ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(государственный технический университет)

Кафедра 304
(вычислительные машины, системы и сети)

Курсовой проект по курсу «Операционные системы»

Семафоры и синхронизация процессов
(наименование работы)

Вариант задания *№9*.

Курсовой проект выполнил:

студент гр. 13-501, Резвяков Денис Михайлович _____.
(должность) *(Ф. И. О.)* *(подпись)*

Курсовой проект принял:

доц. каф.304, к.т.н. Алещенко Алла Степановна _____.
(должность) *(Ф. И. О.)* *(подпись)*

«*8*» *декабря* 2009 г.
(дата приёма)

Синхронизация процессов

Синхронизация процессов — приведение двух или нескольких процессов к такому их протеканию, когда определённые стадии разных процессов совершаются в определённом порядке, либо одновременно.

Синхронизация необходима в любых случаях, когда параллельно протекающим процессам необходимо взаимодействовать. Для её организации используются средства межпроцессного взаимодействия. Среди наиболее часто используемых средств — сигналы и сообщения, семафоры и мьютексы, каналы (пайпы) и совместно используемая память.

История семафоров

Семафоры, критические секции и исключения придумал выдающийся нидерландский учёный Э́дсгер Ви́бе Де́йкстра (Edsger Wybe Dijkstra) в начале 70 годов (1965-1973). В 1960-х годах он участвовал в создании ОС THE — первой операционной системы, построенной в виде множества параллельно исполняющихся взаимодействующих процессов. Именно в процессе этой работы появились понятия синхронизации процессов, идея семафора, а также была чётко осознана необходимость в структуризации процесса программирования и самих программ.

Рассматривая ситуации с многопользовательским доступом, он ввёл понятие критический интервал (critical section). Критический интервал — последовательность шагов обработки, в которой последовательный процесс не должен прерываться никаким другим процессом. Он сопоставил каждому, общему

для некоторых процессов, набору данных некоторую запирающую переменную (mutex), начальное значение которой равно 1. Если к началу критического интервала значение переменной больше нуля, то интервал должен выполняться, иначе ждать.

В результате Дейкстра составил следующий алгоритм:

```
mutex := 1;

{в начале критического интервала}
mutex := mutex - 1;
if mutex < 0 then
  {жди}
else
  {выполняй}

{в конце критического интервала}
mutex := mutex + 1;
if mutex <= 0 then
  {возьми из очереди очередной процесс и выполняй его}
else
  {всё, больше никого нет}
```

Во время критического интервала значение mutex равно нулю или отрицательному числу. Абсолютное значение его равно длине очереди. Чтобы лучше понять эти идеи нужно смотреть не со стороны программы, а со стороны операционной системы. Этот алгоритм именно для неё. Только когда два потока обратятся к критической секции в очереди станет 1 (= |−1|). И когда значение станет 1, тогда ни один поток не находится в критической секции. Для синхронизации со стороны программы было введено понятие семафор, который просто разрешал или запрещал обращаться потоку. Вот эту синхронизацию и назвали синхронизацией запираний или взаимным исключением (mutual exclusion — mutex). Вследствие ошибок при программировании синхронных событий могут возникать системные заторы (system-deadlock) или мертвые хватки (deadly embrace).

Семафоры

Семафо́р — это примитив синхронизации потоков, который позволяет войти в заданный участок кода не более чем `n` потокам. Определение ввёл нидерландский учёный — Э́дсгер Ви́бе Де́йкстра.

Операции семафора

С семафором можно выполнить три операции:

- `init(n)` — инициализировать счётчик семафора в значение `n`;
- `wait` — ждать пока счётчик станет больше `0`; после этого уменьшить счётчик на единицу (атомарная операция);
- `signal` — увеличить счётчик на единицу.

Принцип работы семафора

```
semaphore.init(3)

<...>

semaphore.wait
<...работа с ресурсом...>
semaphore.signal
```

Перед началом работы с семафором его инициализируют, устанавливая количество доступных ресурсов. Если потоку понадобился разделяемый ресурс, то он сначала занимает его операцией `wait`, а затем использует его. После окончания работы с ресурсом поток освобождает его операцией `signal`. Операция `wait` не допустит к ресурсу потоков больше, чем было указано при инициализации семафора. Если ресурс полностью занят, то очередной поток приостанавливается операцией `wait` до освобождения ресурса.

Стандартный семафор не гарантирует порядок, в котором заблокированные потоки будут получать доступ к ресурсу, т.е. доступ к освободившемуся ресурсу получит случайный поток.

