Отчет о выполнении первого задания практикума по предмету "Распределенные системы"

Р.М. Куприй, 423 группа

Факультет ВМК МГУ имени М.В. Ломоносова

1. Постановка задания

Все 16 процессов, находящихся на разных ЭВМ сети, одновременно выдали запрос на вход в критическую секцию. Реализовать программу, использующую древовидный маркерный алгоритм для прохождения всеми процессами критических секций.

Критическая секция:
<проверка наличия файла “critical.txt”>;
if (<файл “critical.txt” существует>)
<сообщение об ошибке>;
<завершение работы программы>;
else
<создание файла “critical.txt”>;
Sleep (<случайное время>);
<уничтожение файла “critical.txt”>;

Для передачи маркера использовать средства MPI.

Получить временную оценку работы алгоритма. Оценить сколько времени потребуется, если маркером владеет нулевой процесс. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 (Ts=100,Tb=1). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

2. Описание алгоритма

Алгоритм маркерного дерева **Raymond**

Для написания программы использовался Древовидный маркерный алгоритм Рэймонда:

Попадание в критическую секцию(далее КС):

Если есть маркер, то процесс выполняет КС;

Если маркера нет, то процесс:

Помещает запрос в очередь запросов;

Посылает сообщение Запрос в направлении владельца маркера и ожидает сообщений;

Поведение при получении сообщений(есть 2 типа сообщений - Запрос и Маркер):

Пришло сообщение Маркер:

Взять 1-ый процесс из очереди и послать маркер автору(может быть, себе);

Поменять значение указателя в сторону маркера на актуальное;

Исключить запрос из очереди;

Если в очереди есть запросы, отправить Запрос в направлении владельца маркера;

Пришло сообщение Запрос:

Поместить запрос в очередь;

Если маркера нет, отправить Запрос в направлении маркера;

Иначе перейти к пункту 1 для Маркер'а

3. Программная реализация

Для реализации алгоритма спользовался язык C++ с средствами MPI для параллелизации.

Программа основана на использовании структуры дерева со вспомогательными методами и следующими членами класса: рангом процесса, рангами родителя и потомков, направлением на владельца маркера, флагом наличия маркера и очередью запросов.

```
struct tree {
  std::queue<int> request_queue;
  int rank;
  int root;
  int left;
  int right;
  int marker_pointer; // 0 - parent, 1 - left, 2 -right
  bool marker;

  tree() {}

  void critical();

  void receive(message_type message, int sender);

  void print();
}
```

Доступ к критической секции реализован в виде метода класса:

```
void critical() {
  if (access("critical.txt", F_OK) != -1) {
    std::cerr << "File exist" << std::endl;
    MPI_Finalize();
    exit(1);
  } else {
    fopen("critical.txt", "w");
    std::cout << "CRITICAL; rank: " << rank << std::endl;
    sleep(3);
    remove("critical.txt");
  }
}
```

Основная функция обработки запросов приведена ниже.

При посылки сообщения с маркером самому себе - происходит вход в критическую секцию.

4. Временная оценка

По условию задачи мы имеем 16 процессов, которые образуют сбалансированное дерево. Для инициализации необходимо 15 запросов. Затем нужно обойти дерево в глубину. Начнем обход с крайней (15) вершины, тогда для этого потребуется 23 операции. В итоге результирующая формула будет выглядеть так: $38 * (Ts + 1 * Tb)$.

```
void receive(message_type message, int sender) {
  if (message == MARKER) {

    if (!request_queue.empty()) {
      int requester = request_queue.front();
      request_queue.pop();

      std::cout << "send marker from " << rank << " to " << requester
                << std::endl;

      if (requester == rank) {
        marker = true;
        critical();
      } else if (requester == root) {
        marker = false;
        marker_pointer = 0;
        message_type tmp = MARKER;
        MPI_Send(&tmp, 1, MPI_INT8_T, requester, 0, MPI_COMM_WORLD);
      } else if (requester == left) {
        marker = false;
        marker_pointer = 1;
        message_type tmp = MARKER;
        MPI_Send(&tmp, 1, MPI_INT8_T, requester, 0, MPI_COMM_WORLD);
      } else if (requester == right) {
        marker = false;
        marker_pointer = 2;
        message_type tmp = MARKER;
        MPI_Send(&tmp, 1, MPI_INT8_T, requester, 0, MPI_COMM_WORLD);
      } else {
        std::cerr << "Invalid rank in queue!" << std::endl;
        MPI_Finalize();
        exit(2);
      }

      if (!request_queue.empty()) {
        int receiver = request_queue.front();
        request_queue.pop();
        receive(REQUEST, receiver);
      }
    }
  } else if (message == REQUEST) {
    request_queue.push(sender);
    if (marker) {
      receive(MARKER, rank);
    } else {
      if (marker_pointer == 0) {
        message_type tmp = REQUEST;
        MPI_Send(&tmp, 1, MPI_INT8_T, root, 0, MPI_COMM_WORLD);
      } else if (marker_pointer == 1) {
        message_type tmp = REQUEST;
        MPI_Send(&tmp, 1, MPI_INT8_T, left, 0, MPI_COMM_WORLD);
      } else if (marker_pointer == 2) {
        message_type tmp = REQUEST;
        MPI_Send(&tmp, 1, MPI_INT8_T, right, 0, MPI_COMM_WORLD);
      } else {
        std::cerr << "Invalid marker pointer" << std::endl;
        MPI_Finalize();
        exit(3);
      }
    }
  }
}
```