Министерство образования и науки РФ

Томский государственный университет систем управления и радиоэлектроники (ТУСУР)

Кафедра комплексной информационной безопасности электронно-вычислительных систем (КИБЭВС)

А.С. Романов

Системное программирование

Тема № 6 - Синхронизация потоков и процессов

Томск 2014

Цель работы

Изучить средства синхронизации потоков и процессов. Познакомиться с соответствующими функциями WinAPI и POSIX API. В процессе изучения основного материала познакомиться также с функциями отображения файлов на память и директивами препроцессора.

Краткие теоретические сведения

1. Синхронизация, общие сведения

В случае если два или более процесса или потока используются один общий разделяемый ресурс, возникает проблема синхронизации. Например, несколько процессов обрабатывают данные из одного файла, используют общую переменную и т.д. Часть программы, в которой осуществляется доступ к разделяемым данным, называется **критической секцией**. Проблема может решаться приостановкой и активизацией процессов, организацией очередей, блокированием и освобождением ресурсов. Пренебрежение вопросами синхронизации может привести к неправильной работе программы, порче данных и критическим ошибкам операционной системы из-за возникновения следующих ситуаций.

1. **Взаимоблокировки (тупики)**. Процессы находится в тупиковой ситуации, если каждый процесс из группы ожидает события, которое может вызвать только другой процесс из этой же группы.

2. **Голодание (бесконечная отсрочка)**. Остановка работы одного или нескольких процессов на неопределенное время вследствие исполнения процессов с большим или равным приоритетом и/или длительным временем пробуждения процесса или разблокировки.

3. **Гонка данных** - ситуация, при которой конечное состояние системы зависит от порядка и интенсивности выполнения потоков, когда потоки не имеют информации друг о друге, но работают с общим ресурсом и хотя бы один из них изменяет этот ресурс.

Для синхронизации потоков, принадлежащих разным процессам, ОС должна предоставлять потокам системные объекты синхронизации. Рассмотрим основные.

1. **Условные переменные** - блокирование одного или нескольких потоков до момента поступления сигнала от другого потока о выполнении некоторого условия или до истечения максимального промежутка времени ожидания. Над переменной можно выполнять две основные операции: «ожидание» и «сигнал». Поток, выполнившая операцию «ожидание», блокируется до того момента, пока другая нить не выполнит операцию «сигнал». Таким образом, операцией «ожидание» первый поток сообщает системе, что он ждет выполнения какого-то условия, а операцией «сигнал» второй поток сообщает первой, что параметры, от которых зависит выполнение условия, возможно, изменились. Основное применение условных переменных – это сценарий «производитель-потребитель».

Рассмотрим две нити, одна из которых генерирует данные, а другая – перерабатывает их. В простейшем случае производитель помещает каждую следующую порцию данных в разделяемую переменную, а потребитель считывает ее оттуда. При этом могут возникать две проблемы. Если производитель работает быстрее потребителя, то он может записать очередную порцию данных до того, как потребитель прочитает предыдущую. При этом предыдущая порция данных будет потеряна. Если же потребитель работает быстрее производителя, он может обработать одну и ту же порцию данных несколько раз. Классическое решение этой задачи реализуется с использованием условной переменной.

2. **Блокировки чтения-записи** можно использовать для реализации стратегии доступа «параллельное чтение и исключающая запись». Имеют два режима захвата: для чтения и для

записи. Блокировку для чтения могут удерживать несколько потоков одновременно. Блокировку для записи может удерживать только один поток. При этом никакой другой поток не может удерживать эту же блокировку для чтения. Используются блокировки для защиты структур данных, которые читают значительно чаще, чем модифицируют.

3. **Барьеры** – используются для синхронизации потоков, выполняющих части одной и той же работы, например, параллельные вычисления и позволяют гарантировать, что даже при неравномерной загрузке системы, потоки будут выполнять равный объем работы в единицу времени. Пусть N - количество потоков, необходимое для перехода через барьер. Потоки, подходящие к барьеру, вызывают функцию «ожидание». Если количество потоков, ожидающих возле барьера, меньше N-1, поток блокируется. Когда набирается N потоков, все они разблокируются и продолжают исполнение. Барьеры полезны для организации коллективных распределенных вычислений в многопроцессорной конфигурации, когда каждый участник (поток управления) выполняет часть работы, а в точке сбора частичные результаты объединяются в общий итог.

4. **Семафор** – представляет собой целочисленную переменную S, над которой определены две операции $P(S)$ и $V(S)$. $V(S)$ – переменная S увеличивается на 1 атомарным действием (выборка, наращивание и запоминание не могут быть прерваны). К переменной S нет доступа во время выполнения этой операции. $P(S)$ – переменная S уменьшается на 1 атомарным действием, если это возможно, оставаясь при этом в области неотрицательных значений. Если S уменьшить невозможно, поток, выполняющий операцию P, ждет, пока это уменьшение станет возможным. Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания (если $S = 0$). Операция V может при некоторых обстоятельствах активизировать процесс, приостановленный операцией P. Иногда семафоры используют в качестве разделяемых целочисленных переменных, например в качестве счетчиков записей в очереди. Использование семафоров обеспечит безопасное использование ресурса только в том случае, если все потоки управления будут захватывать семафор перед использованием ресурса, и освобождать его, как только необходимость в нем отпадет. За взаимосвязь ресурсов и семафоров отвечает прикладная программа. Семафоры также можно использовать для обхода проблемы инверсии приоритета. Если высокоприоритетный и низкоприоритетный потоки должны взаимодействовать, иногда удается реализовать соглашение, что только низкоприоритетный поток может выполнять над семафором операцию уменьшения семафора, а высокоприоритетный поток может делать только операцию увеличения семафора. Такое взаимодействие обычно похоже на схему «производитель-потребитель» с тем отличием, что производитель может терять некоторые порции данных, если потребитель за ним не успевает.

5. **Мьютексы** – представляют собой двоичный семафор. Могут находиться в одном из двух состояний: отмеченном или неотмеченном. Когда поток становится владельцем мьютекса, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным. Мьютексы используются для управления критическими разделами потоков, чтобы предотвратить возникновение условий гонки данных за счет реализации последовательного доступа к критическим секциям.

6. **Критическая секция** – позволяет предотвратить одновременное выполнение некоторого набора операций (обычно связанных с доступом к данным) несколькими потоками. Может использоваться только в пределах одного потока. Выполняет те же задачи, что и мьютекс. Процедура, аналогичная захвату мьютекса, называется входом в критическую секцию. Снятие блокировки мьютекса называется выходом из критической секции. Выполнение обеих операций занимает меньшее время, чем аналогичные операции мьютекса, что связано с отсутствием необходимости обращаться к ядру ОС.

7. **Ожидающие таймеры** – объекты ядра, которые предназначены для отсчета промежутков времени. Они самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Применяется для периодического выполнения задачи. Окончание временного интервала определяется по переходу таймера в свободное состояние. Момент перехода таймера в свободное состояние определяется одной из ожидающих функций.