Типичные ошибки

При использовании семафоров бывают следующие ошибки:

- «утечка семафора» — после окончания работы с ресурсом программист забывает освободить семафор, и ресурс блокируется «навсегда». Реже, случайно в разных местах освобождает его дважды, допуская в дальнейшем большее количество потоков в ресурс.
- взаимная блокировка — при работе с несколькими ресурсами один поток блокирует сначала первый ресурс, а затем, в процессе работы, блокирует и использует ещё и второй. Другой поток блокирует сначала второй ресурс, а затем — первый. В результате возможна ситуация, когда оба потока заблокируют по одному ресурсу и уйдут в ожидание, пока не освободится другой ресурс, не отпуская уже заблокированный ресурс.
- синхронизация процедур семафора — обычно семафоры предоставляет операционная система, и она сама следит за соблюдением атомарности* операций семафора. При создании семафора вручную возможна ситуация, что два потока узнают об освобождении семафора одновременно, а затем они вдвоём уменьшат счётчик на две единицы.

** атомарная операция — неделимая операция, т.е. такая операция либо выполняется полностью, либо не выполняется совсем*

В настоящее время семафорами называются только семафоры общего вида. Для частных случаев были введены упрощенные примитивы синхронизации потоков на основе семафоров.

Мьютекс

Мью́текс (англ. mutex, от mutual exclusion — взаимное исключение) — одноместный или двоичный семафор. Мьютекс может находиться в одном из двух состояний — отмеченном или неотмеченном (открыт и закрыт соответственно).
Мьютекс — это вариант семафора, проинициализированного единицей. В результате в каждый момент времени только один поток может владеть разделяемым ресурсом.

Фьютекс

Фью́текс (англ. futex, сокращение от fast userspace mutex) — альтернативный способ реализации семафоров и мьютексов. Фьютекс представляет собой выровненное целое в общей памяти (для нескольких процессоров). Это целое может быть увеличено или уменьшено на единицу за одну ассемблерную инструкцию. Процессы, «завязанные» на этот фьютекс, ждут, когда это значение станет положительным. Все операции с фьютексами практически полностью проводятся в пользовательском пространстве, а соответствующие функции ядра задействуются лишь в ограниченном наборе спорных случаев. Так как спорные случаи возникают редко, то скорость работы таких семафоров значительно возрастает.

Событие (Event)

Событие (event) — это объект синхронизации, состояние которого может быть установлено в сигнальное путём вызова специальных операций `SetEvent` или `PulseEvent`.

Событие — это вариант двоичного семафора, обычное состояние которого считается «занято», что блокирует ожидающие потоки. Если потоку следует подать определённый сигнал, то семафор просто освобождается.

Такой подход позволяет передать сигнал ещё до окончания другой работы потока. Либо поток может просто периодически проверять `event` на наличие сигнала, а в случае его отсутствия выполнять другую работу. Также механизм событий позволяет потоку остановиться и ожидать сигнала от одного из нескольких или сразу от всех событий.

Различают события с ручным и автоматическим сбросом. Вариант события с автоматическим сбросом, в отличие от ручного, сам устанавливается в несигнальное состояние после очередного считывания наличия сигнала ожидающим потоком.

Сравнение объектов синхронизации

Объект	Относительная скорость	Доступ нескольких процессов	Подсчёт числа обращений к ресурсу
Критическая секция	быстро	нет	нет (эксклюзивный доступ)
Мьютекс	медленно	да	нет (эксклюзивный доступ)
Фьютекс	быстро (если нет ожидания)	да	нет или автоматически
Семафор	медленно	да	автоматически
Событие	медленно	да	да

Постановка задачи

Разработаем программу, эмулирующую работу нескольких процессов работающих параллельно на одном микропроцессоре. *...т.е. обрабатываемых процессором последовательно* Все процессы будут обрабатываться в цикле. В зависимости от приоритета процессу может быть либо дана одна виртуальная единица времени для работы, либо он будет пропущен. Процессам для выполнения работы потребуется определённое устройство. В системе доступно несколько таких устройств. Работа с устройством такова, что не допускает использование одного устройства несколькими процессами в единицу времени, но само устройство не отслеживает, кто с ним работает. В таком случае процессы сами должны позаботиться, чтобы не мешать работе других процессов. Для этого воспользуемся вводом в эмулируемую систему семафора.

Разработанная программа позволит наглядно увидеть работу процессов с использованием семафоров, а также сделать вывод, действительно ли параллельная работа процессов с ограниченными ресурсами выгоднее, чем последовательная.

Планирование

Результатом работы программы будет выводимая ею в консоль статистика. Статистика будет состоять из трёх частей: заголовочной, пошаговой и итоговой.

Заголовочная часть содержит: список процессов и их приоритетов, а также заголовки свободного количества устройств (состояние семафора на конец текущего шага) и времени выполнения.

Внешний вид заголовочной части:

```
processes:  A     B     C     ... sem.time/i
priority:  high  norm  low    ...
```

`processes` — процессы, `priority` — приоритет, `high` — высокий (1), `norm` — средний (2), `low` — низкий (3), `sem.` — текущее значение семафора, `time/i` *(idle)* — время работы и время простоя на текущем шаге и за всё время.

