Федеральное агентство связи

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра вычислительных систем

**Лабораторная работа №4**

«Векторные часы (логические часы)»

Выполнил:

студент гр. МГ-211 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / \*\*\*.\*.\*/

подпись

Проверил:

Профессор

кафедры ВС \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ /Курносов М.Г./

ОЦЕНКА, подпись

Оглавление

[Задание 3](#_Toc126613000)

[Описание работы 4](#_Toc126613001)

[Результаты работы 6](#_Toc126613002)

[Код программы 9](#_Toc126613003)

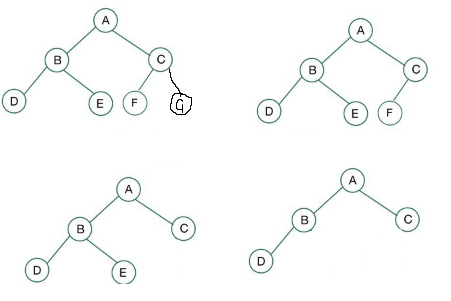
# Задание

1. Реализовать передачу сообщения из процесса 0 во все остальные по схеме завершенного бинарного дерева (complete binary tree) – каждый процесс на основные своего номера rank и числа процессов commsize определяет свое положение в дереве.

2. Какой процесс по логическим часам завершает работу последним?

# Описание работы

Завершённое двоичное дерево (complete binary tree) - это особый тип двоичного дерева, в котором все уровни дерева заполнены полностью, за исключением узлов самого низкого уровня, которые заполняются как можно левее.



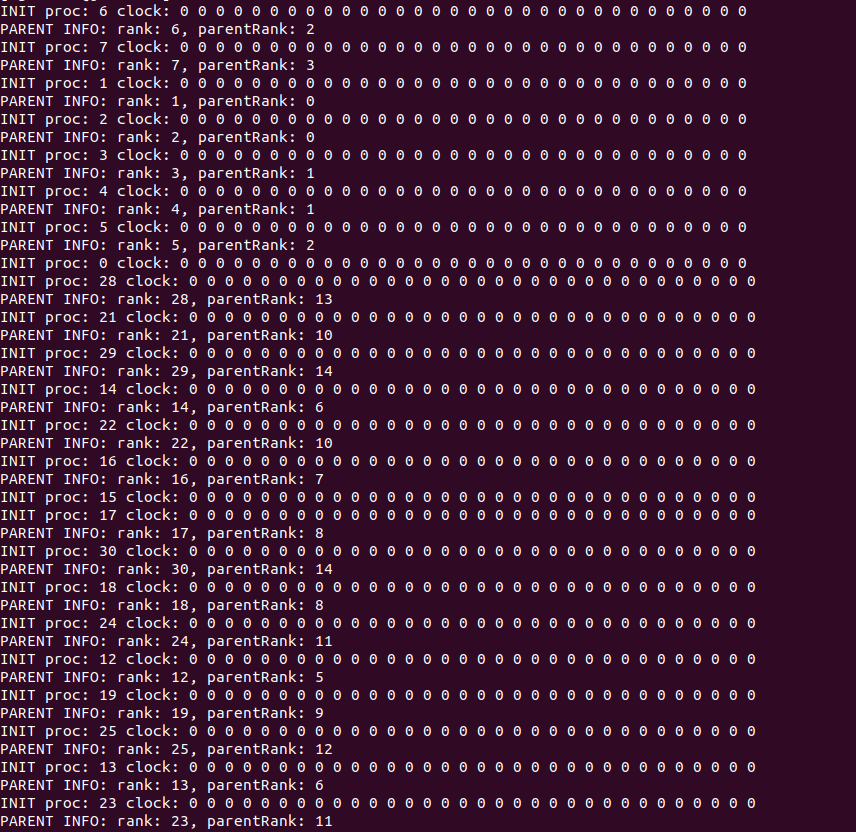
Векторные часы позволяют сравнивать любые два события и проверять, являются ли они причинно зависимыми или параллельными. Иными словами, можно не только показать, что если A "происходит до" B, то VectorClock(A) < VectorClock(B), но и утверждать обратное: если VectorClock(A) < VectorClock(B), тогда A "произошло до" B.