8. **Спин-блокировки** – низкоуровневое средство синхронизации, предназначенное в для применения в многопроцессорной конфигурации с разделяемой памятью. Аналогично мьютексам могут иметь два значения и реализуются как атомарно устанавливаемое булево значение: истина – блокировка установлена, ложь – блокировка снята. Заметим, что все ожидания в спин-блокировках непрерываемые. При попытке установить спин-блокировку, если она захвачена кем-то другим, как правило, применяется активное ожидание освобождения с постоянным опросом в цикле состояния блокировки. Активное ожидание и установка не связаны с переключением контекстов, активизацией планировщика и т.п. Спин-блокировка устанавливается на очень короткий участок кода и должна быть реализована весьма эффективно, чтобы накладные расходы не оказались высокими. Если ожидание оказывается кратким, минимальными оказываются и накладные расходы. В этом заключает основное преимущество спин-блокировок перед мьютексами.

Рассмотрим, каким образом данные синхронизирующие объекты реализованы в операционных системах.

2. Средства синхронизации потоков в Windows

В ОС Windows существуют следующие средства синхронизации:
1) семафоры (Semaphore);
2) мьютексы (Mutex);
3) события (Event);
4) критические секции (Critical Section) и спин-блокировки;
5) таймеры ожидания (Waitable Timer).

2.1 Семафоры

Основные функции для работы с семафорами это создание семафора ***CreateSemaphore()*** и увеличение счетчика семафора ***ReleaseSemaphore()***:

```
HANDLE CreateSemaphore // создание семафора
// При успешном выполнении вернет идентификатор семафора, иначе NULL
(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,  // атрибут доступа
  LONG lInitialCount, // инициализированное начальное состояние счетчика
  LONG lMaximumCount, // максимальное количество обращений
  LPCTSTR lpName         // имя объекта
);
```

```
BOOL ReleaseSemaphore // увеличение счетчика
// При успешном выполнении возвращаемое значение ненулевое
(
      HANDLE hSemaphore,      // хендл семафора
      LONG lReleaseCount,     // значение инкремента (положительное значение)
      LPLONG lpPreviousCount  // предыдущее значение
);
```

В случае, когда необходимо синхронизовать задачи разных процессов, следует определить имя семафора. При этом один процесс создает семафор с помощью функции CreateSemaphore(), а второй открывает его, получая идентификатор для уже существующего семафора. Открыть существующий семафор можно с помощью функции ***OpenSemaphore()***:

```
HANDLE OpenSemaphore( // открытие семафора
  DWORD   fdwAccess,               // требуемый доступ
  BOOL    fInherit,                // флаг наследования
  LPCTSTR lpszSemaphoreName        // адрес имени семафора
);
```

В API операционной системы Microsoft Windows нет функции, специально предназначенной для уменьшения значения счетчика семафора. Этот счетчик уменьшается, когда задача вызывает функции ожидания, такие как ***WaitForSingleObject()*** или ***WaitForMultipleObject()***. Если задача вызывает несколько раз функцию ожидания для одного и того же семафора, содержимое его счетчика каждый раз будет уменьшаться.

Удаляется семафор также как и все другие объекты ядра функцией **CloseHandle:**

```
BOOL CloseHandle // удалить семафор (как объект)
(
      HANDLE hobj        // передать хендл семафора
);
```

Рассмотрим пример использования семафоров (см. Листинг 1). Запускаются три потока, каждый из которых обращается к семафору. Однако так как значение семафора равно двум, то к нему разрешается обращаться всего двум потокам. Поэтому третий поток будет ждать, пока кто-то освободит семафор. Также обратите внимание на то, что вместо использования CreateThread(), поток создается с помощью функции ***_beginthread(),*** а завершается функцией ***_endthread()*** – это ещё один из возможных способов работы с потоками (стандарт ANSI).

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
#include "process.h"

HANDLE hSemaphore;
LONG cMax = 2;

void Test1(void *);
void Test2(void *);
void Test3(void *);

void main()
{
      hSemaphore = CreateSemaphore(
          NULL,   // нет атрибута
          cMax,   // начальное состояние = 2
          cMax,   // максимальное состояние = 2
          NULL    // семафор без имени
      );

      if (!hSemaphore == NULL)
      {
            if (_beginthread(Test1,1024,NULL)==-1)
```

```
                cout << "Error begin thread " << endl;
            if (_beginthread(Test2,1024,NULL)==-1)
                cout << "Error begin thread " << endl;
            if (_beginthread(Test3,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
            Sleep(10000);
            CloseHandle(hSemaphore);
      }
      else
            cout << "error create semaphore" << endl;
}

void Test1(void *)
{
      cout << "Test1 Running" << endl;
      DWORD dwWaitResult;
      while(dwWaitResult!=WAIT_OBJECT_0)
      {
            dwWaitResult = WaitForSingleObject(
                  hSemaphore, // указатель на семафор
                  1           // интерфал ожидания
            );
            cout << "Test 1 TIMEOUT" << endl;
      }
      Sleep(1000);
      if (ReleaseSemaphore(
                  hSemaphore, // указатель на светофор
                  1,          // изменяет счетчик на 1
              NULL)
      )
      cout << " ReleaseSemaphore Ok Test1" << endl;
      _endthread();
}

void Test2(void *)
{
      cout << "Test2 Running" << endl;
      DWORD dwWaitResult;
      while(dwWaitResult!=WAIT_OBJECT_0)
      {
            dwWaitResult = WaitForSingleObject(hSemaphore,1);
            cout << "Test 2 TIMEOUT" << endl;
      }
      Sleep(1000);
      if (ReleaseSemaphore(hSemaphore,1,NULL))
            cout << " ReleaseSemaphore Ok Test2" << endl;
      _endthread();
}

void Test3(void *)
{
      cout << "Test2 Running" << endl;
      DWORD dwWaitResult;
      while(dwWaitResult!=WAIT_OBJECT_0)
      {
            dwWaitResult = WaitForSingleObject(hSemaphore,1);
            cout << "Test 3 TIMEOUT" << endl;
      }
      if (ReleaseSemaphore(hSemaphore,1,NULL))
            cout << " ReleaseSemaphore Ok Test3" << endl;
      _endthread();
```

```
}
```

Листинг 1 – Пример использования семафоров

2.2 Мьютексы

Для создания мьютекса предназначена функция ***CreateMutex()***, для освобождения ***ReleaseMutex()***. Удалить мьютекс можно с помощью функции ***CloseHandle()***.

