CS39002 Operating Systems Laboratory
Spring 2026

Lab Assignment: 8
Date posted: 20-Mar-2026

Simulation of Deadlock Detection and Recovery in a Multi-Threaded Application

In this assignment, you develop a **pthread-based** application where there are M resource types each with multiple instances, and N threads keep on making allocation and release requests. If an allocation request can be granted from the AVAILABLE pool, the requesting thread continues (with the additionally allocated resources). However, if the AVAILABLE pool cannot satisfy an allocation request, the requesting thread has to wait. Later, when resources are available, a waiting thread is granted its pending request.

Since no efforts are made to prevent or avoid deadlocks, the threads may end up in a deadlock situation. A separate deadlock-handler thread periodically wakes up and checks whether the system is in a deadlock. If a deadlock situation is detected, the deadlock handler preempts one or more threads one by one until there is no deadlock (we assume that the resources are such that allocated resources can be preempted and returned to the AVAILABLE pool with impunity).

Each worker thread makes a total of a predetermined number R of allocation/release requests. Since all worker threads can continue after recovery from deadlock, each worker thread eventually makes R allocation/release requests. After this, the thread makes a final release request (release of all resources held at that point), and exits. When all the worker threads exit, the deadlock-handler thread and subsequently the master thread exit too.

The threads

This assignment deals with the following threads (of the same process).

- **Manager thread:** This is the main or the master thread. We call it the (resource) manager because resource-allocation and resource-release requests are handled by this thread. This thread also does the initial bookkeeping: (i) generate the **number of instances of each resource type** (**random in [8, 32]**), (ii) initialize all global variables, (iii) initialize the mutexes and condition variables to be used later, and (iv) create the other threads. After this, the manager goes to a loop waiting for handling the next resource-related request from worker threads.

  Whenever a worker thread wants to make an allocation or a release request, it wakes up the manager thread. If it is a release request, the released resources are added to the AVAILABLE pool. If it is an allocation request, the master thread checks whether that request can be immediately granted. A request is granted either completely or not at all, that is, even if some components of a request can be allocated, the allocation request is not successful if some other components cannot be allocated at this moment. If all components in the allocation request can be allocated to the requesting thread, the allocation is made. Otherwise, the requesting thread has to wait until in future the AVAILABLE pool becomes large enough to accommodate all the components in the request. The manager thread maintains a queue RQ of the waiting threads. After a release request, the manager visits the queue from front to back. Whichever pending requests can now be accommodated with the increased AVAILABLE pool are granted. After handling an allocation/release request, the manager thread waits for the next request.

- **Worker threads:** N worker threads are created by the manager. Each worker thread enters a loop of making a total of R (non-zero) allocation/release requests. Between two consecutive requests, it sleeps for some random time. Adjust the sleep-time such that each worker thread can make $2-5$ requests per second. After the sleep, the worker thread decides whether it is going to make an **allocation request** (**with probability ⅓**) or a **release request** (**with probability ⅔**). A random request vector is generated. A sample implementation for generating an allocation request and a release request is supplied to you. This implementation guarantees that (i) no thread asks for more resource instances than the total number of instances (for each resource type), and (ii) a worker thread never releases more instances than it is holding (for each resource type). The functions return 0 if the vector generated is zero, or 1 if the vector contains at least one positive component. The first request must be an allocation request. No request should be made with the zero vector, so the return value of these functions should be consulted before making the request. After R requests are made, the worker thread makes a final release request (the $(R+1)$-st request to return all allocated resources, before exiting).

After making a request, the worker thread waits until an acknowledgment comes from the manager confirming that the request is processed. This handshaking is necessary because multiple worker threads may try to float requests at the same time. But the manager can handle requests only one at a time. So the worker requests followed by manager services must be sequentialized. When one request is being processed, other worker threads must wait for placing their requests.

