CS39002 OPERATING SYSTEMS LABORATORY
SPRING 2025

LAB ASSIGNMENT: 8
DATE: 19–MARCH –2025

Deadlock avoidance by the banker's algorithm

In this assignment, you study the effects of deadlock in a multi-threaded application. We deal with the case of multiple instances of each resource type, so you need to use the banker's algorithm before the allocation of each request. Let there be m resource types, and n threads contending for these resources. Assume that $5 \leqslant m \leqslant 20$, and $10 \leqslant n \leqslant 100$. Use zero-based indexing.

Input

The input to the program is to be given in a subdirectory `input/`. This directory must contain $n + 1$ text files. The file `system.txt` stores m, n, and the total numbers of instances of the m resource types. These instance counts should be used to initialize the `AVAILABLE` array.

The resource requests of the threads are given (in `input/`) as the files `thread00.txt`, `thread01.txt`, `thread02.txt`, and so on (there should be n such files, one for each thread). The file for thread i consists of the following. The first line stores the maximum needs of the thread for the different resource types, that is, the initial values for `NEED[i][0]`, `NEED[i][1]`, . . . , `NEED[i][m - 1]`. This is followed by a number of resource request lines. Each request line looks as follows.

```
DELAY  R  REQ[0]  REQ[1]  REQ[2] . . . REQ[m - 1]
```

`DELAY` is the amount of time that the thread should wait for, after the last request was granted (or after the beginning of running if that is the first request). Each `REQ[j]` can be positive, negative, or 0. Let r = `REQ[j]`. If r is positive, then r additional instances of the j-th resource type is requested. If r is negative, then $|r|$ instances of the j-th resource type is released. If $r = 0$, then the thread continues to work without acquiring or releasing any instance of the j-th resource type.

If `REQ[j]` $\leqslant 0$ for all j, then the request is of type `RELEASE` (meaning release-only). If `REQ[j]` > 0 for at least one j, the request is of type `ADDITIONAL` (implying that additional instances of one or more resource types are asked for). An `ADDITIONAL` request may have `REQ[j]` $\leqslant 0$ for some values of j. That is, a demand for additional instances of certain resource types may be concomitantly placed along with release and/or continue-with components for other resource types.

After the last resource request, a single line should be present in the input file for each thread. This line looks as follows.

```
DELAY Q
```

The thread waits for time `DELAY` after its last resource request was granted, before it quits. Such a request is equivalent to a `RELEASE` request, that is, all the resource instances currently held by the thread should be returned to the `AVAILABLE` pool.

A code *geninput.c* is supplied to you for generating random inputs. This implementation ensures that each `R` line is valid, that is, no thread ever tries to release more instances than it currently holds, nor does the thread ever violate the maximum needs stated in the first line. The program needs the existence of the directory `input/` under the current directory.

The threads

Write a multi-threaded application to simulate the working of the threads. The *master* (main) *thread* plays the role of the OS, and n additional threads (called *user threads*) are created by it to simulate the non-OS threads making the resource requests as mentioned above. Use the `pthread` API.

The master thread first creates the necessary shared memory (global variables) and synchronization tools to be used later. The system-wide configurations are read from `input/system.txt`. The master thread creates the n user threads. Each user thread reads the appropriate file `input/thread??.txt`, and starts simulating the information presented in that file. The delays are simulated by proportional `usleep()`'s. A resource request cannot be handled at the user level, so each user thread sends the request to the master thread. After the last request (`Q` in the file), the user thread quits.

All requests are handled by the master thread. A request of type `RELEASE` should be non-blocking. The user thread that places that request does not need to wait for the request to be served. The master thread simply adds the released instances to the pool, and goes to process the next request.

A request of type `ADDITIONAL` is blocking (to the user thread which makes it). The master thread maintains a FIFO queue Q of pending requests. Upon receiving an `ADDITIONAL`-type request `REQ` from a user thread i, the master thread first looks at the negative entries in `REQ`. These are release components, so these instances are added to the `AVAILABLE` pool, and the corresponding entries in `REQ` are replaced by 0. Then, the pair (i, `REQ`) is enqueued to the back of Q. The user thread must wait until its request is granted.

Any request (even an `ADDITIONAL`-type one) potentially changes the `AVAILABLE` vector. The master thread, after releasing the instances in the request followed possibly by enqueuing the request, visits the queue Q in the sequence front to back. For each request R stored there, the master thread first checks whether the `AVAILABLE` pool can serve the request R. Moreover, if deadlock avoidance is in force, the master thread additionally checks whether serving R will leave the system in a safe state. If everything about serving R is feasible, that request is granted by changing the system-wide information `AVAILABLE`, `NEED`, and `ALLOC`. Moreover, R is removed from Q, and the user thread from which R originated is unblocked to proceed further.