```
HANDLE CreateMutex // создать мьютекс
// Вернет дескриптор объекта mutex, а если такое имя есть, то дескрпитор
существующего.
(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,     // атрибут безопасности
  BOOL bInitialOwner,          // флаг начального владельца
  LPCTSTR lpName               // имя объекта
);
```

```
BOOL ReleaseMutex              // освободить мьютекс
(
  HANDLE hMutex                // дескриптор mutex
);
```

В Листинге 2 приведен пример использования мьютекса.

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h“
void main()
{
      HANDLE mut;
      mut = CreateMutex(NULL, FALSE, “Mutex"); // создаем мьютекс
      DWORD result;
      result = WaitForSingleObject(mut,0); // если его удается захватить
      if (result == WAIT_OBJECT_0)
      {     // захватили, выполняем программу
            cout << "program running" << endl;
            int i;
            cin >> i;
            ReleaseMutex(mut); // освобождаем мьютекс
      }
      else  // не удалось захватить
            cout << "fail program running" << endl;
      CloseHandle(mut); // удаляем мьютекс
}
```

Листинг 2 – Пример использования мьютекса

2.3 События

Объект Событие создается с помощью функции ***CreateEvent()***:

```
HANDLE CreateEvent      // создать объект событие
// В случае успеха вернет дескриптор события. Если событие с таким именем уже
создано - вернется дескриптор уже созданного события
(
  LPSECURITY_ATTRIBUTES lpEventAttributes,     // атрибут защиты
  BOOL bManualReset,           // тип сброса TRUE – ручной
```

```
  BOOL bInitialState,         // начальное состояние TRUE - сигнальное
  LPCTSTR lpName        // имя объекта
);
```

Изменить состояние события на сигнальное можно с помощью функции **SetEvent()**:

```
BOOL SetEvent           // изменить состояние на сигнальное
// В случае успеха вернет ненулевое значение
(
      HANDLE hEvent           // дескриптор события
);
```

Изменить состояние события на несигнальное можно с помощью функции **ResetEvent()**:

```
BOOL ResetEvent   // изменить состояние на несигнальное
(
      HANDLE hEvent           // дескриптор события
);
```

В следующем примере создается объект Event, запускается три потока. Каждый поток ждет, когда объект синхронизации перейдет в сигнальное состояние. После некоторой задержки в основной процедуре программы устанавливаем его в сигнальное состояние, выжидаем некоторое время, чтобы потоки среагировали и сбрасываем. Обратите внимание, что в данном случае объектов синхронизации потоков может быть любое количество.

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
#include "process.h"

HANDLE event;

void Test1(void *);
void Test2(void *);
void Test3(void *);

void main()
{
      event=CreateEvent(NULL,TRUE,FALSE,"FirstStep");
      if (_beginthread(Test1,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (_beginthread(Test2,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (_beginthread(Test3,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (event!=NULL){
            Sleep(1000);
            SetEvent(event);
            Sleep(1000);
            ResetEvent(event);
            CloseHandle(event);
      } else {
            cout << "error create event" << endl;
      }
}
```

```
void Test1(void *)
{
	DWORD dwWaitResult;
	while(dwWaitResult!=WAIT_OBJECT_0) {
		dwWaitResult = WaitForSingleObject(event,1);
		cout << "Test 1 TIMEOUT" << endl;
	}
	cout << "Event Test 1 " << endl;
	_endthread();
}

void Test2(void *)
{
	DWORD dwWaitResult;
	while(dwWaitResult!=WAIT_OBJECT_0){
		dwWaitResult = WaitForSingleObject(event,1);
		cout << "Test 2 TIMEOUT" << endl;
	}
	cout << "Event Test 2 " << endl;
	_endthread();
}

void Test3(void *)
{
	DWORD dwWaitResult;
	while(dwWaitResult!=WAIT_OBJECT_0){
		dwWaitResult = WaitForSingleObject(event,1);
		cout << "Test 3 TIMEOUT" << endl;
	}
	cout << "Event Test 3 " << endl;
	_endthread();
}
```

Листинг 3 – Пример использования событий

2.4 Критические секции

Для инициализации критической секции используется функция ***InitializeCriticalSection()***:

```
VOID InitializeCriticalSection     // инициализация кр. секции
(
  LPCRITICAL_SECTION lpCriticalSection // указатель на переменную критическая
секция
);
```

Для обозначения точки входа и выхода в/из критической секции применяются соответственно функции ***EnterCriticalSection()*** и ***LeaveCriticalSection()***:

```
VOID EnterCriticalSection          // объявление начала критической секции
(
  LPCRITICAL_SECTION lpCriticalSection  // указатель на переменную критическая
секция
);
```

```
VOID LeaveCriticalSection          // выход из критической секции
(
```

```
  LPCRITICAL_SECTION lpCriticalSection  // указатель на переменную критическая
секция
);
```

В следующем примере (см. Листинг 4) объект array, над которым можно производить некие действия, объявлен глобально и к нему может обратиться любой из потоков. В случае если количество работающих с этим объектом потоков превышает один, может возникнуть ряд проблем: одна функция не успеет очистить массив, а вторая может уже начать писать или печатать не до конца очищенный массив. Тот код, который правит распределенный ресурс, является критической секцией и выделен соответствующими функциями. Благодаря их использованию потоки дожидаются своей очереди и только тогда выполняют необходимые действия.

```
#include "stdafx.h"
#include "windows.h"
#include "iostream.h"
#include "process.h"

#define MAX_ARRAY 5

CRITICAL_SECTION critsect;

int array[MAX_ARRAY];

void EmptyArray(void *);
void PrintArray(void *);
void FullArray(void *);

void main()
{
      InitializeCriticalSection(&critsect);
      if (_beginthread(EmptyArray,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (_beginthread(PrintArray,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (_beginthread(FullArray,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (_beginthread(PrintArray,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (_beginthread(EmptyArray,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      if (_beginthread(PrintArray,1024,NULL)==-1)
            cout << "Error begin thread " << endl;
      Sleep(10000);
}

void EmptyArray(void *)
{
      cout << "EmptyArray" << endl;
      EnterCriticalSection(&critsect);
      for (int x=0;x<(MAX_ARRAY+1); x++) array[x]=0;
      Sleep(1000);
      LeaveCriticalSection(&critsect);
      _endthread();
}

void PrintArray(void *)
{
      cout << "PrintArray" << endl;
```

```
      EnterCriticalSection(&critsect);
      for (int x=0;x<(MAX_ARRAY+1); x++) cout << array[x] << " ";
      cout << endl;
      Sleep(1000);
      LeaveCriticalSection(&critsect);
      _endthread();
}

void FullArray(void *)
{
      cout << "FullArray" <<
 endl;
      EnterCriticalSection(&critsect);
      for (int x=0;x<(MAX_ARRAY+1); x++) array[x]=x;
      Sleep(1000);
      LeaveCriticalSection(&critsect);
      _endthread();
}
```

Листинг 4 – Пример использования критических секций

Для использования спин-блокировки в критической секции нужно инициализировать счетчик циклов, вызвав функцию ***InitializeCriticalSectionAndSpinCount()***:

```
BOOL WINAPI InitializeCriticalSectionAndSpinCount(
  LPCRITICAL_SECTION lpCriticalSection,  // указатель на переменную критическая
секция
  DWORD dwSpinCount                      // счетчик
);
```

2.5 Ожидающие таймеры

Для создания таймера применяется функция ***CreateWaitableTimer()***:

```
HANDLE CreateWaitableTimer // создает таймер в занятом состоянии, из которого он
выводится принудительно (после создания объект не активен)
(
  LPSECURITY_ATTRIBUTES lpTimerAttributes, // атрибуты таймера
  BOOL bManualReset, // автосброс (запускает один поток) или ручной (запускаются
                  // все потоки, которые его ждали)
  LPCTSTR lpTimerName // имя таймера
);
```

Параметр fManualReset определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом - лишь одного из потоков.

Объекты данного типа всегда создаются в занятом состоянии. Для запуска и настройки таймера используется функция ***SetWaitableTimer():***

