Deadlock Prevention/Recovery

Real-world computation tasks usually require different resources to run, e.g. GPUs, disk spaces for caching dataset, and network bandwidth for communication. When multiple user tasks come with different resource requirements, deadlocks may happen and thus the program makes no progress. In this experiment, you should simulate a deadlock situation and either prevent it or detect and recover from it (we give you an deadlock example in `data/example.in`).

Codebase Introduction

We consider four types of resources: GPU, Network, Memory, Disk. We provide a basic thread-safe implementation of a *resource manager* `lib/resource_manager.h`, which holds a certain number of instances of each resource type. The resource manager assigns the resources to each user task (a thread with specified workload), and blocks the thread if it is in short of that type of resource. A user task, before/after any resource use, should call the resource manager with the `request` and `release` methods to access the resource. Other than the resource requests that might cause a deadlock, these tasks are independent, and you do not need to consider other synchronization among these tasks. (Of course, the resource manager itself should be thread-safe).

We also provide a simple implementation of a *thread manager* in `lib/thread_manager.h`, with which you are able to create, kill, and restart the threads that have been killed. You can add any information you needed to the thread manager (see the hints in the code). Note that you should carefully account for the resources held by the thread after you kill it.

In `lib/workload.h`, we provide an example user task that requests two resource types. Each user task requests two types of resources and uses them for an extended time period. In this experiment, you should not rely on this `workload` function, as we may adopt different implementation during testing. However, for any workload type, the worker thread will call `ResourceManager::budget_claim` to claim all resources it is going to use. The worker threads will eventually use all the resources they claim. It may also release some resources during processing. You are encouraged to add your own workload functions to test the correctness of your implementation under different circumstances.

In the example `main.cc` and `workload.h`, we provide a deadlock case example. Each user task is represented as a sequence of integers (i.e., workload arguments, see `main.cc` and `workload.h` for details). The order of different tasks are arbitrary -- you can start them simultaneously or sequentially, as long as they are thread-safe.

TODO

Given the initial number of instances of each resource type and the definition of each user task, your job is to run these tasks concurrently without deadlocks. To do this, you may prevent the deadlock from happening or recover from deadlocked threads by implementing a smart `ResourceManager`. Note that your implementation of the resource manager should not rely on the implementation of the workload.

If you use a prevention method, in cases where two tasks cannot run concurrently, the resource manager should deny requests to a task, and the task that does not get the resource should wait until it gets the requested resource.

If you use a detect-and-recover scheme, the resource manager should detect deadlocks and recover by killing a thread. You can assume that all the threads are recoverable, as long as you correctly reclaim all resources managed by the resource manager.

Your method should maximize the concurrency level to shorten the overall execution time and increase system utilization. That is, you should do your best to allow all possible concurrently executable threads to run concurrently.

You do not need to consider failure cases (i.e., a task fails in the middle because of user logic errors). I.e., you can safely assume that all tasks, if not deadlocked, will finish correctly and return the resource to the resource manager.

Grading

We will use a mixture of user tasks to test your scheduler for the following two requirements:

1. Correctness: all the tasks should eventually finish without deadlocks;
2. Performance: we will consider the time it takes for all the tasks. It should be significantly faster than running all tasks sequentially - and quite close to the theoretically best performance (i.e., an offline schedule that knows all the task behaviors previously and carefully schedule them so that no deadlock can happen).