Пошаговая часть содержит информацию о текущем шаге работы менеджера процессов. Внешний вид:

```
step  12:  1>0   2>1   1>0   ...  -3- + 2/0
           run  sleep d:run  ...     = 15/1
```

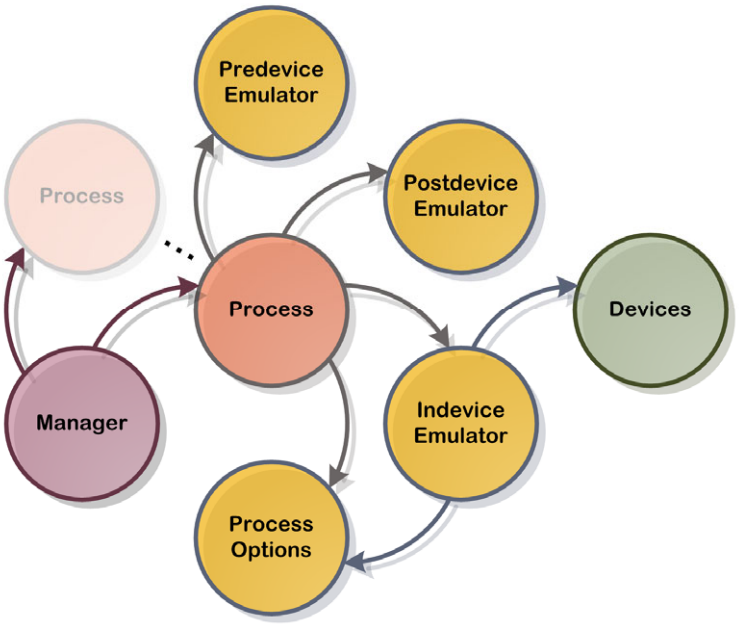
`step 12` — номер текущего шага, `1>0` — состояние переменной для соблюдения приоритета процесса до и после выполнения текущего шага, `-3-` — значение семафора, `+ 2/0` — время работы и простоя на текущем шаге, `= 15/1` — время работы/простоя за всё время моделирования, `run` — состояние процесса.

Возможные состояния процесса: `sleep` — процесс пропущен на текущем шаге, `run` — процесс отработал на текущем шаге одну единицу времени, `wait` — процесс ждёт освобождения устройства, `d:run` — процесс занял устройство и отработал с ним одну единицу времени, `d:slp` — процесс занял устройство и ждёт результатов его работы, `end` — процесс завершил свою работу.

Итоговая часть содержит время выполнения каждого процесса в отдельности и всех вместе. Внешний вид:

```
----------------------------------------
TotalTime:  13    26    37   ...     =  37
```

Граф взаимодействия объектов программы



Описание классов:

- `Manager` — класс эмулирующий работу менеджера процессов;
- `Process` — класс эмулирующий работу процесса;
- `ProcessOptions` — шаблон работы процесса;
- `PredeviceEmulator` — эмулятор работы процесса до работы с устройством;
- `IndeviceEmulator` — эмулятор работы процесса с устройством;
- `PostdeviceEmulator` — эмулятор работы процесса после работы с устройством;
- `Devices` — менеджер устройств, семафор.

Исходный код программы

Исходный код файла «Data.h»:

```
#pragma once

#define TotalDeviceCount 3

// Доступные типы эмуляторов работы процесса
enum EmulatorType { EmulatorType_Predevice, EmulatorType_Indevice,
    EmulatorType_Postdevice };

// Доступные варианты результата работы процесса в течении выделенного
// интервала времени.
enum Result { Result_Sleep, Result_Run, Result_Wait, Result_Finish };

// Абстрактный класс эмулятора работы процесса
class BaseEmulator;

// Класс задаёт шаблон работы одного процесса
class ProcessOptions {
public:
    ProcessOptions(int PredeviceTicks, int IndeviceTicks, int WaitDeviceTicks,
        int DeviceCount, int PostdeviceTicks, int TotalCount, int Priority);
    int PredeviceTicks;  // Время выполнения до захвата устройства
    int IndeviceTicks;   // Время выполнения работы с устройством
    int WaitDeviceTicks; // Время ожидания формирования результата устройством
    int DeviceCount;     // Кол-во повторений работы с устройством
    int PostdeviceTicks; // Время выполнения после захвата устройства
    int TotalCount;      // Кол-во полных повторений работы процесса
    int Priority;        // Приоритет процесса
};

// Класс эмулирует работу одного процесса по заданному шаблону
class Process {
public:
    Process(ProcessOptions* Options); // Конструктор класса

protected:
    ProcessOptions* m_Options; // Шаблон работы процессора
    int m_CurrentPriority;     // Текущий приоритет
    int InstateTicks;          // Отработано тактов в текущем состоянии
    int TotalStep;             // Текущий шаг общий
    BaseEmulator* m_Emulator;  // Текущий эмулятор работы процесса
    int m_FinishTick;          // Время, на котором работа была окончена
    bool m_NoPrint;            // Блокировка вывода статистики
    Result m_LastResult;       // Последний результат работы процесса

public:
    Result DoTick(int TotalTick); // Отработать текущий шаг
    void PrintState();            // Вывести состояние в последнем вызове
    void PrintPriority();         // Вывести приоритет в последнем вызове
    static void PrintSemafore();  // Вывести кол-во свободных устройств
    int GetFinishTick();          // Время, на котором работа была окончена
    int GetPriority();            // Приоритет процесса по шаблону
    Result GetResult();           // Последний результат работы процесса
};
```

