ВВЕДЕНИЕ В МУЛЬТИПРОГРАММИРОВАНИЕ

ЛЕКЦИЯ 13-14

Мультипрограммирование

Основные понятия

- **Мультипроцессирование** - использование нескольких процессоров для одновременного выполнения задач.
- **Мультипрограммирование** - одновременное выполнение нескольких задач на одном или нескольких процессорах.

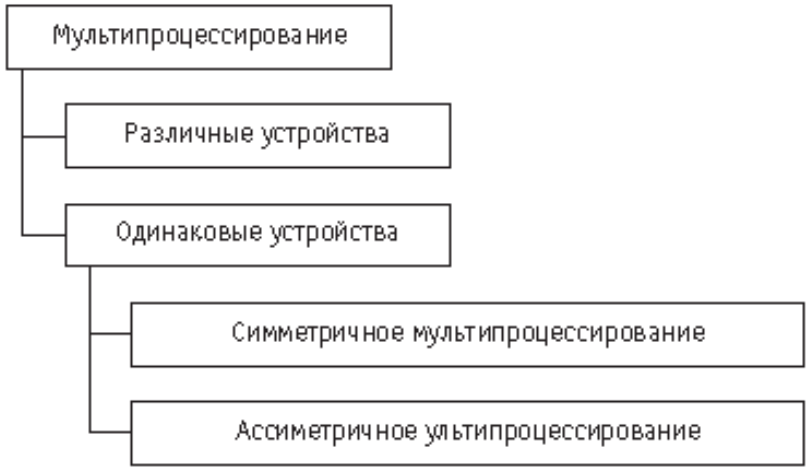
Мультипроцессирование

Мультипроцессирование бывает нескольких видов:

- когда используются различные виды оборудования, например, одновременная работа центрального процессора и графического ускорителя видеокарты;
- когда организуется одновременная работа равноправных устройств, выполняющих сходные задачи.

Во втором случае бывает два подхода:

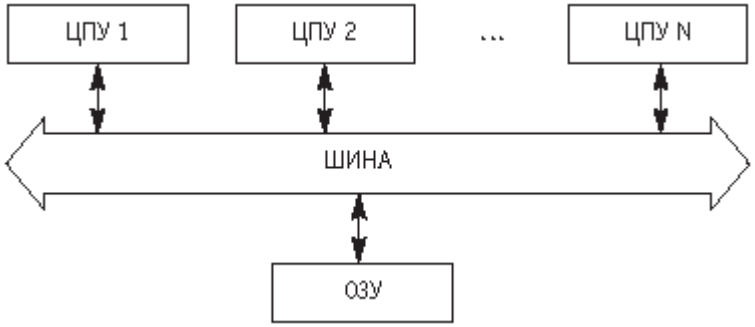
- с выделением управляющего и подчиненных устройств (асимметричное мультипроцессирование)
- с использованием полностью равноправных (симметричное мультипроцессирование).