```
BOOL SetWaitableTimer // запускает таймер и настраивает. Может быть вызвана в
любой момент, но параметры применятся при перезапуске
(
  HANDLE hTimer,                    // хэндл таймера
  const LARGE_INTEGER *pDueTime,    // время срабатывания таймера
  LONG lPeriod,                     // период повторения (0 - один раз)
```

```
  PTIMERAPCROUTINE pfnCompletionRoutine,  // указатель на асинхр. функцию
  PVOID pvArgToCompletionRoutine,       // параметры асинхронной функции
  BOOL bResume // если != 0, выводит из спящего состояния по срабатыванию
);
```

hTimer определяет нужный таймер. Следующие два параметра (pDueTime и lPeriod) используются совместно. Первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем.

В качестве четвертого и пятого параметров указывается некая асинхронная функция и её параметры, которые нужно реализовать отдельно. Функция должна выглядеть следующим образом:

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompleUonRoutine, DWORD
dwTimerLowValue, DWORD dwTimerHighValue)
{
  // здесь делаем то, что нужно
}
```

Последний параметр функции lResume полезен на компьютерах с поддержкой режима сна. Обычно в нем передают FALSE, но если передать TRUE, то когда таймер сработает, машина выйдет из режима сна (если она находилась в нем), и пробудятся потоки, ожидавшие этот таймер.

Функция ***CancelWaitableTimer()*** останавливает таймер и отменяет выполнение асинхронной функции, не изменяя состояния таймера.

```
BOOL CancelWaitableTimer(HANDLE hTimer)// остановка таймера
```

3. Средства синхронизации потоков в Linux/Unix

В ОС Linux/Unix существуют следующие средства синхронизации:
1) семафоры;
2) мьютексы;
3) условные переменные;
4) блокировки чтения-записи;
5) барьеры;
6) спин-блокировки.

Ниже рассмотрим основные функции для работы с синхронизирующими объектами POSIX. Для получения полного списка функций по каждому объекту рекомендуется воспользоваться дополнительной справочной литературой и специализированными сайтами. В частности, не рассматриваются варианты функций синхронизации с контролем времени, функции для работы с атрибутами объектов и др.

3.1 Семафоры

В Unix-системах реализованы три типа **семафоров** – семафоры System V, семафоры POSIX и семафоры в разделяемой памяти. Все объявления функций и типов, относящиеся к POSIX-семафорам, можно найти в файле /usr/include/nptl/semaphore.h.

Семафоры бывают двух типов – именованные и неименованные. Те и другие семафоры хранятся в переменных типа sem_t, но процедура инициализации и уничтожения этих переменных отличается.

Неименованные семафоры инициализируются функцией ***sem_init()***.

```
#include <semaphore.h>
int sem_init(
  sem_t *sem,           // инициализируемый семафор
  int pshared,          // 0 если семафор будет локальным в пределах процесса,
                        // ненулевое значение - если семафор будет разделяемым
                        // между процессами
  unsigned int value    // начальное значение флаговой переменной семафора
);
```

После работы семафор необходимо уничтожить функцией ***sem_destroy()***:

```
#include <semaphore.h>
int sem_destroy(
  sem_t *sem            // уничтожаемый семафор
);
```

Дальнейшая работа с семафором осуществляется с помощью функций ***sem_wait()***, ***sem_trywait()*** и ***sem_post()***:

```
#include <semaphore.h>
int sem_wait(
  sem_t *sem            // указатель на идентификатор семафора
);

int sem_trywait(
  sem_t *sem            // указатель на идентификатор семафора
);

int sem_post(
  sem_t *sem            // указатель на идентификатор семафора
);
```

Эти функции используют указатель на идентификатор семафора, созданного функцией sem_init() и оперируют его значением value. Функция sem_wait() приостанавливает выполнение вызвавшего ее потока до тех пор, пока значение семафора не станет больше нуля, после чего функция уменьшает значение семафора на единицу и возвращает управление. Функция sem_post() увеличивает значение семафора, идентификатор которого был передан ей в качестве параметра, на единицу. Функция sem_trywait() аналогична функции sem_wait(), но если семафор не может быть захвачен, то функция sem_trywait() не устанавливает поток управления в очередь, а возвращает управление с кодом ошибки «семафор не захвачен». В очереди к семафору может находиться одновременно несколько потоков управления. Потоки управления, находящиеся в очереди, упорядочены по приоритетам, а потоки, имеющие равный приоритет, упорядочены по времени установки в очередь.

Получить значение семафора можно функцией ***sem_getvalue()***.

```
#include <semaphore.h>
int sem_getvalue(sem_t *sem, int *sval);
```

Функция получает значение семафора в некоторый неопределенный момент времени. В интервале между исполнением sem_getvalue() и проверкой значения флаговая переменная

семафора может измениться, поэтому тот факт, что sem_getvalue() вернул ненулевое значение, не означает, что вызов sem_wait() с этим семафором не будет заблокирован.

Именованные семафоры создаются функцией ***sem_open()***. С её же помощью можно получить доступ к уже существующему именованному семафору. Если процесс попытается несколько раз открыть один и тот же семафор, ему будут возвращать один и тот же указатель.

```
#include <fcntl.h>   /* For O_* constants */
#include <sys/stat.h>  /* For mode constants */
#include <semaphore.h>

sem_t *sem_open(
  const char *name,           // имя семафора
  int oflag,                  // флаги (0, O_CREAT и O_CREAT|O_EXC)
  mode_t mode,                // необязательный, используется, только если flags
                              // содержит бит O_CREAT
  unsigned int value          // необязательный, используется только если flags
                              // содержит бит O_CREAT
);
```

Именованные семафоры всегда разделяемые между процессами. При доступе к существующему семафору действует схема проверки прав, аналогичная проверке прав к файлам. Для доступа к семафору процесс должен иметь права чтения и записи.

Для отсоединения от семафора и освобождения памяти используется функция ***sem_close()***:

```
#include <semaphore.h>
int sem_close(
  sem_t *sem                  // указатель на идентификатор семафора
);
```

Следует помнить, что закрытие именованного семафора процессом не прекращает существования семафора. Чтобы удалить семафор, необходимо вызвать функцию ***sem_unlink()***. Это лишит новые процессы возможность видеть семафор как существующий и позволит создать новый семафор с тем же именем. Если с семафором на момент вызова функции работали потоки, то семафор останется в рабочем состоянии, пока все процессы не выполнят sem_close().

Во всем остальном именованный семафор не отличается от неименованного. Над ним можно выполнять те же операции, что и над неименованным, при помощи тех же функций.