Исходный код файла «Data.cpp»:

```
#include "stdafx.h"
#include "Data.h"

// Класс эмулирующий работу семафора
class Devices {
public:
    // Конструктор класса
    Devices(int DeviceCount) {
        this->CurrentCount = DeviceCount;
    }

    // Текущее кол-во свободных устройств
    int CurrentCount;

    // Попытаться занять устройство
    bool GetDevice() {
        if (this->CurrentCount) {
            this->CurrentCount--;
            return true;
        } else
            return false;
    }

    // Освободить устройство
    void ReleaseDevice()
    { this->CurrentCount++; }
};

// Устройства, эмулятор семафора
Devices* g_Devices = new Devices(TotalDeviceCount);

// Базовый абстрактный класс эмулятора работы процесса
class BaseEmulator abstract {
public:
    Result virtual DoTick(int TotalTick) = 0; // Отработать текущий шаг
    bool virtual IsFinished() = 0;            // Эмулятор закончил работу?
    EmulatorType virtual GetType() = 0;       // Тип класса эмулятора
    void virtual PrintState() = 0;            // Вывод текущего состояния
};

// Базовый класс эмулятор работы процесса до и после работы с устройством
class BaseNodeviceEmulator abstract : public BaseEmulator
{
protected:
    int m_TicksToWork; // Время, которое осталось обрабатывать устройство

public:
    // Отработать текущий шаг
    Result DoTick(int TotalTick) {
        m_TicksToWork--;
        return Result_Run;
    }

    // Эмулятор закончил эмуляцию своей части шаблона работы процесса?
    bool IsFinished() { return !m_TicksToWork; }
    // Вывод текущего состояния работы процесса
    void PrintState() { printf(" run  "); }
};
```

```
// Эмулятор работы процесса до работы с устройством
class PredeviceEmulator : public BaseNodeviceEmulator {
public:
    // Конструктор класса
    PredeviceEmulator(ProcessOptions* Options)
    { this->m_TicksToWork = Options->PredeviceTicks; }
    // Тип класса
    EmulatorType GetType() { return EmulatorType_Predevice; }
};

// Эмулятор работы процесса после работы с устройством
class PostdeviceEmulator : public BaseNodeviceEmulator {
public:
    // Конструктор класса
    PostdeviceEmulator(ProcessOptions* Options)
    { this->m_TicksToWork = Options->PostdeviceTicks; }
    // Тип класса
    EmulatorType GetType() { return EmulatorType_Postdevice; }
};

// Эмулятор работы процесса с устройством
class IndeviceEmilator : public BaseEmulator {
public:
    // Конструктор класса
    IndeviceEmilator(ProcessOptions* Options) {
        this->m_Options = Options;
        this->m_DeviceStep = (Options->DeviceCount
            && (Options->IndeviceTicks || Options->WaitDeviceTicks)) ? -1 : 0;
        this->m_SleepToTick = 0;
        this->m_TicksToWork = Options->IndeviceTicks;
    }

protected:
    ProcessOptions* m_Options; // Ссылка на шаблон работы процесса
    int m_DeviceStep;    // Текущий шаг в устройстве
    int m_SleepToTick;   // Ожидание до указанного здесь времени
    int m_TicksToWork;   // Время, которое осталось обрабатывать устройство
    Result m_LastResult; // Последний результат эмулятора

public:
    // Отработать текущий шаг
    Result DoTick(int TotalTick) {
        if (m_DeviceStep == -1) {
            if (!g_Devices->GetDevice()) return m_LastResult = Result_Wait;
            m_DeviceStep = m_Options->DeviceCount;
        }
        while (m_DeviceStep) {
            if (m_SleepToTick) { // Режим ожидания устройства
                if (m_SleepToTick <= TotalTick) { // Ожидание окончено
                    m_SleepToTick = 0;
                    m_TicksToWork = m_Options->IndeviceTicks;
                    m_DeviceStep--;
                    continue;
                } else return m_LastResult = Result_Wait; // Нужно ждать дальше
            } else { // Режим работы
                bool l_Runned = false;
                if (m_TicksToWork) { // Если работа не окончена
                    if (--m_TicksToWork) return m_LastResult = Result_Run;
                    l_Runned = true; // Если работа выполнена на текущем шаге,
                        // то следует вычислить такт окончания ожидания
                }
```

```
                m_SleepToTick = TotalTick + m_Options->WaitDeviceTicks;
                if (l_Runned) {
                    m_SleepToTick++; return m_LastResult = Result_Run; }
                else continue;
            }
        }
        g_Devices->ReleaseDevice();
        return m_LastResult = Result_Finish;
    }
    // Эмулятор закончил эмуляцию своей части шаблона работы процесса?
