EI338 Computer Systems Engineering

Project 6

November 26, 2019

Zhihui Xie, 517030910356

The environment used in this project is **Deepin 15.11**, the latest version of an open source operating system based on Debian's stable branch. The kernel version is **Linux version 4.15.0**.

Exercise

In this exercise, we will implement the banker's algorithm. Customers request and release resources from the bank. The banker will grant a request only if it leaves the system in a safe state. A request that leaves the system in an unsafe state will be denied.

Basic Structure

The banker will consider requests from n customers for m resources types and keep track of the resources. The banker will grant a request if it satisfies the safety algorithm outlined as follows:

Algorithm 1: Safety Algorithm

initialize $Work \leftarrow Available$ and $Finish[i] \leftarrow false$ for $i = 0, 1, \ldots, n-1$;
foreach *index i of customer* **do**
  if $Finish[i] == false$ && $Need[i] \leq Work$ **then**
    update $Work$, $Finish$;
  end
  $i \leftarrow 0$;
end
if $Finish[i] == true$ *for all* i **then**
  return *true*
end
return *false*

If a request does not leave the system in a safe state, the banker will deny it.

Implementation

To implement the banker's algorithm, we realize mainly three functions:

- *request_resource()*, which is used to request some amount of resources by a customer.
- *release_resource()*, which is used to release some amount of resources by a customer.
- *is_safe()*, which is used to check whether the current state is saft or not.

In addition, we maintain four global variables to keep track of the resources.

```
// the available amount of each resource
int available_res[NUMBER_OF_RESOURCE];

// the maximum demand of each customer
int max_demand[NUMBER_OF_CUSTOMER][NUMBER_OF_RESOURCE];

// the amount currently allocated to each customer
int allocated_res[NUMBER_OF_CUSTOMER][NUMBER_OF_RESOURCE];

// the remaining need of each customer
int remain_demand[NUMBER_OF_CUSTOMER][NUMBER_OF_RESOURCE];
```

Listing 1: Data Structures for Tracked Resources

is_safe()

The function *is_safe()* checks whether it is a safe state based on the current resources. It directly follows Algorithm 1.

Firstly, we copy data from the global variables to prevent changing the original data.

```
// copy from available_res
for (int i = 0; i < NUMBER_OF_RESOURCE; ++i)
    avalible_tmp[i] = available_res[i];

// test if customers are satisfied
for (int i = 0; i < NUMBER_OF_CUSTOMER; ++i) {
    finish = TRUE;

    for (int j = 0; j < NUMBER_OF_RESOURCE; ++j)
        finish &= (remain_demand[i][j] == 0);

    finish_tmp[i] = finish;
}
```

Listing 2: is_safe() I

Then, we want to keep finding a customer which can achieve its maximal demand with avalible resources so that its allocated resources can be released. Every time we find one, we start again from the beginning. This process terminates with no such a customer exists.

```
// find an index
for (int i = 0; i < NUMBER_OF_CUSTOMER; ++i) {
    is_enough = TRUE;

    for (int j = 0; j < NUMBER_OF_RESOURCE; ++j)
        is_enough &= (remain_demand[i][j] <= avalible_tmp[j]);

    if (!finish_tmp[i] && is_enough) {
        // update avaliable_tmp
        for (int j = 0; j < NUMBER_OF_RESOURCE; ++j)
            avalible_tmp[j] += allocated_res[i][j];

        // update finish_tmp
        finish_tmp[i] = TRUE;

        // every time avaliable_tmp is updated, back to front
```

```
            i = 0;
        }
    }
```

Listing 3: is_safe() II

Ana finally, if every customer is fully satisfied, i.e., every one releases its allocated resources once, the original state is safe. Otherwise it is unsafe.

```
for (int i = 0; i < NUMBER_OF_CUSTOMER; ++i)
    res &= finish_tmp[i];

return res;
```

Listing 4: is_safe() III

request_resource()

The function *request_resource()* will return -1 to deny the request if it does not leave the system in a saft state or return 0 to grant a request if it satisfies the safety algorithm. To achieve it in a simplest way, our strategy is to first update the global variables and then check whether it is safe using *is_safe()*. If not, we call *release_resource()* to easily release the wrongly allocated resources.

```
// check if the request is too greedy
for (int i = 0; i < NUMBER_OF_RESOURCE; ++i) {
    if (remain_demand[customer_id][i] < update[i] || available_res[i] < update[i])
        return -1;
}

// update
for (int i = 0; i < NUMBER_OF_RESOURCE; ++i) {
    available_res[i] -= update[i];
    allocated_res[customer_id][i] += update[i];
    remain_demand[customer_id][i] -= update[i];
}

if (!is_safe()) {
    // if not, recover the update by releasing the resources
    // release_resource(customer_id, update);
    release_resource(customer_id, update);

    return -1;
}
```

Listing 5: request_resource() I

release_resource()

Releasing resources is even easier. As long as the customer processes such amount of resources, we can directly update tracked resources in global variables to release them.

```
// check if the release is affordable
for (int i = 0; i < NUMBER_OF_RESOURCE; ++i) {
    if (allocated_res[customer_id][i] < update[i])
        return -1;
}
```

```
for (int i = 0; i < NUMBER_OF_RESOURCE; ++i) {
    available_res[i] += update[i];
    allocated_res[customer_id][i] -= update[i];
    remain_demand[customer_id][i] += update[i];
}

return 0;
```

Listing 6: release_resource()