If the request (being processed) is a release request, the worker thread continues as soon as the acknowledgment is received. If the request is an allocation request, the worker thread (after receiving the acknowledgment) waits until the request is granted. If the request can be immediately granted, the manager does so, and signals the waiting worker thread. Otherwise, the worker thread keeps on waiting until signaled at some later point of time when the manager (or the deadlock-handler) can grant that request.

- **Deadlock-handler thread:** This is again created by the master thread at the beginning. This thread periodically (after every second) and asynchronously (not initiated by any other thread) wakes up, and handles possible deadlocks present at that time. Make sure that sufficiently many (but not too many) resource requests are made during the time the deadlock handler sleeps. Recall that a worker makes $2-5$ requests per second. A worker may be blocked if an allocation request cannot be immediately served. Nevertheless, during a period of one second, $2N-5N$ requests are possible. It seems reasonable for the deadlock handler to check the status of the system after (at most) these many requests.

  When the deadlock handler wakes up, it first locks the system so that the usual request-service interaction between the worker threads and the manager is disabled. This is needed because the state of the system must not be altered by other threads when the deadlock handler is in action.

  After the lock is obtained, the deadlock handler runs the deadlock-detection algorithm of our textbook. If no deadlock is detected, that is, if all the active threads (that is, those which have not exited) can finish from the current situation, then there is nothing to be done. So the deadlock handler unlocks the system (so that the usual request-service interaction can resume), and goes to sleep again.

  Now suppose that a deadlock is detected by the deadlock handler. It takes a greedy approach to recover from the deadlock. For each worker thread that cannot finish from the current situation, the total resource allocation (the sum of instances of all resource types, allocated to that thread) is computed. The thread holding the maximum total allocation is chosen as the victim. If multiple threads hold the same maximum amount, the one with the smallest serial number is chosen from them as the victim. Notice that the victim must be staying in the queue RQ of waiting threads (otherwise it could finish). The deadlock handler releases the entire allocation made to the victim at that moment. Its pending request in the wait queue is kept as it is.

  After a resource preemption, the deadlock handler visits the waiting queue RQ from front to back, and serves those pending requests that can now be served by the increased AVAILABLE pool. Then the deadlock handler again checks if the system is still in a deadlocked state. If so, a second victim is chosen, its current allocation is preempted, and another pass in the wait queue is made to grant pending requests.

  This sequence of deadlock detection followed by preempting resources form a chosen victim continues until the system is no longer in a deadlocked state. The greedy choices of the victims tend to reduce the number of iterations in the deadlock-handling loop (although this is not an optimal strategy). After restoring the system in a deadlock-free state, the deadlock handler releases the system lock, and goes to sleep for the next second.

  The deadlock handler has a final work to do for terminating the session. The manager is in an infinite loop waiting for serving the next request. When the deadlock handler wakes up after a sleep and finds that there are no more active worker threads (all have exited), it sends a special QUIT request to the manager. This breaks the waiting loop for the manager, and the deadlock handler and the manager now exit.

Synchronization among the threads

The interaction among the threads needs to be synchronized. First, the system must be protected from simultaneous updates by multiple threads. Second, there are several waits involved: the manager waits for requests, and each worker thread waits for an acknowledgment from the manager and also for the granting of an allocation request. The system is to be protected by a single mutex. All waits are to be implemented by condition variables.

- **The resource mutex RMTX:** The program maintains several global variables (that constitute the state of the system). We let N be the number of worker threads, and M the number of resource types. An M-vector TOTAL stores the total numbers of instances of the M resource types. This is initialized by the manager at the beginning (before any other thread is created), and is used later in a read-only manner. Another M-vector AVAILABLE stores the counts of available instances of the M resource types. An $N \times M$ matrix ALLOCATION stores the current allocation of the resources to the threads, that is, ALLOCATION[i][j] is the number of instances of resource type j currently allocated to the worker thread i. The $N \times M$ matrix REQUEST stores the current allocation request made by the worker threads, that is, REQUEST[i][j] is the number of instances of resource type j that the worker thread i requests for (additional) allocation. An allocation request from Worker i may be served immediately or later. As soon as it is served, REQUEST[i] is cleared by the manager. Finally, an $N \times M$ matrix RELEASE is used for release requests. The manager serves a release request immediately, and clears the relevant row of the RELEASE matrix.