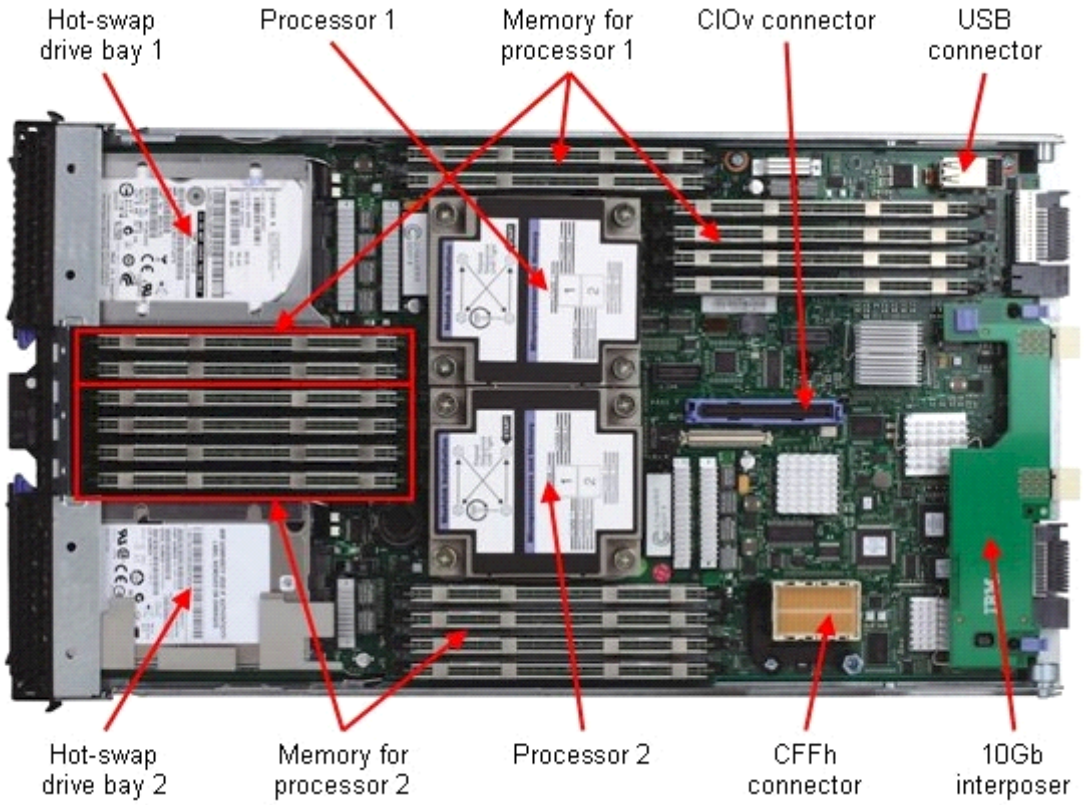
SMP

Shared Memory Processor или Symmetric MultiProcessor

В таких компьютерах несколько процессоров подключены к общей оперативной памяти и имеют к ней равноправный и конкурентный доступ. По мере увеличения числа процессоров производительность оперативной памяти и коммутаторов, связывающих процессоры с памятью, становится критически важной. Обычно в SMP используются 2-8 процессоров; реже число процессоров достигает десятков. Взаимодействие одновременно выполняющихся процессов осуществляется посредством использования общей памяти, к которой имеют равноправный доступ все процессоры



Пример SMP:
IBM HS23 Blade



MPP

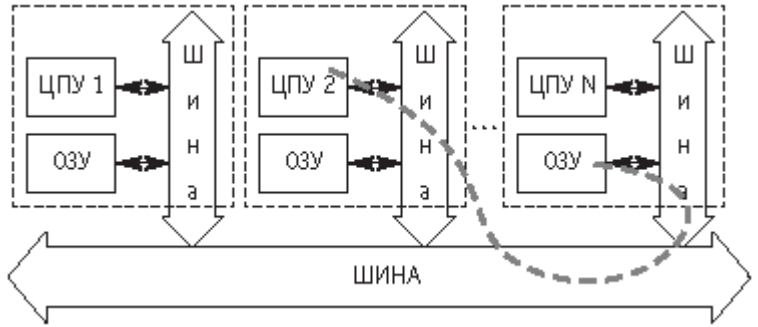
Massively Parallel Processors

В MPP используют несколько однопроцессорных или SMP-систем, объединяемых с помощью некоторого коммуникационного оборудования в единую сеть. При этом может применяться как специализированная высокопроизводительная среда передачи данных, так и обычные сетевые средства - типа Ethernet. В MPP системах оперативная память каждого узла обычно изолирована от других узлов, и для обмена данными требуется специально организованная пересылка данных по сети. Для MPP систем критической становится среда передачи данных; однако в случае мало связанных между собой процессов возможно одновременное использование большого числа процессоров. Число процессоров в MPP системах может измеряться сотнями и тысячами.

NUMA

Non-Uniform Memory Access

Является компромиссом между SMP и MPP системами: оперативная память является общей и разделяемой между всеми процессорами, но при этом память неоднородна по времени доступа. Каждый процессорный узел имеет некоторый объем оперативной памяти, доступ к которой осуществляется максимально быстро; для доступа к памяти другого узла потребуется значительно больше времени.