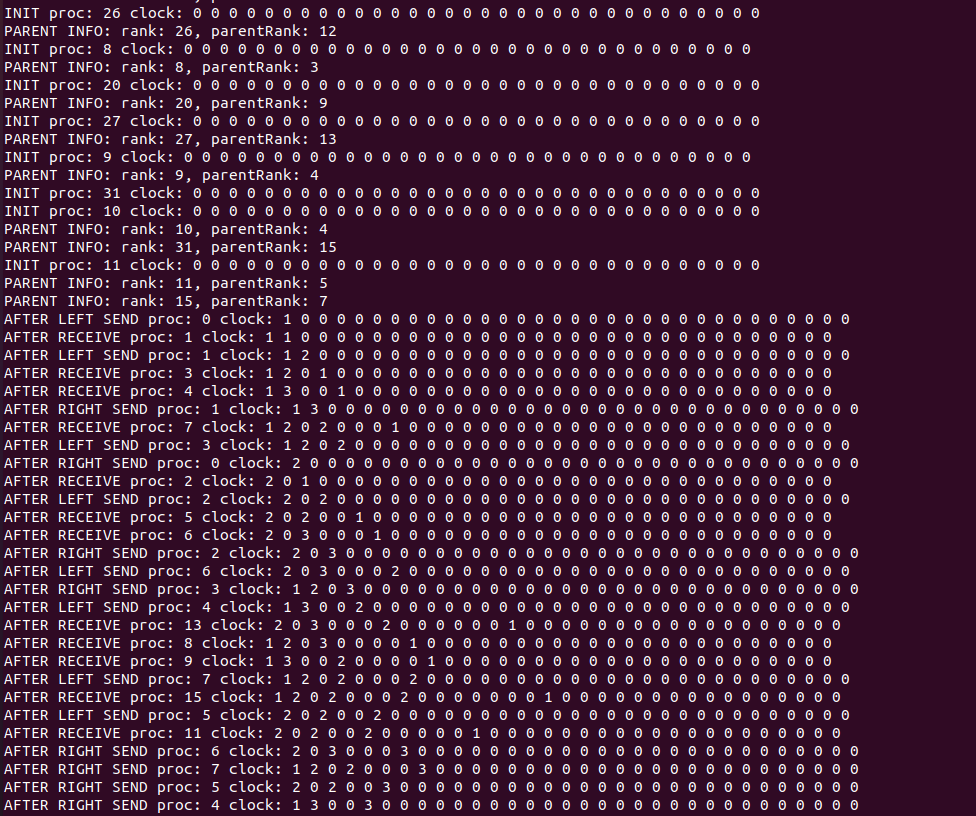
1. Для реализации задания был доработан исходный код программы, приложенной к заданию:

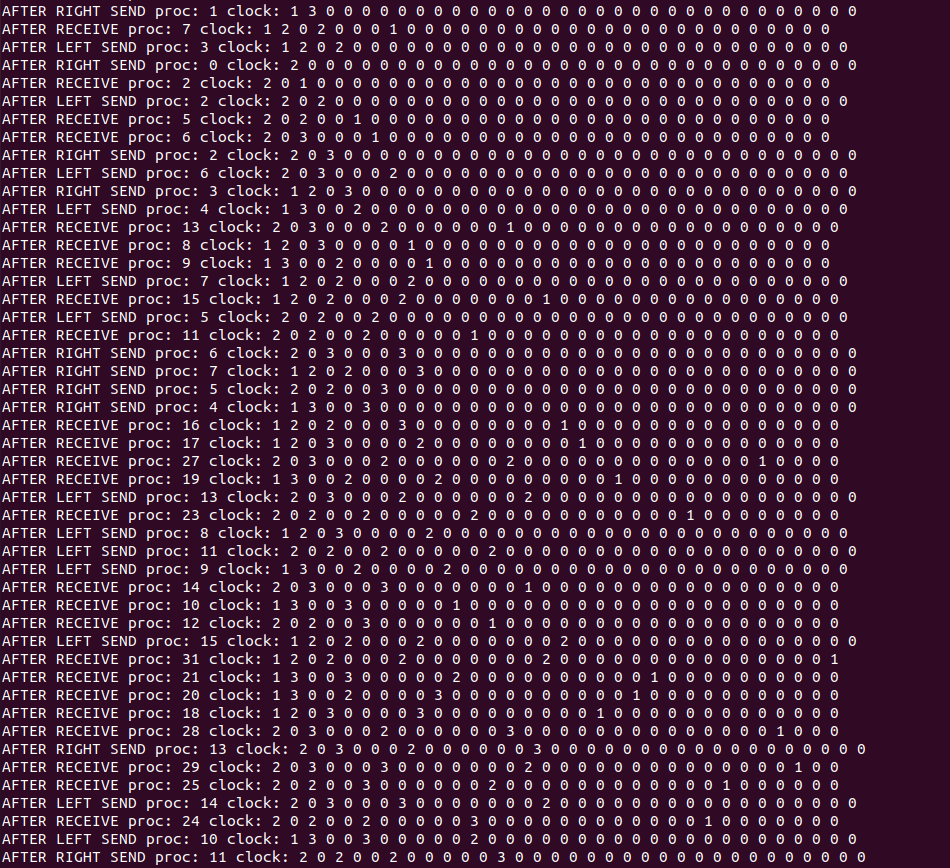
* После инициализации MPI среды, все процессы, кроме нулевого, вычисляют номер своего родителя в бинарном дереве по формуле: (собственный номер -1) div 2
* Этот номер они используют для получения сообщения от процесса-родителя с помощью метода dc\_recv.
* После принятия сообщения и корректировки своего локального вектора времени – процесс вычисляет номера двух процессов-потомков в бинарной структуре и отправляет собственный локальный вектор времени им.
* Далее освобождается память и завершается работа MPI среды процесса.

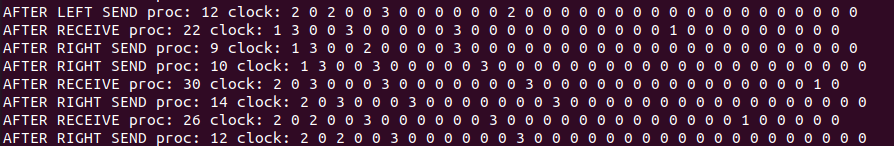
1. По логическим часам сложно определить какой процесс был завершён последним, но можно сказать какой процесс предшествовал какому, а какие выполнялись параллельно.

# Результаты работы









# Код программы

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct ds

{

int \*vclock;

int nprocs;

int rank;

MPI\_Comm comm;

} ds\_t;

ds\_t \*ds\_create(MPI\_Comm comm)

{

ds\_t \*ds = malloc(sizeof(\*ds));

if (NULL == ds)

{

return NULL;

}

ds->comm = comm;

MPI\_Comm\_size(comm, &ds->nprocs);

MPI\_Comm\_rank(comm, &ds->rank);

// sizeof(int) \* Comm\_size

ds->vclock = malloc(sizeof(\*(ds->vclock)) \* ds->nprocs);

if (NULL == ds->vclock)

{

free(ds);

return NULL;

}

for (int i = 0; i < ds->nprocs; i++) {

ds->vclock[i] = 0;

}

return ds;

}

void ds\_free(ds\_t \*ds)

{

free(ds->vclock);

free(ds);

}

void ds\_print\_clock(ds\_t \*ds)

{

printf("proc: %d clock: ", ds->rank);

for (int i = 0; i < ds->nprocs; i++)

printf("%d ", ds->vclock[i]);

printf("\n");

}

void ds\_event(ds\_t \*ds, char \*event)

{

ds->vclock[ds->rank]++;

printf("proc: %d event: '%s' time: %d\n", ds->rank, event, ds->vclock[ds->rank]);