  Besides these, some other information are kept as global variables: N, M, R, REQFROM (which worker thread made the resource-related request), REQTYPE (the type of the request: ALLOCATE/RELEASE/QUIT), NACTIVE (the number of worker threads that have not exited), and an N-vector STATUS (the status of the N worker threads: ACTIVE/EXITED). Moreover, the queue RQ stores all the pending allocation requests. This queue may store only the serial numbers of the waiting threads; their exact request amounts can be obtained from the REQUEST matrix. So long as a thread waits, it cannot overwrite its earlier request, and after a request is served, the REQUEST row is cleared by the manager. Therefore the requested amounts themselves need not be stored in RQ.

  If needed, you can use additional global variables.

  All these global variables must be protected for mutually exclusive accesses (reads and writes). A **single mutex** RMTX is used for that purpose.

- **The service condition variable (SCND, SMTX):** The manager waits on the condition variable SCND. The mutex to be used in conjunction with SCND is SMTX. The manager locks this mutex all the time except implicitly releasing it when it starts a conditional wait on SCND. When a signal comes to SCND, the manager wakes up, does the resource-related work, and (for resource requests) sends an acknowledgment signal and (whenever appropriate) an allocation-granted signal to a worker thread, and restarts waiting on SCND. At the end, upon the reception of the QUIT request from the resource handler, the manager terminates.

- **The acknowledgment condition variable (ACND, AMTX):** In order to make a request, a worker thread locks RMTX first (that is, two requests cannot be placed and processed simultaneously). The worker then sets REQTYPE and REQFROM appropriately, and prepares a request or release vector. It then wakes up the manager by sending a signal to SCND, and starts waiting on the condition variable ACND. The manager attends the request, updates the global data structures appropriately, and sends a signal to ACND. Only after receiving this signal, the worker releases RMTX. The manager itself does not lock RMTX while updating global data. It instead proceeds under the security of the lock held by the worker thread, the request from which the manager is processing. Note that the same condition variable ACND and the associated mutex AMTX can be used by all worker threads. Since processing of resource requests are sequentialized by RMTX, there is no necessity to have worker-specific acknowledgment condition variables.

- **The condition variables for waiting for a resource (WCND_i, WMTX_i):** After receiving the acknowledgment signal from the manager and releasing RMTX, the worker thread does the following. If the request was of type release, the worker thread does not have to wait, so it goes to the top of its (sleep-and-)make-the-next-request loop. However, if the request was of the allocation type, the worker thread starts waiting on the worker-specific condition variable WCND_i. If the allocation request can be immediately granted, a signal comes to the worker thread immediately after the signal to ACND. Otherwise, the manager puts the worker thread (more precisely, its ID i) in the waiting queue RQ, and the worker thread continues its wait on its WCND_i. Later, when sufficiently many resource instances are available, a signal is sent to WCND_i.

We need M wait condition variables and M associated mutexes. This is quite needed, because a common wait condition variable may wake up any thread waiting on it (you have no control over which worker thread it would be). But the thread whose allocation request is granted must wake up. Therefore a common wait condition variable cannot be used.

It is to be noted that when no thread is waiting on a condition variable, a signal sent to that condition variable is lost (unlike in LA5, the signal is not stored). So any thread must first lock the associated mutex before it initiates the event that causes a signal to come to the condition variable. For example, the worker thread i must first lock AMTX (and WMTX_i for allocation requests) **before** it sends a signal to the manager's condition variable SCND. In order to send an acknowledgment signal to ACND, the manager has to lock AMTX. This is not possible until the worker thread (atomically) releases AMTX and starts waiting on ACND.