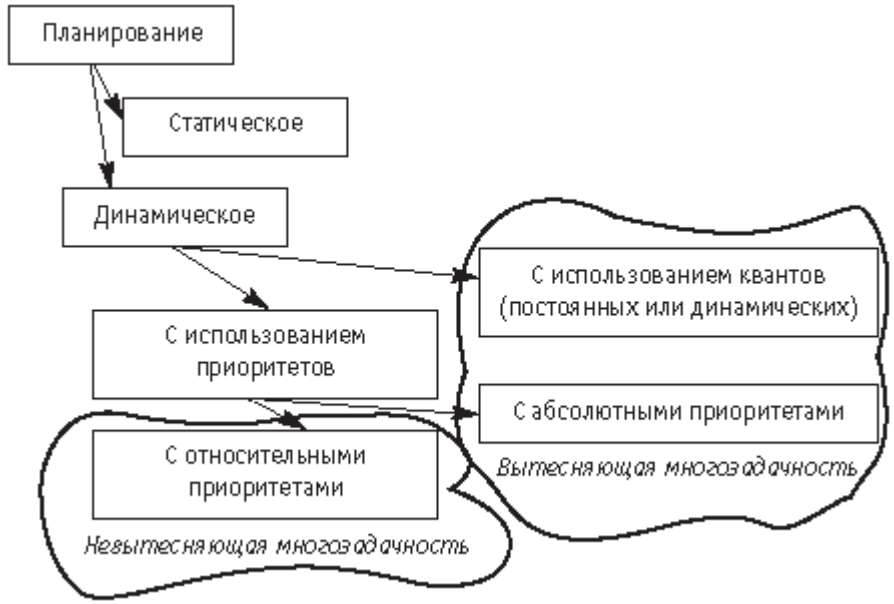


Процессы и потоки

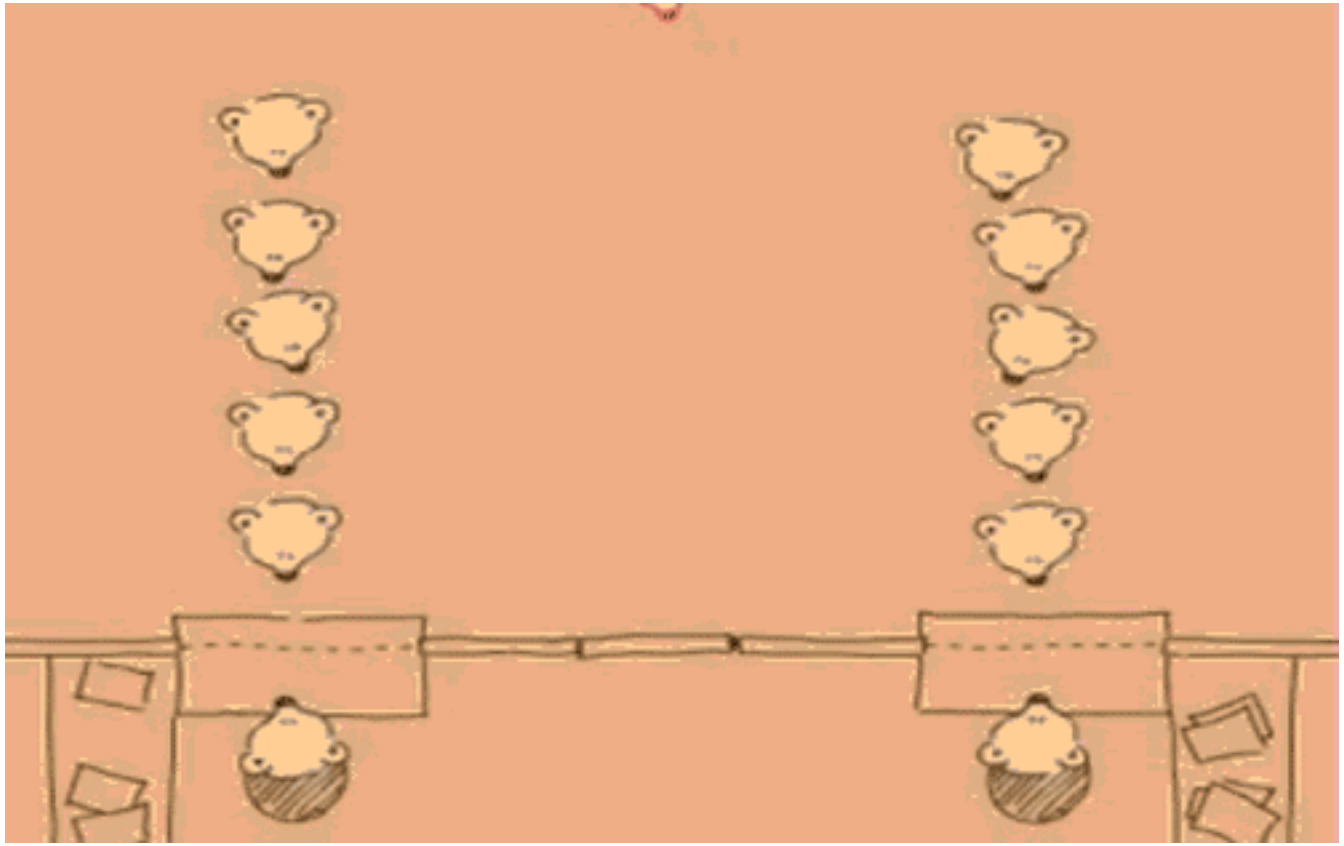
- Программа (program) – это последовательность команд, реализующая алгоритм решения задачи.
- Процесс (process) – это программа (пользовательская или системная) в ходе выполнения.
- В современных операционных системах процесс представляет собой объект – структуру данных, содержащую информацию, необходимую для выполнения программы. Объект "Процесс" создается в момент запуска программы (например, пользователь дважды щелкает мышью на исполняемом файле) и уничтожается при завершении программы.
- Процесс может содержать один или несколько потоков (thread) – объектов, которым операционная система предоставляет процессорное время. Сам по себе процесс не выполняется – выполняются его потоки. Таким образом, машинные команды, записанные в исполняемом файле, выполняются на процессоре в составе потока. Если потоков несколько, они могут выполняться одновременно.