Notice that the queue Q and the matrices/vectors needed are not used (not even read) by any user thread. Only the OS (the master thread) needs to handle these data structures. So these data structures need not be global. The implementations of these data structures are left to you. That is, you are allowed to use available library implementations.

Communications and synchronizations among the threads

In order that the master thread and the n user threads work properly, you need to use primitives supplied by the `pthread` API. The requirements for this assignment are enumerated now.

- After the initial book-keeping, the master thread launches all of the n user threads at the beginning. They must all be ready together before the subsequent simulation can start. Achieve this using a barrier BOS (beginning of session) initialized to $n + 1$.

- Requests (types, who sent them, and the m-vectors) are shared between user threads and the master thread using global memory. Maintain a single global copy of these, that is, all user threads share these data for communicating with the master thread. Mutual exclusion is to be enforced by using a mutex `rmtx`.

- A set of barriers is needed for appropriate handshakings among the threads. The master thread uses the barrier `REQB`, and the *i*-th user thread uses the barrier $\texttt{ACKB}_i$ (a single `ACKB` may also serve our needs, but it would be safer to use a thread-specific barrier for each of the *n* user threads).

  The master thread always waits on the barrier `REQB` (initialized to 2) for synchronizing itself with a resource request. A user thread *i*, after locking `rmtx`, writes its next request to global memory, and joins `REQB`. Both the master thread and the user thread *i* wake up when the two barrier joins are made. The master thread goes to read the details of the request from global memory.

  The user thread *i* must not release `rmtx` immediately after its handshaking using `REQB` is over. It must wait until the master thread reads the request completely to its *local* memory. This second handshaking is to be done using the acknowledgment barrier $\texttt{ACK}_i$. This barrier should again be initialized to 2. Only when the read-acknowledgment from the main thread is synchronized using this barrier, the user thread can release `rmtx`. Notice that the master thread does not need to write to global memory associated with the requests. It only reads the information to its local memory for future processing. So the master thread does not need to lock `rmtx`. It can safely operate under the lock held by the requester.

- A `RELEASE`-type request is non-blocking to the requesting thread. So the requesting thread can proceed with the next line from its input file. The master thread too releases the resources in the request (it affects only `ALLOC`, `NEED`, and `AVAILABLE` maintained by it privately), and proceeds to check the possibility of serving pending requests.

- An `ADDITIONAL`-type request is not served immediately (even if it can be). The master thread enqueues it and serves it later (even if it is the only available request). The requesting thread goes to a conditional wait using ($\texttt{cv}_i$, $\texttt{cmtx}_i$) specific to the requesting thread *i*. At a later time, when the master thread can serve this request of the user thread *i* stored in *Q*, it sends a signal to $\texttt{cv}_i$. Use condition variables. A general-purpose counting semaphore is not needed for this. This is because each user thread waits in its private condition queue, and a waiting user thread cannot send a new request. Make sure that a signal does not come before the conditional wait begins (a signal without a waiting thread is lost, so a subsequent wait on the condition variable will never end). This is unlike a counting semaphore where the count of signals is stored.

- In order that printing is not garbled, you may use a mutex `pmtx`.

- The master thread can keep track of how many user threads have exited. This is exactly the same as the number of `Q` requests it has handled so far. When all user threads are gone, the master thread can break out of its request-processing loop, and terminate itself.

Deadlock avoidance

Keep a compile-time flag whether you allow deadlocks to happen or avoid deadlocks using the banker's algorithm. If you allow deadlocks, then the sole criterion for serving a request is the availability of the instances requested. If you avoid deadlocks, you need to *additionally* run the banker's algorithm to check whether granting a request (even when the required instances are available) throws the system to an unsafe state. Only when sufficient resource instances are available and there are no safety issues, the master thread can grant a request pending in *Q*.

A sample `makefile` is given below. Run `make allow` to allow deadlocks to happen. Run `make avoid` to enforce deadlock avoidance by the banker's algorithm.

```
all:
	gcc -Wall -o resource -pthread resource.c
	gcc -Wall -D_DLAVOID -o resource_nodeadlock -pthread resource.c

allow: all
	./resource

avoid: all
	./resource_nodeadlock

db: geninput.c
	gcc -Wall -o geninput geninput.c
	./geninput 10 20

clean:
	-rm -f resource resource_nodeadlock geninput
```

Sample Output

Two sample input directories `input1` and `input2` will be provided to you. The simulation following the data of `input1` should not lead to a deadlock although the system sometimes go to unsafe states. The simulation following the data of `input2` would lead to a deadlock, and your program will hang. Wait for some time, and when you see that no new requests are coming (and all running threads are waiting in *Q*), you know that this is a case of deadlock (no need to detect deadlocks). Run the deadlock-avoidance version for the data of `input2`. Your program should now no longer allow the system to go to a deadlock, so it will run to completion. The associated output transcripts will also be supplied to you. Follow the output formats (what to print and how) as given in the transcripts.

Submit a single C/C++ source file.