106119029 , OS Lab 11

Dipesh Kafle

Code

```cpp
#include <algorithm>
#include <iostream>
#include <list>
#include <numeric>
#include <string>
#include <unordered_map>
#include <vector>

using namespace std;

#define FREE false
#define ALLOCATED true

// has memory details
struct memory_struct {
  bool isAllocated;
  size_t size;
  memory_struct(bool isAllocated, size_t size)
      : isAllocated(isAllocated), size(size) {}
};

// has process details
struct process_struct {
  unsigned id;
  unsigned memory_reqd;
  unsigned arrival;
  unsigned burst;
  bool isDone;
  list<memory_struct>::iterator allocated_address;
  process_struct(unsigned id, unsigned reqd, unsigned arrival, unsigned burst,
                 bool isDone = false)
      : id(id), memory_reqd(reqd), arrival(arrival), burst(burst),
        isDone(isDone), allocated_address(nullptr) {}
  unsigned get_free_time() const { return arrival + burst; }
```

```cpp
};

void allocate(vector<process_struct> &processes, list<memory_struct> &mem_list,
              unsigned sec) {
  for (process_struct &process : processes) {
    if (process.arrival == sec) {
      // first fit strategy
      bool found_enough_memory = false;
      for (auto mem = mem_list.begin(); mem != mem_list.end();
           std::advance(mem, 1)) {

        if (mem->isAllocated == false && mem->size >= process.memory_reqd) {
          unsigned remaining_mem = mem->size - process.memory_reqd;
          if (remaining_mem) {
            memory_struct new_mem_node_with_remaining_size(FREE, remaining_mem);
            // insert new node before the current node
            mem_list.insert(mem, new_mem_node_with_remaining_size);
          }
          process.allocated_address = mem;
          mem->isAllocated = true;
          mem->size = process.memory_reqd;
          found_enough_memory = true;
          cout << "P" << process.id << " is allocated memory.\n";
          break;
        }
      }
      if (!found_enough_memory) {
        cout << "P" << process.id << " has to wait.\n";
        process.arrival++;
      }
    }
  }
}

void free_the_completed_ones(vector<process_struct> &processes,
                             list<memory_struct> &mem_list, unsigned sec) {
  // check all the process and see if they want to be freed
  // i.e if their finishing time has been reached
  for (process_struct &process : processes) {
    if (process.get_free_time() == sec) {
      process.allocated_address->isAllocated = false;
      cout << "P" << process.id << " has been freed.\n";

      // memory before the current block
      list<memory_struct>::iterator address_before =
          std::next(process.allocated_address, -1);
```

```cpp
      // memory after the current block
      list<memory_struct>::iterator address_after =
          std::next(process.allocated_address, 1);

      // for joining memory
      if (address_after != mem_list.end() &&
          address_after->isAllocated == false) {
        process.allocated_address->size += (address_after->size);
        mem_list.erase(address_after);
      }
      if (address_before != mem_list.end() &&
          address_before->isAllocated == false) {
        address_before->size += (process.allocated_address->size);
        mem_list.erase(process.allocated_address);
        process.allocated_address = mem_list.end();
      }
    }
  }
}

int main() {
  list<memory_struct> mem_list;
  mem_list.push_back(
      memory_struct(FREE, 1000)); // initially has  a block of size 1000kb
  vector<process_struct> processes{
      process_struct(1, 212, 0, 2),  process_struct(2, 417, 2, 5),
      process_struct(3, 112, 4, 10), process_struct(4, 426, 6, 3),
      process_struct(5, 300, 8, 12), process_struct(6, 500, 9, 13),
      process_struct(7, 600, 13, 4)};

  unsigned sec = 0;
  while (true) {

    // check if all the process have been allocated memory and completed their
    // burst time as well. Can be checked by looping through the process and
    // checking if current time (sec) > process.get_free_time() . get_free_time
    // method just returns arrival+burst time
    if (all_of(processes.begin(), processes.end(),
               [sec](const process_struct &process) {
                 return process.get_free_time() < sec;
               })) {
      break;
    }
    std::cout << "At t=" << sec << "\n";
    std::cout << string(20, '-');
    std::cout << '\n';
```

```
    // frees the process that are to be freed at time = sec
    free_the_completed_ones(processes, mem_list, sec);

    // allocates memory to whichever procees is demanding it
    allocate(processes, mem_list, sec);
    cout << "\n";

    sec++;
  }

  std::cout << "\n\nFinal Allocation and deallocation times\n";
  std::cout << string(40, '-');
  std::cout << '\n';
  for (auto &process : processes) {
    cout << "P" << process.id << " allocated at " << process.arrival
         << " and freed at " << process.get_free_time()
         << ", size: " << process.memory_reqd << '\n';
  }
}
```

Output

```
λ ~/Acads/Sem4/CSLR42-OSLab/Lab11 ➜ ./a.out
At t=0
--------------------
P1 is allocated memory.

At t=1
--------------------

At t=2
--------------------
P1 has been freed.
P2 is allocated memory.

At t=3
--------------------

At t=4
--------------------
P3 is allocated memory.

At t=5
--------------------

At t=6
--------------------
P4 is allocated memory.

At t=7
--------------------
P2 has been freed.

At t=8
--------------------
P5 is allocated memory.
```

```
At t=9
--------------------
P4 has been freed.
P6 has to wait.

At t=10
--------------------
P6 has to wait.

At t=11
--------------------
P6 has to wait.

At t=12
--------------------
P6 has to wait.

At t=13
--------------------
P6 has to wait.
P7 has to wait.

At t=14
--------------------
P3 has been freed.
P6 is allocated memory.
P7 has to wait.

At t=15
--------------------
P7 has to wait.

At t=16
--------------------
P7 has to wait.
```

```
At t=17
--------------------
P7 has to wait.

At t=18
--------------------
P7 has to wait.

At t=19
--------------------
P7 has to wait.

At t=20
--------------------
P5 has been freed.
P7 has to wait.

At t=21
--------------------
P7 has to wait.

At t=22
--------------------
P7 has to wait.

At t=23
--------------------
P7 has to wait.

At t=24
--------------------
P7 has to wait.

At t=25
--------------------
P7 has to wait.
```

```
At t=26
--------------------
P7 has to wait.

At t=27
--------------------
P6 has been freed.
P7 is allocated memory.

At t=28
--------------------

At t=29
--------------------

At t=30
--------------------

At t=31
--------------------
P7 has been freed.



Final Allocation and deallocation times
----------------------------------------
P1 allocated at 0 and freed at 2, size: 212
P2 allocated at 2 and freed at 7, size: 417
P3 allocated at 4 and freed at 14, size: 112
P4 allocated at 6 and freed at 9, size: 426
P5 allocated at 8 and freed at 20, size: 300
P6 allocated at 14 and freed at 27, size: 500
P7 allocated at 27 and freed at 31, size: 600
```