Рассмотрим пример (см. Листинг 5). В нем связанный процесс обращается к общему участку памяти, полученному в результате отображения файла на память.

```
#include <semaphore.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc, char **argv)
{
```

```
  int fd, i,count=0,nloop=10,zero=0,*ptr;
  sem_t semaphore;

  //открываем файл и отображаем его в память

  fd = open("log.txt",O_RDWR|O_CREAT,S_IRWXU);
  write(fd,&zero,sizeof(int));
  ptr = mmap(NULL,sizeof(int),PROT_READ |PROT_WRITE,MAP_SHARED,fd,0);
  close(fd);


  /* создаем семафор */
  if ((semaphore = sem_open("/mysemaphore", O_CREAT, 0644, 1)) == SEM_FAILED) {
    perror("semaphore initilization");
    exit(0);
  }

  if (fork() == 0) { /* child process*/
    for (i = 0; i < nloop; i++) {
      sem_wait(&semaphore);
      printf("child: %d\n", (*ptr)++);
      sem_post(&mutex);
    }
    exit(0);
  }
  /* возвращаемся к родительскому процессу */
  for (i = 0; i < nloop; i++) {
    sem_wait(&semaphore);
    printf("parent: %d\n", (*ptr)++);
    sem_post(&semaphore);
  }
  exit(0);
}
```

Листинг 5 – Пример использования семафора для синхронизации процессов

Для отображения файла в память используется функция ***mmap()***:

```
#include<sys/mman.h>
void *mmap(
  void *addr,  // адрес начала участка отображенной памяти
  size_t len,   // количество байт, которое нужно отобразить в память
  int prot,       // степень защищённости отображенного участка памяти
  int flag,       // атрибуты области
  int filedes,   // дескриптор файла, который нужно отобразить
  off_t off       // смещение отображенного участка от начала файла
)
```

Возвращает адрес начала участка отображаемой памяти или MAP_FAILED в случае неудачи.

3.2 Мьютексы

Все функции и типы данных, имеющие отношение к мьютексам, определены в файле pthread.h. Мьютекс создается вызовом функции ***pthread_mutex_init()***.Перед освобождением памяти из-под мьютекса его необходимо уничтожить функцией ***pthread_mutex_destroy()***. Уничтожение мьютекса без выполнения pthread_mutex_destroy() может приводить к утечке памяти или исчерпанию системных ресурсов. Выполнение операции pthread_mutex_destroy

над мьютексом, на котором заблокирован один или более потоков, приводит к неопределенным последствиям.

```
#include <pthread.h>
int pthread_mutex_init(
  pthread_mutex_t *mutex,          // идентификатор нового мьютекса
  const pthread_mutexattr_t *attr  // атрибуты мьютекса. NULL - для обычного
);

int pthread_mutex_destroy(
  pthread_mutex_t *mutex           // идентификатор уничтожаемого мьютекса
);
```

Для того чтобы получить исключительный доступ к глобальному ресурсу (захватить мьютекс), поток вызывает функцию ***pthread_mutex_lock()***. Закончив работу с глобальным ресурсом, поток освобождает мьютекс с помощью функции ***pthread_mutex_unlock()***.

```
#include <pthread.h>
int pthread_mutex_lock(
  pthread_mutex_t *mutex           // идентификатор нового мьютекса
);
int pthread_mutex_trylock(
  pthread_mutex_t *mutex           // идентификатор нового мьютекса
);
int pthread_mutex_unlock(
  pthread_mutex_t *mutex           // идентификатор нового мьютекса
);
```

Если поток вызовет функцию pthread_mutex_lock() для мьютекса, уже захваченного другим потоком, эта функция не вернет управление до тех пор, пока другой поток не освободит мьютекс с помощью вызова pthread_mutex_unlock(). Функция ***pthread_mutex_trylock()*** выполняет те же действия, что и pthread_mutex_lock(), но в случае, если мьютекс занят другим процессом возвращает ошибку.

По умолчанию операции над мьютексами не осуществляют никаких проверок. При повторном захвате мьютекса той же нитью произойдет взаимная блокировка (тупик). Используя pthread_attr_t, при инициализации мьютекса можно задать параметры, которые заставят систему делать проверки при работе с мьютексами и возвращать ошибки при некорректных последовательностях операций, но это повлечет дополнительную нагрузку на операционную систему.

Простейший пример работы с мьютексом приведен ниже (см. листинг 6).

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

pthread_mutex_t m;
// глобальная переменная
int count=0;

void* inc_count (void* arg)
{
  pthread_mutex_lock(&m); // сначала захватываем мьютекс
  // выполняем действие над
  // глобальной переменной
```

```
  count = count + 1;
  printf("[%d]",count);
  pthread_mutex_unlock(&m); // отпускаем мьютекс
}
int main()
{
  // создаем мьютекс
  pthread_mutex_init(&m,NULL);
  pthread_t t1, t2;

  inc_count(NULL);

  pthread_create(&t1,NULL,inc_count,NULL); // создаем поток 1

  pthread_create(&t2,NULL,inc_count,NULL); // создаем поток 2

  // ожидаем завершение потоков
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
  return 0;
}
```

Листинг 6 – Пример использования мьютекса

3.3 Условные переменные

Процедура создания и уничтожения условных переменных в целом аналогична процедуре создания и уничтожения ранее рассматривавшихся объектов синхронизации. Для создания переменной служит функция ***pthread_cond_init()***, для удаления ***pthread_cond_destroy()***:

```
#include <pthread.h>

int pthread_cond_init(
  pthread_cond_t *cond,            // идентификатор новой условной переменной
  const pthread_condattr_t *attr   // атрибуты переменной
);

int pthread_cond_destroy(
  pthread_cond_t *cond       // идентификатор удаляемой условной переменной
);
```

Как мы помним, над условной переменной определены две основные операции: «ожидание» и «сигнал». Первая функция реализуется с помощью ***pthread_cond_wait()***, вторая – с помощью ***pthread_cond_signal()***:

```
#include <pthread.h>
int pthread_cond_wait(
  pthread_cond_t *cond,            // идентификатор условной переменной
  pthread_mutex_t *mutex           // идентификатор мьютекса
);

int pthread_cond_signal(
  pthread_cond_t *cond             // идентификатор условной переменной
);
```

При вызове pthread_cond_wait() мьютекс, переданный в качестве второго параметра, должен быть захвачен, в противном случае результат не определен. Wait освобождает мьютекс и блокирует поток до момента вызова другим потоком pthread_cond_signal(). После пробуждения wait пытается захватить мьютекс. Если это не получается, он блокируется до того момента, пока мьютекс не освободят. Мьютекс используется для защиты данных, используемых при вычислении условия, с которым связана условная переменная. Условие необходимо проверять как перед вызовом pthread_cond_wait(), так и после выхода из этой функции. Проверка условия перед вызовом позволяет защититься от ситуации, когда производитель вызвал signal в то время, когда потребитель еще не был заблокирован в wait. Повторная проверка условия необходима на случай, когда производителей или потребителей несколько и между ними возникает конкуренция.

Ниже приведен пример использования условной переменной (см. Листинг 7)

```
#define _MULTI_THREADED
#include <pthread.h>

#include <stdio.h>

int                 conditionMet = 0;
pthread_cond_t      cond  = PTHREAD_COND_INITIALIZER; // условная переменная
pthread_mutex_t     mutex = PTHREAD_MUTEX_INITIALIZER; // мьютекс

#define NTHREADS    5

/* Функция проверяет код возврата и завершает программу, если функция не была
выполнена
*/
static void compResults(char *string, int rc) {
  if (rc) {
    printf("Ошибка в : %s, rc=%d",
           string, rc);
    exit(EXIT_FAILURE);
  }
  return;
}

void *threadfunc(void *parm) // функция потока
{
  int rc;

  rc = pthread_mutex_lock(&mutex);
  checkResults("pthread_mutex_lock()\n", rc);

  while (!conditionMet) {
    printf("Thread blocked\n");
    rc = pthread_cond_wait(&cond, &mutex);
    checkResults("pthread_cond_wait()\n", rc);
  }

  rc = pthread_mutex_unlock(&mutex);
  checkResults("pthread_mutex_lock()\n", rc);
  return NULL;
}

int main(int argc, char **argv)
{
  int                   rc=0;
  int                   i;
```

