the status of a worker is WAITING (or WORKING), the assigned part number which the worker is waiting for (or working on) should be stored in a global array WINFO.

After this initial bookkeeping, the main thread creates the M worker threads, and pass them their respective IDs (in the range $0, 1, 2, \ldots, M-1$) as the parameter of the thread's main function.

Synchronization among the threads

The entire production work (on each car) is carried out cooperatively by the worker threads following any sequence that obeys the dependency (or prerequisite) requirements. The main thread only plays the role of starting and finishing the work on each car, and to signal all-productions-done at the end. Use *only* the following synchronization primitives. Use of artificial delays (`sleep`/`usleep`) for achieving synchronization is strictly prohibited.

- **Barriers:** Two barriers bop (begin of production) and eop (end of production) are used by all the threads. The global memory stores data for both types of car. Before starting the work on each car, the main thread prepares the information on which set of data to use. It also initializes the PSTAT (all PENDING) and the WSTAT (all START) arrays. It is not needed to initialize the WINFO array. After this initialization is done, all the $M+1$ threads (the main thread and the M worker threads) synchronize using the barrier bop.

  When all workers are done on the production of the current car, the $M+1$ threads again synchronize using the barrier eop. After this synchronization, the main thread can start the production of the next car (or notify end-of-production). Note that the cars are produced one by one.
- **Mutex:** A single mutex mtx is to be used for the mutual exclusion of all shared data, and for use with all condition variables explained below. The data on the cars (dependency, prerequisites, to-do lists) are read-only, and do not need mutual exclusion. However, the other global arrays (PSTAT, WSTAT, and WINFO) allow both reading and writing. Therefore any read or write on these arrays must be protected by mtx.
- **Condition variables:** For each worker, a condition variable cnd is to be used. Indeed, cnd[w] is the waiting place for the worker w (when there are PENDING prerequisites of the next part in the to-do list of w). Let p be the next part for worker w. If all the prerequisites of p are already DONE, the worker does not need to wait, and finish part p immediately. If not, it waits on cnd[w]. Later on, another worker thread v which finishes the *last* PENDING prerequisite for p wakes up w by sending a signal to cnd[w]. Use the same mutex mtx (described above) in conjunction with each cnd[w].

The main thread creates all these synchronization resources before creating the worker threads. It also initializes the barriers bop and eop (to $M+1$ each), mtx to unlocked, and each cnd[w] to the default initializer. At the end, these may be destroyed by the main thread.

The main thread should wait for the termination of all the worker threads (use `pthread_join` and `pthread_exit` appropriately; the worker threads must be JOINABLE).

Write the working of the main thread in `manager.c`(`pp`), and the working of each worker thread in `work.c`(`pp`). You may also declare all the global variables in `global.c`(`pp`). The outermost wrapper (with a `main` function) may be `foobar.c`(`pp`) which uses suitable `#include` directives to include the other files. Also write a *makefile* with compile, run, and clean targets.

Submit a single zip/tar/tgz archive storing all the source files and the makefile.

Sample Output

You can generate sample inputs by running the program given as *genschedule.c*. It takes the following optional command-line arguments: N_1, N_2 (the numbers of parts of a **foocar** and of a **barcar**), M_1, M_2 (the numbers of workers needed for a **foocar** and for a **barcar**), and the two file names storing dependency and worker-assignment information for a **foocar** and for a **barcar**. The default values are respectively 32, 40, 8, 10, *fooschedule.txt*, and *barschedule.txt*.

```
$ make genschedule
gcc -Wall -o genschedule genschedule.c
$ ./genschedule 16 20 3 5 demofoo.txt demobar.txt
```

On a sample run, the two files generated are as follows.

demofoo.txt

```
$ cat demofoo.txt
16 3
 0  1  2  3  4 12 -1
 1  1  2  3  4 -1
 2  2  9 11 12 14 -1
 3  1  7 11 13 -1
 4  0 12 14 -1
 5  2 13 -1
 6  0 10 13 15 -1
 7  2  8 10 12 -1
 8  2 12 -1
 9  1 11 13 15 -1
10  1 11 14 -1
11  1 -1
12  0 15 -1
13  1 -1
14  2 15 -1
15  1 -1
```

demobar.txt

```
$ cat demobar.txt
20 5
 0  2  3 13 17 -1
 1  0  3 17 19 -1
 2  1  6 10 15 18 -1
 3  4  5 10 13 14 17 19 -1
 4  3  7 12 14 -1
 5  1 11 18 -1
 6  4 10 11 15 17 -1
 7  2  8  9 14 -1
 8  4  9 13 16 -1
 9  0 14 -1
10  1 11 14 -1
11  4 -1
12  1 19 -1
13  0 14 18 -1
14  3 15 -1
15  4 17 19 -1
16  0 18 -1
17  1 18 19 -1
18  3 -1
19  3 -1
```