Мультипрограммирование

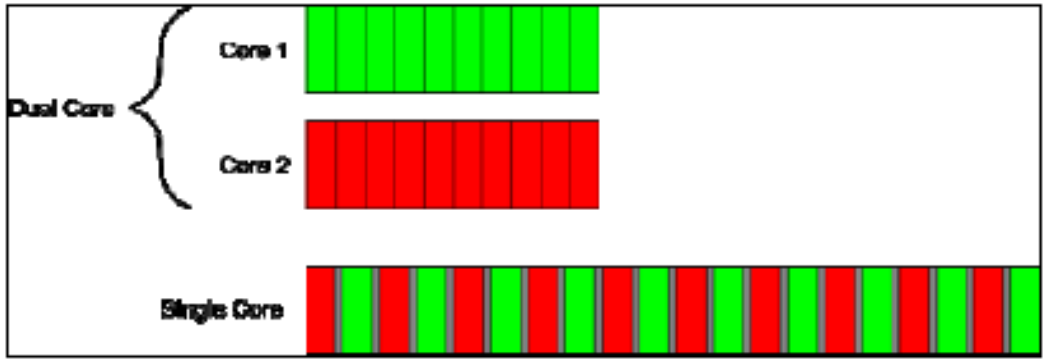
- В мультипрограммировании ключевым местом является способ составления расписания, по которому осуществляется переключение между задачами (планирование), а также механизм, осуществляющий эти переключения.
- По времени планирования можно выделить статическое и динамическое составление расписания.
  - При статическом планировании расписание составляется заранее, до запуска приложений, и операционная система в дальнейшем просто выполняет составленное расписание.
  - В случае динамического планирования порядок запуска задач и передачи управления задачам определяется непосредственно во время исполнения.



Планирование задач



Мультипроцессирование vs Мультипрограммирование

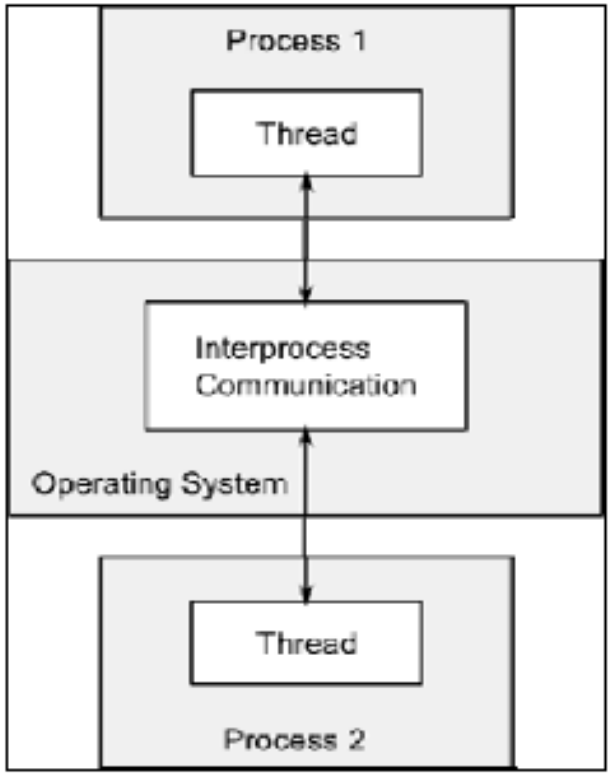


Несколько вычислительных ядер процессора позволяют выполнять несколько задач одновременно.