    bool IsFinished() { return !m_DeviceStep; }
    // Процесс находится в ожидании устройства (освобождения или ответа)?
    bool IsWaitOrSleep() { return m_LastResult == Result_Wait; }
    // Тип класса
    EmulatorType GetType() { return EmulatorType_Indevice; }
    // Вывод текущего состояния работы процесса
    void PrintState() { printf(m_DeviceStep == -1
        ? " wait " : m_LastResult == Result_Run ? "d:run " : "d:slp "); }
};

/*****************************************************************************
*************                      ProcessOptions                    *************
*****************************************************************************/

ProcessOptions::ProcessOptions(int PredeviceTicks, int IndeviceTicks,
        int WaitDeviceTicks, int DeviceCount, int PostdeviceTicks,
        int TotalCount, int Priority) {
    this->PredeviceTicks  = PredeviceTicks;
    this->IndeviceTicks   = IndeviceTicks;
    this->WaitDeviceTicks = WaitDeviceTicks;
    this->DeviceCount     = DeviceCount;
    this->PostdeviceTicks = PostdeviceTicks;
    this->TotalCount      = TotalCount;
    this->Priority        = Priority;
}

/*****************************************************************************
*************                        Process                         *************
*****************************************************************************/

// Конструктор класса
Process::Process(ProcessOptions* Options) {
    this->m_Options = Options;
    m_CurrentPriority = Options->Priority;
    this->TotalStep = Options->TotalCount;
    this->m_Emulator = this->TotalStep ? new PredeviceEmulator(Options) : 0;
    this->m_FinishTick = 0;
    this->m_NoPrint = false;
}

// Отработать текущий шаг
Result Process::DoTick(int TotalTick) {
    bool l_Sleep = false;
    if (!m_Emulator) return m_LastResult = Result_Finish;
    if ((m_Emulator->GetType() == EmulatorType_Indevice
                && ((IndeviceEmilator*)m_Emulator)->IsWaitOrSleep())
            || !--m_CurrentPriority) m_CurrentPriority = m_Options->Priority;
    else l_Sleep = true;

    Result result;
    do {
        while (m_Emulator && m_Emulator->IsFinished())
```

```
            switch (m_Emulator->GetType()) {
                case EmulatorType_Predevice:
                    delete m_Emulator;
                    m_Emulator = new IndeviceEmilator(m_Options);
                    break;
                case EmulatorType_Indevice:
                    delete m_Emulator;
                    m_Emulator = new PostdeviceEmulator(m_Options);
                    break;
                case EmulatorType_Postdevice:
                    delete m_Emulator;
                    if (!--TotalStep) {
                        m_Emulator = 0; m_FinishTick = TotalTick;
                        return m_LastResult = Result_Finish; }
                    m_Emulator = new PredeviceEmulator(m_Options);
                    break;
            }
        if (l_Sleep) return m_LastResult = Result_Sleep;
        result = m_Emulator->DoTick(TotalTick);
    } while (result == Result_Finish);
    return m_LastResult = result;
}

// Вывести состояние в последнем вызове
void Process::PrintState() {
    if (!m_Emulator) printf(m_NoPrint || !(m_NoPrint = true)
            ? "      " : " end  ");
    else if (m_Options->Priority == m_CurrentPriority) m_Emulator->PrintState();
    else printf("sleep ");
}

// Вывести приоритет в последнем вызове
void Process::PrintPriority() {
    if (!m_Emulator) printf(m_NoPrint ? "      " : " ---  ");
    else if (m_Emulator && m_Emulator->GetType() == EmulatorType_Indevice
            && ((IndeviceEmilator*)m_Emulator)->IsWaitOrSleep())
        printf("  -   ");
    else if (m_Options->Priority == m_CurrentPriority) printf(" 1>0  ");
    else {
        char buffer[7];
        strcpy(buffer, " ");
        itoa(m_CurrentPriority + 1, buffer + 1, 36);
        strcpy(buffer + 2, ">");
        itoa(m_CurrentPriority, buffer + 3, 36);
        strcpy(buffer + 4, "  ");
        printf(buffer);
    }
}

// Вывести кол-во свободных устройств
void Process::PrintSemafore() {
    char buffer[5];
    itoa(g_Devices->CurrentCount, buffer + 1, 36);
    buffer[0] = '-';
    strcpy(buffer + 2, "- ");
    printf(buffer);
}
// Время, на котором работа была окончена
int Process::GetFinishTick() { return m_FinishTick; }
// Приоритет процесса по шаблону
int Process::GetPriority() { return m_Options->Priority; }
// Последний результат работы процесса
Result Process::GetResult() { return m_LastResult; }
```