```
  pthread_t             threadid[NTHREADS];

  printf("Enter Testcase - %s\n", argv[0]);

  printf("Create %d threads\n", NTHREADS);
  for(i=0; i<NTHREADS; ++i) {
    rc = pthread_create(&threadid[i], NULL, threadfunc, NULL);
    checkResults("pthread_create()\n", rc);
  }

  sleep(5);
  rc = pthread_mutex_lock(&mutex);
  checkResults("pthread_mutex_lock()\n", rc);

  /* Условие выполнилось. Устанавливаем флаг и будим все ожидающие потоки */
  conditionMet = 1;
  printf("Wake up all waiting threads...\n");
  rc = pthread_cond_broadcast(&cond);
  checkResults("pthread_cond_broadcast()\n", rc);

  rc = pthread_mutex_unlock(&mutex);
  checkResults("pthread_mutex_unlock()\n", rc);

  printf("Wait for threads and cleanup\n");
  for (i=0; i<NTHREADS; ++i) {
    rc = pthread_join(threadid[i], NULL);
    checkResults("pthread_join()\n", rc);
  }
  pthread_cond_destroy(&cond);
  pthread_mutex_destroy(&mutex);

  printf("Main completed\n");
  return 0;
}
```

Листинг 7 – Пример использования условной переменной

Обратите внимание, мы использовали ещё одну функцию ***pthread_cond_broadcast()*** в этой программе. Это широковещательный аналог функции pthread_cond_signal().

3.4 Блокировки чтения-записи

API для работы с блокировками чтения-записи в целом похож на API для работы с мьютексами и включает в себя следующие основные функции.

Создание блокировки происходит с помощью функции ***pthread_rwlock_init()***, удаление - ***pthread_rwlock_destroy()***:

```
#include <pthread.h>
int pthread_rwlock_init(
  pthread_rwlock_t *rwlock,              // идентификатор блокировки
  const pthread_rwlockattr_t *attr       // атрибуты блокировки
);
int pthread_rwlock_destroy(
  pthread_rwlock_t *rwlock         // идентификатор уничтожаемой блокировки
);
```

Функция pthread_rwlock_init() выделяет ресурсы, необходимые для использования объекта блокировки чтения-записи, адресуемого параметром rwlock, и инициализирует (он переходит в незаблокированное состояние) с использованием объекта атрибутов,

адресуемого параметром attr. Если параметр attr содержит значение NULL, для блокировки чтения-записи будут использованы атрибуты, действующие по умолчанию.

Для блокировки чтения используются функции ***pthread_rwlock_rdlock()*** и ***pthread_rwlock_tryrdlock()***.

```
#include <pthread.h>

int pthread_rwlock_rdlock(
  pthread_rwlock_t *rwlock          // идентификатор блокировки
);
int pthread_rwlock_tryrdlock(
  pthread_rwlock_t *rwlock          // идентификатор блокировки
);
```

Для блокировки записи применяются функции ***pthread_rwlock_wrlock()*** и ***pthread_rwlock_trywrlock()***:

```
#include <pthread.h>

int pthread_rwlock_wrlock(
  pthread_rwlock_t *rwlock          // идентификатор блокировки
);
int pthread_rwlock_trywrlock(
  pthread_rwlock_t *rwlock          // идентификатор блокировки
);
```

Варианты ***try*** блокировок выбрасывают исключения, если данная блокировка уже захвачена каким-либо из потоков.

Для снятия блокировки применяется функция ***pthread_rwlock_unlock()***:

```
#include <pthread.h>
int pthread_rwlock_unlock(
  pthread_rwlock_t *rwlock          // идентификатор блокировки
);
```

Если функция вызывается, чтобы освободить блокировку чтения, и существуют другие блокировки чтения, удерживаемые в данный момент по этому объекту блокировки чтения-записи, то он (объект) останется в состоянии блокирования для обеспечения чтения. Если с помощью этой функции освобождается последняя блокировка для чтения по заданному объекту блокировки чтения-записи, то этот объект перейдет в разблокированное состояние и, соответственно, не будет иметь владельцев.

Если эта функция вызывается, чтобы освободить блокировку для обеспечения записи по заданному объекту блокировки чтения-записи, то этот объект перейдет в разблокированное состояние.

Рассмотрим пример использования блокировок (см. Листинг 8). В ней показано как использовать функцию pthread_rwlock_rdlock() для захвата блокировки чтения-записи для чтения.

```
#define _MULTI_THREADED
#include pthread.h
#include stdio.h

pthread_rwlock_t       rwlock;
```

```
static void compResults(char *string, int rc) {
  if (rc) {
    printf("Ошибка в : %s, rc=%d",
           string, rc);
    exit(EXIT_FAILURE);
  }
  return;
}

void *rdlockThread(void *arg)
{
  int rc;

  printf("Выполняется нить, получение блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock()\n", rc);
  printf("блокировка rwlock захвачена для чтения\n");

  sleep(5);

  printf("освобождение блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);
  printf("Дополнительная нить разблокировала\n");
  return NULL;
}

void *wrlockThread(void *arg)
{
  int rc;

  printf("Выполняется нить, получение блокировки для записи\n");
  rc = pthread_rwlock_wrlock(&rwlock);
  compResults("pthread_rwlock_wrlock()\n", rc);

  printf("Блокировка rwlock захвачена для записи, освобождение блокировки\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);
  printf("Дополнительная нить разблокирована\n");
  return NULL;
}

int main(int argc, char **argv)
{
  int                   rc=0;
  pthread_t             thread, thread1;

  printf("Запуск теста - %s\n", argv[0]);

  printf("Главная нить, инициализация блокировки чтения-записи\n");
  rc = pthread_rwlock_init(&rwlock, NULL);
  compResults("pthread_rwlock_init()\n", rc);

  printf("Главная нить, захват блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock()\n",rc);

  printf("Главная нить, повторный захват этой же блокировки для чтения\n");
  rc = pthread_rwlock_rdlock(&rwlock);
  compResults("pthread_rwlock_rdlock() second\n", rc);
```

```
  printf("Главная нить, создание нити для захвата блокировки для чтения\n");
  rc = pthread_create(&thread, NULL, rdlockThread, NULL);
  compResults("pthread_create\n", rc);

  printf("Главная нить - освобождение первой блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Главная нить, создание нити для захвата блокировки для записи\n");
  rc = pthread_create(&thread1, NULL, wrlockThread, NULL);
  compResults("pthread_create\n", rc);

  sleep(5);
  printf("Главная нить - освобождение второй блокировки для чтения\n");
  rc = pthread_rwlock_unlock(&rwlock);
  compResults("pthread_rwlock_unlock()\n", rc);

  printf("Главная нить, ожидание завершения нитей\n");
  rc = pthread_join(thread, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_join(thread1, NULL);
  compResults("pthread_join\n", rc);

  rc = pthread_rwlock_destroy(&rwlock);
  compResults("pthread_rwlock_destroy()\n", rc);

  printf("Главная нить завершена\n");
  return 0;
}
```

Листинг 8 – Пример использования блокировок чтения-записи

3.5 Барьеры

Инициализация и разрушение барьеров происходит с помощью функций ***pthread_barrier_init()***, ***pthread_barrier_destroy()***:

```
#include <pthread.h>
int pthread_barrier_init (
  pthread_barrier_t *restrict barrier,         // идентификатор барьера
  const pthread_barrierattr_t *restrict attr,  // атрибуты барьера
  unsigned count       // количество синхронизируемых потоков управления
);

int pthread_barrier_destroy (
  pthread_barrier_t *barrier
);
```

Аргумент count в функции инициализации барьера задает количество синхронизируемых потоков управления. Столько потоков должны вызвать функцию синхронизации на барьере ***pthread_barrier_wait()***, прежде чем каждый из них сможет успешно завершить вызов и продолжить выполнение:

```
#include <pthread.h>
int pthread_barrier_wait (
  pthread_barrier_t *barrier             // идентификатор барьера
);
```

Когда к функции pthread_barrier_wait() обратилось требуемое число потоков управления, одному из них в качестве результата возвращается именованная константа PTHREAD_BARRIER_SERIAL_THREAD, а всем другим выдаются нули. После этого барьер возвращается в начальное (инициализированное) состояние, а выделенный поток может выполнить соответствующие объединительные действия.

Пример программы, использующий барьеры приведен в Листинге 9. Барьеры используются для слияния результатов вычислений логарифма по основанию 2, при этом используется два потока. Первый считает ряд для положительных значений, второй – для отрицательных.

```
#define _XOPEN_SOURCE 600

#include <pthread.h>
#include <stdio.h>
#include <errno.h>

static pthread_barrier_t mbrr;

static double sums [2] = {0,0}; // два потока «+» и «-»

static void *start_func (void *ns)
{
    double d = 1;
    double s = 0;
    int i;

    for (i = (int) ns; i <= 100000000; i += 2)
    {
        s += d / i;
    }

    sums [(int) ns - 1] = s;

    /* Синхронизируемся для получения общего итога */
    if (pthread_barrier_wait (&mbrr) == PTHREAD_BARRIER_SERIAL_THREAD)
    {
        sums [0] -= sums [1];
    }
    return (sums); // указатель на итог
}

int main (void)
{
    pthread_t pt1, pt2;
    double *pd;

    if ((errno = pthread_barrier_init(&mbrr, NULL, 2)) != 0)
    {
        perror ("PTHREAD_BARRIER_INIT");
        return (errno);
    }

    pthread_create (&pt1, NULL, start_func, (void *) 1);
    pthread_create (&pt2, NULL, start_func, (void *) 2);

    pthread_join (pt1,(void **) &pd);
    pthread_join (pt2, (void **) &pd);
```

```
    printf ("Совместно вычисленное значение ln(2): %g\n", *pd);

    return (pthread_barrier_destroy(&mbrr));
}
```

Листинг 9 - Пример программы, использующей барьеры

3.6 Спин-блокировки

Для использования примитивов спин-блокировки необходимо подключить файл <linux/spinlock.h>. Фактическая блокировка имеет тип spinlock_t. Для инициализации спин-блокировки используется функция ***pthread_spin_init()***, для разрушения используется функция ***pthread_spin_destroy()***.

```
#include <pthread.h>

int pthread_spin_init (
  pthread_spinlock_t *lock,   // идентификатор блокировки
  int pshared
);

int pthread_spin_destroy (
  pthread_spinlock_t *lock    // идентификатор блокировки
);
```

Перед входом в критическую секцию необходимо установить блокировку с помощью функций ***pthread_spin_lock()*** или ***pthread_spin_trylock***, для освобождения полученной блокировки нужно передать её в функцию ***pthread_spin_unlock()***:

```
#include <pthread.h>

int pthread_spin_lock (
  pthread_spinlock_t *lock);

int pthread_spin_trylock (
  pthread_spinlock_t *lock);

int pthread_spin_unlock (
  pthread_spinlock_t *lock);
```

Посмотрим на сколько спин-блокировки работают быстрее, чем мьютексы. В листинге программы 10 используются директивы препроцессора #ifdef, #endif, позволяющие указать дополнительную опцию при компиляции программы. Если программа скомпилирована с опцией USE_SPINLOCK используются спин-блокировки, иначе – обычные мьютексы. Программа создает список целых чисел и два потока, которые эти числа из списка удаляют, а средства синхронизации применяются для координации действий потоков. По окончании программы выводится информация о времени, прошедшем с момента создания потоков до их окончания.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>
#include <sys/time.h>
```

```
#include <list>

#define LOOPS 10000000

using namespace std;

list<int> the_list;

#ifdef USE_SPINLOCK // используем директивы препроцессора
pthread_spinlock_t spinlock;
#else
pthread_mutex_t mutex;
#endif

pid_t gettid() { return syscall( __NR_gettid ); }

void *consumer(void *ptr)
{
    int i;
    printf("Consumer TID %lu\n", (unsigned long)gettid());
    while (1)
    {
#ifdef USE_SPINLOCK
        pthread_spin_lock(&spinlock);
#else
        pthread_mutex_lock(&mutex);
#endif

        if (the_list.empty())
        {
#ifdef USE_SPINLOCK
            pthread_spin_unlock(&spinlock);
#else
            pthread_mutex_unlock(&mutex);
#endif
            break;
        }

        i = the_list.front();
        the_list.pop_front();

#ifdef USE_SPINLOCK
        pthread_spin_unlock(&spinlock);
#else
        pthread_mutex_unlock(&mutex);
#endif
    }

    return NULL;
}

int main()
{
    int i;
    pthread_t thr1, thr2;
    struct timeval tv1, tv2;

#ifdef USE_SPINLOCK
    pthread_spin_init(&spinlock, 0);
#else
```

```
    pthread_mutex_init(&mutex, NULL);
#endif

    // Creating the list content...
    for (i = 0; i < LOOPS; i++)
        the_list.push_back(i);

    // Оценка времени между стартом потоков
    gettimeofday(&tv1, NULL);

    pthread_create(&thr1, NULL, consumer, NULL);
    pthread_create(&thr2, NULL, consumer, NULL);

    pthread_join(thr1, NULL);
    pthread_join(thr2, NULL);

    // Оценка времени после выполнения потоков
    gettimeofday(&tv2, NULL);

    if (tv1.tv_usec > tv2.tv_usec)
    {
        tv2.tv_sec--;
        tv2.tv_usec += 1000000;
    }

    printf("Result - %ld.%ld\n", tv2.tv_sec - tv1.tv_sec,
        tv2.tv_usec - tv1.tv_usec);

#ifdef USE_SPINLOCK
    pthread_spin_destroy(&spinlock);
#else
    pthread_mutex_destroy(&mutex);
#endif

    return 0;
}
```

Листинг 10 – Программа, сравнивающая скорость работы мьютексов и спин-блокировок

Скопилируйте программу два раза (с опцией USE_SPINLOCK и без неё) и сравните результаты:

```
[user@host:~]$  g++ -Wall -pthread main.cc
[user@host:~]$ ./a.out
[user@host:~]$  g++ -DUSE_SPINLOCK -Wall -pthread main.cc
[user@host:~]$ ./a.out
```

Задание

1. Изучить краткие теоретические сведения и лекционный материал по теме практического задания.