The deadlock-handler thread does not need to wait for any event, so no condition variables are needed for this thread. After its sleep, the thread locks RMTX and releases it only before going to the next sleep. This disables any processing of resource requests throughout the period when deadlock detection and recovery is in progress. Moreover, the deadlock handler needs to look at the STATUS array. The deadlock-detection algorithm should work only on those threads that are ACTIVE (not EXITED). If the currently allocated resources of a victim worker are preempted, then the deadlock handler attempts to serve some pending requests stored in RQ. If a pending request from worker thread i is granted, then the deadlock handler sends a signal to the i-th condition variable WCND_i.

What to submit

Write the work of the manager thread in a file *manager.c*(*pp*). Write the work of each worker thread in *worker.c*(*pp*). Finally, write the work of the deadlock handler in a file *dlhandler.c*(*pp*). Only the manager code has a *main*() function. This file `#include`'s the codes for workers and the deadlock handler. This should be done to enhance the readability of your submission. You may use other source files (like *global.c*(*pp*) to store all the global variables). Write a *makefile* with compile, run, and clean targets.

Pack all your source files and the *makefile* in a single zip/tar/tgz archive. Submit only that archive.

Sample Output

Use the default values $M = 8$, $N = 16$, and $R = 32$. For these (and even for smaller) values of M, N, and R, the output is huge. Some extracts are given below for $M = 5$, $N = 15$, and $R = 25$, in order to illustrate the format of printing. The complete transcript is made available in a separate file. The number of instances for each resource type is chosen randomly in the range $[8, 32]$.

```
$ make demorun
gcc -Wall -pthread -o manager manager.c
./manager 5 15 25
                            TOTAL = [ 22, 14, 10, 26, 12 ]
Worker  3 makes allocation request  [  0,  0,  0,  0,  1 ]
Worker  3 granted request
                        AVAILABLE = [ 22, 14, 10, 26, 11 ]
Worker  8 makes allocation request  [  0,  2,  0,  0,  0 ]
Worker  8 granted request
                        AVAILABLE = [ 22, 12, 10, 26, 11 ]
Worker 13 makes allocation request  [  0,  0,  0,  0,  2 ]
Worker 13 granted request
                        AVAILABLE = [ 22, 12, 10, 26,  9 ]
Worker 14 makes allocation request  [  0,  0,  0,  2,  2 ]
Worker 14 granted request
                        AVAILABLE = [ 22, 12, 10, 24,  7 ]
. . .

Worker 10 makes allocation request  [  2,  1,  0,  0,  1 ]
Worker 10 has to wait
        Workers waiting: (  1, 13, 10 )
Worker  6 makes release request     [  0,  0,  0,  1,  1 ]
                        AVAILABLE = [ 18,  1,  2, 12,  1 ]
        Workers waiting: (  1, 13, 10 )
Worker 10 granted pending request   [  2,  1,  0,  0,  1 ]
                        AVAILABLE = [ 16,  0,  2, 12,  0 ]
Worker  3 makes release request     [  1,  0,  0,  0,  0 ]
                        AVAILABLE = [ 17,  0,  2, 12,  0 ]
        Workers waiting: (  1, 13 )
Worker  4 makes allocation request  [  0,  0,  0,  2,  0 ]
Worker  4 granted request
```