Исходный код файла «Manager.h»:

```
#pragma once

// Класс эмулирует менеджер управления процессами
class Manager {
private:
    int m_Step;           // Номер шага обработки всех процессов
    int m_TotalTick;      // Текущее виртуальное время эмулятора
    int m_Idle;           // Время простоя эмулируемого процессора
    int m_StepTicks;      // Время работы на текущем шаге
    int m_StepIdle;       // Время простоя на текущем шаге
    int m_ProcessCount;   // Кол-во эмулируемых процессов
    Process** m_Processes; // Ссылка на массив эмулируемых процессов

    // Форматирует текст о времени работы по числовым значениям
    void FormatTicks(char* Buffer, int Ticks, int Idle);
    // Выводит текст в центре области указанной длины
    void PrintIntAtCenter(int Value, int Length);

public:
    Manager(Process** Processes, int Count); // Конструктор класса
    bool RunStep();      // Запустить обработку очередного шага
    void PrintHeader(); // Вывод заголовка статистики
    void PrintStep();   // Вывод статистики по текущему шагу
    void PrintFooter(); // Вывод итогов статистики
};
```

Исходный код файла «Manager.cpp»:

```
#include "stdafx.h"
#include "Data.h"
#include "Manager.h"

// Конструктор класса
Manager::Manager(Process** Processes, int Count) {
    m_Step = 0;
    m_TotalTick = 0;
    m_Idle = 0;
    m_StepTicks = 0;
    m_StepIdle = 0;
    m_Processes = Processes;
    m_ProcessCount = Count;
}

// Запустить обработку очередного шага
bool Manager::RunStep() {
    int l_BeginTick = m_TotalTick;
    bool l_AllWait = true;
    bool l_AllFinish = true;
    for (int index = 0, count = m_ProcessCount; index < count; index++) {
        Process* l_Process = m_Processes[index];
        Result l_Result = l_Process->DoTick(m_TotalTick);
        if (l_Result == Result_Run) m_TotalTick++;
        if (l_Result != Result_Finish) {
            l_AllFinish = false;
            if (l_Result != Result_Wait) l_AllWait = false;
        }
    }
```

```
    m_Step++;
    m_StepIdle = 0;
    if ((m_StepTicks = m_TotalTick - l_BeginTick) == 0) {
        if (l_AllFinish) return false;
        if (l_AllWait) {
            m_TotalTick++;
            m_Idle++;
            m_StepIdle = 1;
        }
    }
    return true;
}

// Вывод заголовка статистики
void Manager::PrintHeader() {
    char l_Buffer[32];
    int count = m_ProcessCount;

    printf("processes:");
    strcpy(l_Buffer, "  x   ");
    for (int index = 0; index < count; index++) {
        l_Buffer[2] = index + 'A';
        printf(l_Buffer);
    }
    printf("sem.time/i\npriority: ");

    strcpy(&l_Buffer[0],  " high ");
    strcpy(&l_Buffer[8],  " norm ");
    strcpy(&l_Buffer[16], " low  ");
    strcpy(&l_Buffer[24], " (x)  ");
    for (int index = 0; index < count; index++) {
        int l_Priority;
        switch (l_Priority = m_Processes[index]->GetPriority()) {
            case 1:  printf(&l_Buffer[0]); break;
            case 2:  printf(&l_Buffer[8]); break;
            case 3:  printf(&l_Buffer[16]); break;
            default: l_Buffer[26] = l_Priority + '0'; printf(&l_Buffer[24]);
                break;
        }
    }
    printf("\n");
}

// Вывод статистики по текущему шагу
void Manager::PrintStep() {
    printf("\n");

    char buffer[11];
    itoa(m_Step, buffer, 10);
    int lenght = strlen(buffer);
    memcpy(buffer + 8 - lenght, buffer, lenght);
    memcpy(buffer, "step    ", 8 - lenght);
    strcpy(buffer + 8, ": ");
    printf(buffer);

    int count = m_ProcessCount;
    for (int index = 0; index < count; index++)
        m_Processes[index]->PrintPriority();
```

```
    Process::PrintSemafore();
    FormatTicks(buffer, m_StepTicks, m_StepIdle);
    buffer[0] = '+';
    printf(buffer);

    printf("\n          ");
    for (int index = 0; index < count; index++)
        m_Processes[index]->PrintState();

    FormatTicks(buffer, m_TotalTick - m_Idle, m_Idle);
    printf("   =%s\n", buffer);
}

// Форматирует текст о времени работы по числовым значениям
void Manager::FormatTicks(char* Buffer, int Ticks, int Idle) {
    itoa(Ticks, Buffer, 10);
    int lenght = strlen(Buffer);
    if (lenght < 3) {
        memcpy(Buffer + 3 - lenght, Buffer, lenght);
        memcpy(Buffer, "  ", 3 - lenght);
    }
    Buffer[3] = '/';
    itoa(Idle, Buffer + 4, 10);
    if (Buffer[5]) Buffer[6] = 0; else strcpy(Buffer + 5, " ");
}

// Вывод итогов статистики
void Manager::PrintFooter() {
    printf("----------");
    for (int index = m_ProcessCount; index; index--)
        printf("------");
    printf("----------\n");

    printf("TotalTime:");
    for (int index = 0, count = m_ProcessCount; index < count; index++)
        PrintIntAtCenter(m_Processes[index]->GetFinishTick(), 6);
    printf("   =");
    PrintIntAtCenter(m_TotalTick, 6);
    printf("\n");
}

// Выводит текст в центре области указанной длины
void Manager::PrintIntAtCenter(int Value, int Length) {
    char* buffer = new char[Length + 1];
    itoa(Value, buffer, 10);
    int length = strlen(buffer);
    int begin = (Length - length) / 2;
    if (begin) {
        memcpy(buffer + begin, buffer, length);
        memset(buffer, ' ', begin);
    }
    memset(buffer + begin + length, ' ', Length - begin - length);
    buffer[Length] = 0;
    printf(buffer);
}
```