2. Используя Docker и соответствующий образ подготовить среду для разработки (уже готовы если сделаны предыдущие лабораторные работы).

3. Предусмотреть в варианте задания использование разделяемого ресурса. Описать какие из средств синхронизации и как могут быть применены для решения новой задачи.

4. Реализовать один (или несколько) вариантов синхронизации на языке C++ для Linux (ваш вариант образа).

5. Реализовать один (или несколько) вариантов синхронизации средствами встроенных высокоуровневых возможностей языка программирования.

6. Сравнить возможности обоих подходов, сделать выводы.

Варианты индивидуальных заданий

Варианты заданий (образ ОС для Docker)
1. CentOS
2. Ubuntu
3. Debian
4. Alpine
5. Amazon Linux

Варианты заданий (Язык программирования)
1. c++ (gcc)
2. golang
3. python
4. PHP
5. Java

Варианты заданий (программа)

1. Написать две программы. Первая реализует алгоритм поиска простых чисел в некотором интервале. Вторая - разбивает заданный интервал на диапазоны, осуществляет поиск простых чисел в каждом из интервалов в отдельном процессе, выводит общий результат.

2. Написать две программы. Первая реализует алгоритм поиска указанной подстроки в строке. Вторая - разбивает входной файл на фрагменты, осуществляет поиск подстроки в каждом из фрагментов в отдельном процессе, выводит общий результат.

3. Написать две программы. Первая реализует алгоритм умножения двух векторов произвольной длины. Вторая - умножает матрицу произвольного порядка на вектор, при этом умножение каждой строки на вектор производить в отдельном процессе.

4. Написать две программы, одна из которых осуществляет проверку доступности узла сети на доступность (команда ping), а вторая - осуществляет проверку доступности диапазона IP-адресов сети класса «С» (254 адреса, маска 255.255.255.0), разделенного на несколько поддиапазонов, с помощью первой программы.

5. Координаты заданного количества шариков изменяются на случайную величину по вертикали и горизонтали, при выпадении шарика за нижнюю границу допустимой области шарик исчезает. Напишите программу изменения координат одного шарика и программу, создающую для каждого из заданного количества шариков порожденный процесс изменения их координат.

6. Противостояние двух команд – каждая команда увеличивается на случайное количество бойцов и убивает случайное количество бойцов участника. Напишите программу, которая бы осуществляла уменьшение числа бойцов в противостоящей команде и увеличение в своей на случайную величину и программу, в которой бы в родительском процессе запускались порожденные процессы, реализующие деятельность одной команды.

7. Написать две программы. Первая - копирует файл. Вторая - копирует содержимое директории пофайлово с помощью первой программы в отдельных процессах.

8. Написать две программы.Первая – вычисляет контрольную сумму файла. Вторая – вычисляет контрольную сумму всех файлов в директории, при этом обработка каждого отдельного файла осуществляется с помощью первой программы в отдельном процессе.

9. Написать две программы. Первая – вычисляет математическое ожидание и дисперсию в массиве данных. Вторая – вычисляет математическое ожидание и дисперсию в нескольких массивах данных, при этом обработка каждого массива осуществляется отдельно с помощью первой программы в новом процессе.

10. Написать две программы. Первая – вычисляет частоты встречаемости в тексте биграмм символов (аа, аб, ав, … яэ, яю, яя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм символов, путем вызова первой программы для отдельных фрагментов текста.

11. Написать две программы. Первая – вычисляет частоты встречаемости в тексте триграмм символов (ааа, ааб, аав, … яяэ, яяю, яяя). Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм символов, путем вызова первой программы для отдельных фрагментов текста.

12. Написать две программы. Первая – вычисляет частоты встречаемости в тексте биграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет», «вычисляет частоты», «частоты встречаемости», «встречаемости в» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте биграмм слов, путем вызова первой программы для отдельных фрагментов текста.

13. Написать две программы. Первая – вычисляет частоты встречаемости в тексте триграмм слов. Например, для предыдущего предложения биграммы слов это пары «первая вычисляет частоты», «вычисляет частоты встречаемости», «частоты встречаемости в», «встречаемости в тексте» и т.д. Вторая – принимает на вход текст, делит его на отдельные фрагменты и вычисляет частоты встречаемости в тексте триграмм слов, путем вызова первой программы для отдельных фрагментов текста.

14. Медведь и пчелы. Заданное количество пчел добывают мед равными порциями, задерживаясь в пути на случайное время. Медведь потребляет мед порциями заданной величины за заданное время и столько же времени может прожить без питания. Написать две программы. Первая - реализует работу одной пчелы. Вторая - осуществляет работу медведя, при этом у для заданного количества пчел вызывается отдельный процесс работы одной пчелы.

15. Авиаразведка - создается условная карта в виде матрицы, размерность которой определяет размер карты, содержащей произвольное количество единиц (целей) в произвольных ячейках. Из произвольной точки карты стартуют несколько разведчиков (процессов), курсы которых выбираются так, чтобы покрыть максимальную площадь карты. Каждый разведчик фиксирует цели, чьи координаты совпадают с его координатами и по достижении границ карты сообщает количество обнаруженных целей. Реализуйте соответствующие программы, используя механизмы процессов.

16. Бег с препятствиями - создается условная карта трассы в виде матрицы, ширина которой соответствует количеству бегунов, а высота – фиксирована, содержащей произвольное количество единиц (препятствий) в произвольных ячейках. Стартующие бегуны (процессы) перемещаются по трассе и при встрече с препятствием задерживаются на фиксированное время. По достижении финиша бегуны сообщают свой номер. Реализуйте соответствующие программы, используя механизмы процессов.

Контрольные вопросы

1. Что такое взаимоблокировка? Какие стратегии борьбы с взаимоблокировками Вам известны?

2. Что собой представляет бесконечная отсрочка? Какие стратегии борьбы с бесконечной отсрочкой Вам известны?

3. Что такое гонка данных? Какие стратегии борьбы с гонкой данных Вам известны?

4. Что такое семафор? Каких проблем синхронизации позволяет избежать семафор?

5. Какие типы семафоров есть в ОС Linux/Unix? Чем отличаются именованные семафоры от неименованных?

6. Что такое мьютекс? Каких проблем синхронизации позволяет избежать мьютекс?

7. Что такое условная переменная? Каких проблем синхронизации позволяет избежать условная переменная?

8. Что такое критическая секция? Каких проблем синхронизации позволяет избежать условная переменная?

9. Что такое блокировка чтения-записи? Каких проблем синхронизации позволяет избежать блокировка чтения-записи?

10. Что такое барьер? Каких проблем синхронизации позволяет избежать барьер?

11. Что такое спин-блокировка? Каких проблем синхронизации позволяет избежать спин-блокировка?

12. Что такое ожидающий таймер и для чего он используется?

13. Какие средства синхронизации существуют в ОС Windows?

14. Какие средства синхронизации существуют в ОС Linux/Unix?

15. Зачем нужны директивы препроцессора? Какие директивы препроцессора Вы знаете?

16. Что такое отображения файлов? Какие функции существуют для отображения файла на память?