Sequence of allocation and release requests.
After the release request from Worker 6,
Worker 10 can be granted its pending request.

```
                        AVAILABLE = [ 17,  0,  2, 10,  1 ]
Worker  8 makes allocation request  [  1,  3,  0,  3,  0 ]
Worker  8 has to wait
        Workers waiting: (  1, 13,  8 )
Worker  3 makes release request     [  1,  0,  0,  0,  0 ]
                        AVAILABLE = [ 18,  0,  2, 10,  1 ]
        Workers waiting: (  1, 13,  8 )
Worker 11 makes allocation request  [  0,  0,  0,  2,  0 ]
Worker 11 granted request
                        AVAILABLE = [ 18,  0,  2,  8,  1 ]
Worker  6 makes allocation request  [  2,  0,  0,  0,  0 ]
Worker  6 granted request
                        AVAILABLE = [ 16,  0,  2,  8,  1 ]
Worker  3 makes allocation request  [  0,  3,  3,  0,  3 ]
Worker  3 has to wait
        Workers waiting: (  1, 13,  8,  3 )
Worker 10 makes allocation request  [  0,  1,  0,  2,  0 ]
Worker 10 has to wait
        Workers waiting: (  1, 13,  8,  3, 10 )
Worker  4 makes allocation request  [  1,  1,  1,  3,  0 ]
Worker  4 has to wait
        Workers waiting: (  1, 13,  8,  3, 10,  4 )
```

Growth of the waiting queue

```
. . .
                                                         Deadlock detection in progress
                                                                     Finish sequence: 9
                                                                      Deadlock detected
                    Allocation status: 0:2 1:10 2:3 3:4 4:7 5:0 6:2 7:1 8:9 10:7 11:6 12:1 13:3 14:6
                                                Preempting resources from worker  1 with 10 resources
Worker 10 granted pending request   [  0,  1,  0,  2,  0 ]
                        AVAILABLE = [ 14,  1,  6,  7,  1 ]
Worker  4 granted pending request   [  1,  1,  1,  3,  0 ]
                        AVAILABLE = [ 13,  0,  5,  4,  1 ]

                                                         Deadlock detection in progress
                                  Finish sequence: 4 1 2 5 6 7 8 9 10 0 3 11 12 13 14
                                                                   No deadlock detected
```

Deadlock removed by preempting resources from only one thread

```
. . .

                                                         Deadlock detection in progress
                                                                 Finish sequence: 6 8 7
                                                                      Deadlock detected
                            Allocation status: 0:2 1:7 2:3 3:4 4:0 5:12 9:4 10:7 11:3 12:1 13:6 14:0
                                                Preempting resources from worker  5 with 12 resources
Worker 11 granted pending request   [  1,  0,  3,  2,  1 ]
                        AVAILABLE = [  5,  8,  0, 15,  0 ]

                                                         Deadlock detection in progress
                                                              Finish sequence: 6 8 7 11
                                                                      Deadlock detected
                                  Allocation status: 0:2 1:7 2:3 3:4 4:0 5:0 9:4 10:7 12:1 13:6 14:0
                                                Preempting resources from worker  1 with  7 resources

                                                         Deadlock detection in progress
                                                              Finish sequence: 6 8 7 11
                                                                      Deadlock detected
                                  Allocation status: 0:2 1:0 2:3 3:4 4:0 5:0 9:4 10:7 12:1 13:6 14:0
                                                Preempting resources from worker 10 with  7 resources
Worker  9 granted pending request   [  0,  0,  0,  0,  3 ]
                        AVAILABLE = [  9,  9,  1, 19,  1 ]

                                                         Deadlock detection in progress
                                  Finish sequence: 6 8 7 9 0 1 2 3 4 5 10 11 12 13 14
                                                                   No deadlock detected
```

Deadlock removed by preempting resources from three threads

```
. . .

        Worker  0 going to quit
        Releasing allocation        [  0,  0,  0,  0,  0 ]
                        AVAILABLE = [ 22, 12,  5, 24, 10 ]
        Workers waiting: ( )

                                                         Deadlock detection in progress
                                                                    Finish sequence: 12
                                                                   No deadlock detected
```

No deadlock detected

```
Worker 12 makes release request     [  0,  2,  0,  1,  2 ]
                        AVAILABLE = [ 22, 14,  5, 25, 12 ]
        Workers waiting: ( )
        Worker 12 going to quit
        Releasing allocation        [  0,  0,  5,  1,  0 ]
                        AVAILABLE = [ 22, 14, 10, 26, 12 ]
        Workers waiting: ( )
                                                                       All workers left
$
```

dlhandler finds NACTIVE = 0