Исходный код файла «Semaphore.cpp»:

```
#include "stdafx.h"
#include "Data.h"
#include "Manager.h"

#define ProcessesCount 7

// Инициализация менеджера процессов, задание параметров эмуляции
Manager* Initialize() {
    // Задание шаблонов работы процессов
    ProcessOptions* l_OptionsH = new ProcessOptions(2, 1, 2, 3, 1, 2, 1);
    ProcessOptions* l_OptionsN = new ProcessOptions(2, 1, 2, 3, 1, 2, 2);
    ProcessOptions* l_OptionsL = new ProcessOptions(2, 1, 2, 3, 1, 2, 3);

    Process** l_Processes = new Process*[ProcessesCount];
    l_Processes[0] = new Process(l_OptionsH);
    l_Processes[1] = new Process(l_OptionsH);
    l_Processes[2] = new Process(l_OptionsN);
    l_Processes[3] = new Process(l_OptionsN);
    l_Processes[4] = new Process(l_OptionsN);
    l_Processes[5] = new Process(l_OptionsL);
    l_Processes[6] = new Process(l_OptionsL);
    return new Manager(l_Processes, ProcessesCount);
}

// Главная функция эмулятора
int _tmain(int argc, _TCHAR* argv[]) {
    Manager* l_Manager = Initialize();
    l_Manager->PrintHeader();

    bool result;
    do {
        result = l_Manager->RunStep();
        l_Manager->PrintStep();
    } while (result);
    l_Manager->PrintFooter();
    delete l_Manager;
    scanf("\n");
    return 0;
}
```

Результат работы программы

Эмуляция производилась с тремя устройствами и семью процессами. Двум процессам дан высокий приоритет, трём — средний и ещё двум — низкий. Процессы работают по следующему алгоритму: Процесс отрабатывает два интервала времени. Затем захватывает устройство и работает с ним: один интервал времени работает, два — ждёт результатов, и так три раза. После работы с устройством процесс работает ещё один интервал времени. Вся процедура работы выполняется дважды.

Наглядный пример алгоритма:

```
for a := 1 to 2 do begin
  work(2);
  getDevice;
  for b := 1 to 3 do begin
    work(1);
    waitDevice(2);
  end;
  releaseDevice;
  work(1);
end;
```