The main thread reads these two files, prepares the necessary lists (the dependency and the prerequisites tables, and the workers' to-do lists), and prints these at the beginning.

```
$ make rundemo
gcc -Wall -o manufacture manufacture.c -pthread
./manufacture 2 1 demofoo.txt demobar.txt

+++ Foocar

    Dependencies
     0 ->  2  3  4 12
     1 ->  2  3  4
     2 ->  9 11 12 14
     3 ->  7 11 13
     4 -> 12 14
     5 -> 13
     6 -> 10 13 15
     7 ->  8 10 12
     8 -> 12
     9 -> 11 13 15
    10 -> 11 14
    11 ->
    12 -> 15
    13 ->
    14 -> 15
    15 ->

    Prerequisites
     0 <-
     1 <-
     2 <-  0  1
     3 <-  0  1
     4 <-  0  1
     5 <-
     6 <-
     7 <-  3
     8 <-  7
     9 <-  2
    10 <-  6  7
    11 <-  2  3  9 10
    12 <-  0  2  4  7  8
    13 <-  3  5  6  9
    14 <-  2  4 10
    15 <-  6  9 12 14

    Worker assignment
     0 :  4  6 12
     1 :  0  1  3  9 10 11 13 15
     2 :  2  5  7  8 14
```

```
+++ Barcar

    Dependencies
     0 ->  3 13 17
     1 ->  3 17 19
     2 ->  6 10 15 18
     3 ->  5 10 13 14 17 19
     4 ->  7 12 14
     5 -> 11 18
     6 -> 10 11 15 17
     7 ->  8  9 14
     8 ->  9 13 16
     9 -> 14
    10 -> 11 14
    11 ->
    12 -> 19
    13 -> 14 18
    14 -> 15
    15 -> 17 19
    16 -> 18
    17 -> 18 19
    18 ->
    19 ->

    Prerequisites
     0 <-
     1 <-
     2 <-
     3 <-  0  1
     4 <-
     5 <-  3
     6 <-  2
     7 <-  4
     8 <-  7
     9 <-  7  8
    10 <-  2  3  6
    11 <-  5  6 10
    12 <-  4
    13 <-  0  3  8
    14 <-  3  4  7  9 10 13
    15 <-  2  6 14
    16 <-  8
    17 <-  0  1  3  6 15
    18 <-  2  5 13 16 17
    19 <-  1  3 12 15 17

    Worker assignment
     0 :  1  9 13 16
     1 :  2  5 10 12 17
     2 :  0  7
     3 :  4 14 18 19
     4 :  3  6  8 11 15
```

Then the worker threads are launched, and the order of 2 foocar's and 1 barcar is carried out as follows. Note that the sequence of manufacturing is f, b, f. The sequences of events during the production of the two foocar's are different. This does not matter so long as the sequence is consistent with the dependency and worker-assignment constraints.

```
+++ Production of a foocar begins

    WORKER 0  WORKER 1  WORKER 2  WORKER 3  WORKER 4
    --------  --------  --------  --------  --------
                                  All done
    Wait 4
                                            All done
                        Wait 2
              Part 0
    Wake up   Part 1    Wake up
              Part 3
              Wait 9
    Part 4
    Part 6
    Wait 12
              Wake up   Part 2
                        Part 5
              Part 9
              Wait 10
              Wake up   Part 7
              Part 10
              Part 11
              Part 13
              Wait 15
    Wake up             Part 8
                        Part 14
                        All done
    Part 12   Wake up
    All done
              Part 15
              All done

+++ Production of a barcar begins

    WORKER 0  WORKER 1  WORKER 2  WORKER 3  WORKER 4
    --------  --------  --------  --------  --------
                                  Part 4
                                  Wait 14
                        Part 0
                        Part 7
                        All done
              Part 2
              Wait 5
                                            Wait 3
    Part 1                                  Wake up
              Wake up                       Part 3
                                            Part 6
                                            Part 8
              Part 5
    Part 9
    Part 13
              Part 10             Wake up
                                            Part 11
                                            Wait 15
              Part 12
    Part 16
    All done
              Wait 17
                                  Part 14   Wake up
                                  Wait 18
              Wake up                       Part 15
                                            All done
              Part 17             Wake up
              All done
                                  Part 18
                                  Part 19
                                  All done
```

```
+++ Production of a foocar begins

    WORKER 0  WORKER 1  WORKER 2  WORKER 3  WORKER 4
    --------  --------  --------  --------  --------
                        Wait 2
                                            All done
              Part 0
              Part 1    Wake up
              Part 3
              Wait 9
              Wake up   Part 2
                        Part 5
                        Part 7
                        Part 8
              Part 9
              Wait 10
                        Wait 14
                                  All done
    Part 4
    Part 6    Wake up
    Part 12
    All done
              Part 10   Wake up
                        Part 14
                        All done
              Part 11
              Part 13
              Part 15
              All done

+++ All productions completed

    WORKER 0  WORKER 1  WORKER 2  WORKER 3  WORKER 4
    --------  --------  --------  --------  --------
                                  Quit
                                            Quit
                        Quit
    Quit
              Quit
```

The transcripts are small enough, and you do not need a delayed simulation to see the progress. Do not use any `sleep` or `usleep` (or any such delay function) anywhere in your code. Use the compact format of the work done by the workers, as given above. This is easier for your evaluators to visualize (than a long list of one notification per line).