}

int ds\_send(ds\_t \*ds, void \*buf, int count, MPI\_Datatype datatype, int dest, int tag)

{

int userbuf\_size, clockbuf\_size, packbuf\_size;

MPI\_Pack\_size(count, datatype, ds->comm, &userbuf\_size);

MPI\_Pack\_size(ds->nprocs, MPI\_INT, ds->comm, &clockbuf\_size);

packbuf\_size = userbuf\_size + clockbuf\_size;

char \*outbuf = malloc(sizeof(\*buf) \* packbuf\_size);

if (NULL == outbuf)

{

fprintf(stderr, "error: no enough memory\n");

MPI\_Abort(ds->comm, EXIT\_FAILURE);

}

int position = 0;

ds->vclock[ds->rank]++;

MPI\_Pack(buf, count, datatype, outbuf, packbuf\_size, &position, ds->comm);

MPI\_Pack(ds->vclock, ds->nprocs, MPI\_INT, outbuf, packbuf\_size, &position, ds->comm);

MPI\_Send(outbuf, position, MPI\_PACKED, dest, tag, ds->comm);

free(outbuf);

}

int ds\_recv(ds\_t \*ds, void \*buf, int count, MPI\_Datatype datatype, int src, int tag)

{

int userbuf\_size, clockbuf\_size, packbuf\_size, comm\_size;

MPI\_Pack\_size(count, datatype, ds->comm, &userbuf\_size);

MPI\_Pack\_size(ds->nprocs, MPI\_INT, ds->comm, &clockbuf\_size);

packbuf\_size = userbuf\_size + clockbuf\_size; // MessageSize + ClockVectorSize

char \*inbuf = malloc(sizeof(\*buf) \* packbuf\_size);

int \*vc = malloc(sizeof(\*vc) \* ds->nprocs);

if (NULL == inbuf || NULL == vc)

{

fprintf(stderr, "error: no enough memory\n");

MPI\_Abort(ds->comm, EXIT\_FAILURE);

}

MPI\_Recv(inbuf, packbuf\_size, MPI\_PACKED, src, tag, ds->comm, MPI\_STATUS\_IGNORE);

int position = 0;

MPI\_Unpack(inbuf, packbuf\_size, &position, buf, count, datatype, ds->comm);

MPI\_Unpack(inbuf, packbuf\_size, &position, vc, ds->nprocs, MPI\_INT, ds->comm);

free(inbuf);

ds->vclock[ds->rank]++;

for (int i = 0; i < ds->nprocs; i++)

{

ds->vclock[i] = vc[i] > ds->vclock[i] ? vc[i] : ds->vclock[i];

}

free(vc);

}

int main(int argc, char \*\*argv)

{

// Init

MPI\_Comm comm = MPI\_COMM\_WORLD;

int rank, commSize;

MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(comm, &rank);

MPI\_Comm\_size(comm, &commSize);

ds\_t \*ds = ds\_create(comm);

int buf;

if (rank == 0) {

buf = rank;

}

printf("INIT ");

ds\_print\_clock(ds);

if (rank != 0)

int parentRank = (int) (rank - 1) / 2;

printf("PARENT INFO: rank: %d, parentRank: %d\n", rank, parentRank);

ds\_recv(ds, &buf, 1, MPI\_INT, parentRank, 0);

printf("AFTER RECEIVE ");

ds\_print\_clock(ds);

}

int leftChildRank = 2 \* rank + 1;

int rightChildRank = 2 \* rank + 2;

if (leftChildRank < commSize) {

ds\_send(ds, &buf, 1, MPI\_INT, leftChildRank, 0);

printf("AFTER LEFT SEND ");

ds\_print\_clock(ds);

if (rightChildRank < commSize) {

ds\_send(ds, &buf, 1, MPI\_INT, rightChildRank, 0);

printf("AFTER RIGHT SEND ");

ds\_print\_clock(ds);

}

}

ds\_free(ds);

MPI\_Finalize();

return 0;

}