Программа вернула следующую статистику по эмуляции:

```
processes:  A     B     C     D     E     F     G   sem.time/i
priority:  high  high  norm  norm  norm  low   low

step   1:  1>0   1>0   2>1   2>1   2>1   3>2   3>2  -3- + 2/0
           run   run  sleep sleep sleep sleep sleep    =  2/0

step   2:  1>0   1>0   1>0   1>0   1>0   2>1   2>1  -3- + 5/0
           run   run   run   run   run  sleep sleep    =  7/0

step   3:  1>0   1>0   2>1   2>1   2>1   1>0   1>0  -1- + 4/0
          d:run d:run sleep sleep sleep  run   run     = 11/0

step   4:  1>0   1>0   1>0   1>0   1>0   3>2   3>2  -1- + 5/0
          d:run d:run  run   run   run  sleep sleep    = 16/0

step   5:  1>0   1>0   2>1   2>1   2>1   2>1   2>1  -1- + 2/0
          d:run d:run sleep sleep sleep sleep sleep    = 18/0

step   6:   -     -    1>0    -     -    1>0   1>0  -0- + 3/0
          d:slp d:slp d:run  wait  wait  run   run     = 21/0

step   7:  1>0   1>0   2>1   1>0   1>0   3>2   3>2  -0- + 4/0
           run   run  sleep d:run d:run sleep sleep    = 25/0

step   8:  1>0   1>0   1>0   2>1   2>1   2>1   2>1  -0- + 3/0
           run   run  d:run sleep sleep sleep sleep    = 28/0
```

```
step   9:  1>0   1>0   2>1   1>0   1>0    -     -   -0- + 4/0
           run   run  sleep d:run d:run  wait  wait    = 32/0

step  10:   -     -    1>0   2>1   2>1    -     -   -0- + 1/0
          wait  wait d:run sleep sleep  wait  wait    = 33/0

step  11:   -     -    2>1   1>0   1>0    -     -   -0- + 2/0
          wait  wait sleep d:run d:run  wait  wait    = 35/0

step  12:   -     -    1>0   2>1   2>1   1>0    -   -0- + 2/0
          wait  wait  run  sleep sleep d:run  wait    = 37/0

step  13:   -     -    2>1   1>0   1>0   3>2   1>0  -1- + 3/0
          wait  wait sleep  run   run  sleep d:run    = 40/0

step  14:  1>0    -    1>0   2>1   2>1   2>1   3>2  -0- + 2/0
         d:run  wait  run  sleep sleep sleep sleep    = 42/0

step  15:   -     -    2>1   1>0   1>0   1>0   2>1  -0- + 3/0
         d:slp  wait sleep  run   run  d:run sleep    = 45/0

step  16:  1>0    -    1>0   2>1   2>1   3>2   1>0  -0- + 3/0
         d:run  wait  run  sleep sleep sleep d:run    = 48/0

step  17:  1>0    -    2>1   1>0   1>0   2>1   3>2  -0- + 3/0
         d:run  wait sleep  run   run  sleep sleep    = 51/0

step  18:  1>0   1>0    -    2>1   2>1   1>0   2>1  -0- + 3/0
           run  d:run  wait sleep sleep d:run sleep    = 54/0

step  19:  ---    -     -     -     -    3>2   1>0  -0- + 1/0
           end  d:slp  wait  wait  wait sleep d:run    = 55/0

step  20:         1>0    -     -     -    2>1   3>2  -0- + 1/0
                d:run  wait  wait  wait sleep sleep    = 56/0

step  21:          -     -     -     -    1>0   2>1  -1- + 1/0
                d:slp  wait  wait  wait  run  sleep    = 57/0

step  22:          -    1>0    -     -    3>2   1>0  -1- + 2/0
                d:slp d:run  wait  wait sleep  run     = 59/0

step  23:         1>0   2>1   1>0    -    2>1   3>2  -0- + 2/0
                d:run sleep d:run  wait sleep sleep    = 61/0

step  24:          -    1>0   2>1    -    1>0   2>1  -0- + 2/0
                d:slp d:run sleep  wait  run  sleep    = 63/0

step  25:         1>0   2>1   1>0   1>0   3>2   1>0  -0- + 4/0
                  run  sleep d:run d:run sleep  run     = 67/0

step  26:         ---   1>0   2>1   2>1   2>1   3>2  -0- + 1/0
                  end  d:run sleep sleep sleep sleep    = 68/0

step  27:               2>1   1>0   1>0   1>0   2>1  -0- + 3/0
                      sleep d:run d:run  run  sleep    = 71/0

step  28:               1>0   2>1   2>1   3>2   1>0  -1- + 2/0
                        run  sleep sleep sleep  run     = 73/0

step  29:               ---   1>0   1>0   2>1   3>2  -2- + 2/0
                        end   run  d:run sleep sleep    = 75/0
```

```
step  30:                    ---   2>1   1>0   2>1  -1- + 1/0
                            end  sleep d:run sleep    = 76/0

step  31:                           -    3>2   1>0  -0- + 1/0
                                 d:slp sleep d:run    = 77/0

step  32:                            1>0   2>1   3>2  -1- + 1/0
                                     run  sleep sleep    = 78/0

step  33:                            ---   1>0   2>1  -1- + 1/0
                                     end  d:run sleep    = 79/0

step  34:                                  3>2   1>0  -1- + 1/0
                                          sleep d:run    = 80/0

step  35:                                  2>1   3>2  -1- + 0/0
                                          sleep sleep    = 80/0

step  36:                                    -    2>1  -1- + 0/0
                                          d:slp sleep    = 80/0

step  37:                                    -     -   -1- + 0/1
                                          d:slp d:slp    = 80/1

step  38:                                  1>0   1>0  -1- + 2/0
                                          d:run d:run    = 82/1

step  39:                                  3>2   3>2  -1- + 0/0
                                          sleep sleep    = 82/1

step  40:                                  2>1   2>1  -1- + 0/0
                                          sleep sleep    = 82/1

step  41:                                    -     -   -1- + 0/1
                                          d:slp d:slp    = 82/2

step  42:                                   1>0   1>0  -3- + 2/0
                                            run   run     = 84/2

step  43:                                   ---   ---  -3- + 0/0
                                            end   end     = 84/2
-------------------------------------------------------------------
TotalTime:  54    67    73    75    78    86    86     =  86
```

Вывод

Процессу требовалось для выполнения работы 24 единицы времени, половину из которых процесс проводил в ожидании. В параллельной работе во время ожидания одного процесса процессор обрабатывал другие, что позволило выполнить общую работу почти в два раза быстрее. А эффективно использовать устройства, не мешая друг другу, удалось с помощью семафора.