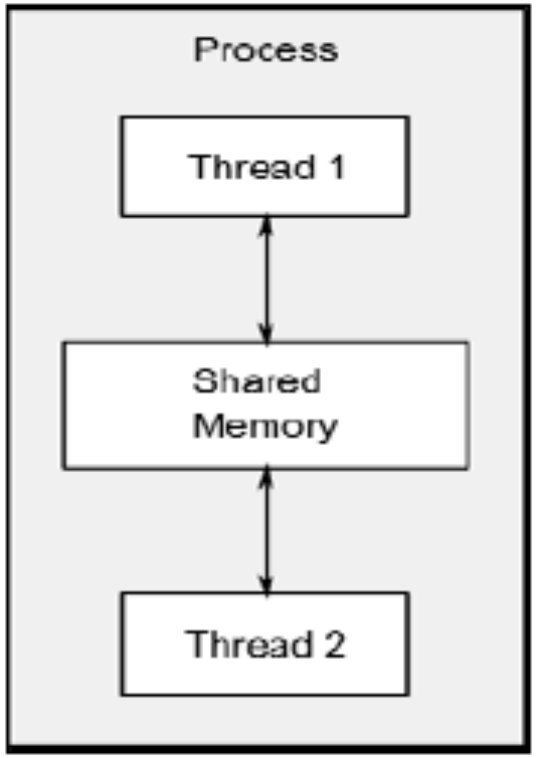
Одно ядро процессора может выполнять несколько задач, только переключаясь между ними.

Два вида мультирпограммирования

Multiple processes

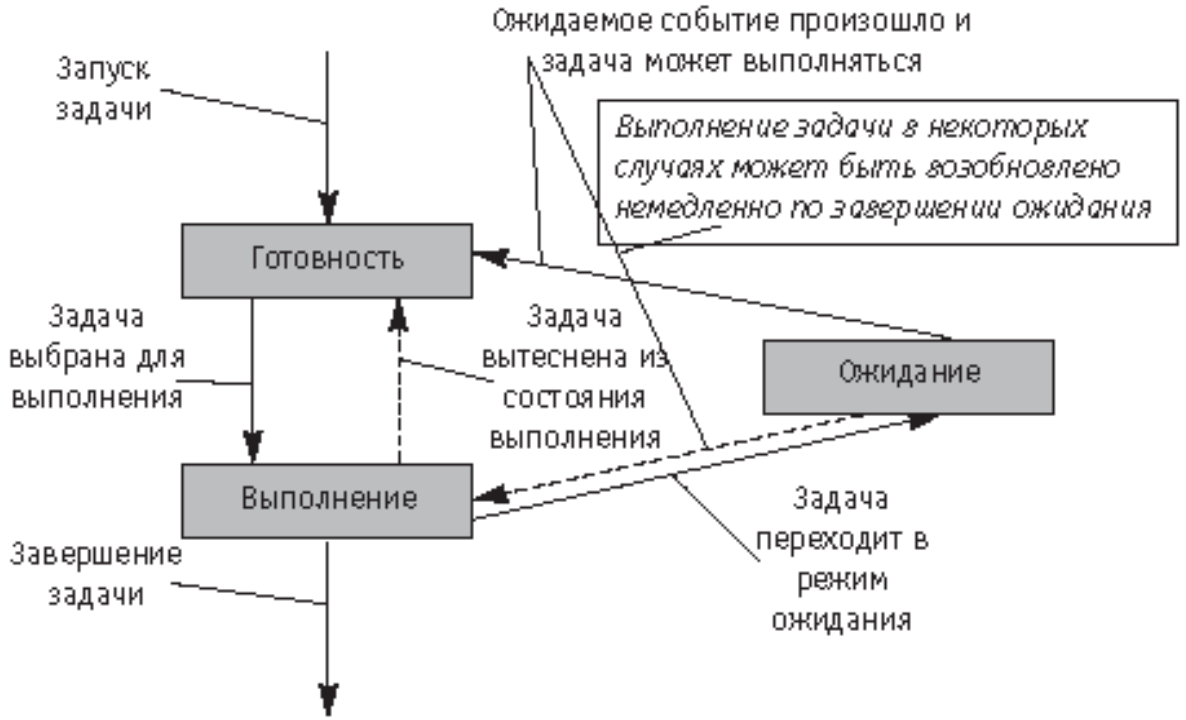


Multiple threads



Два вида управления многозадачностью

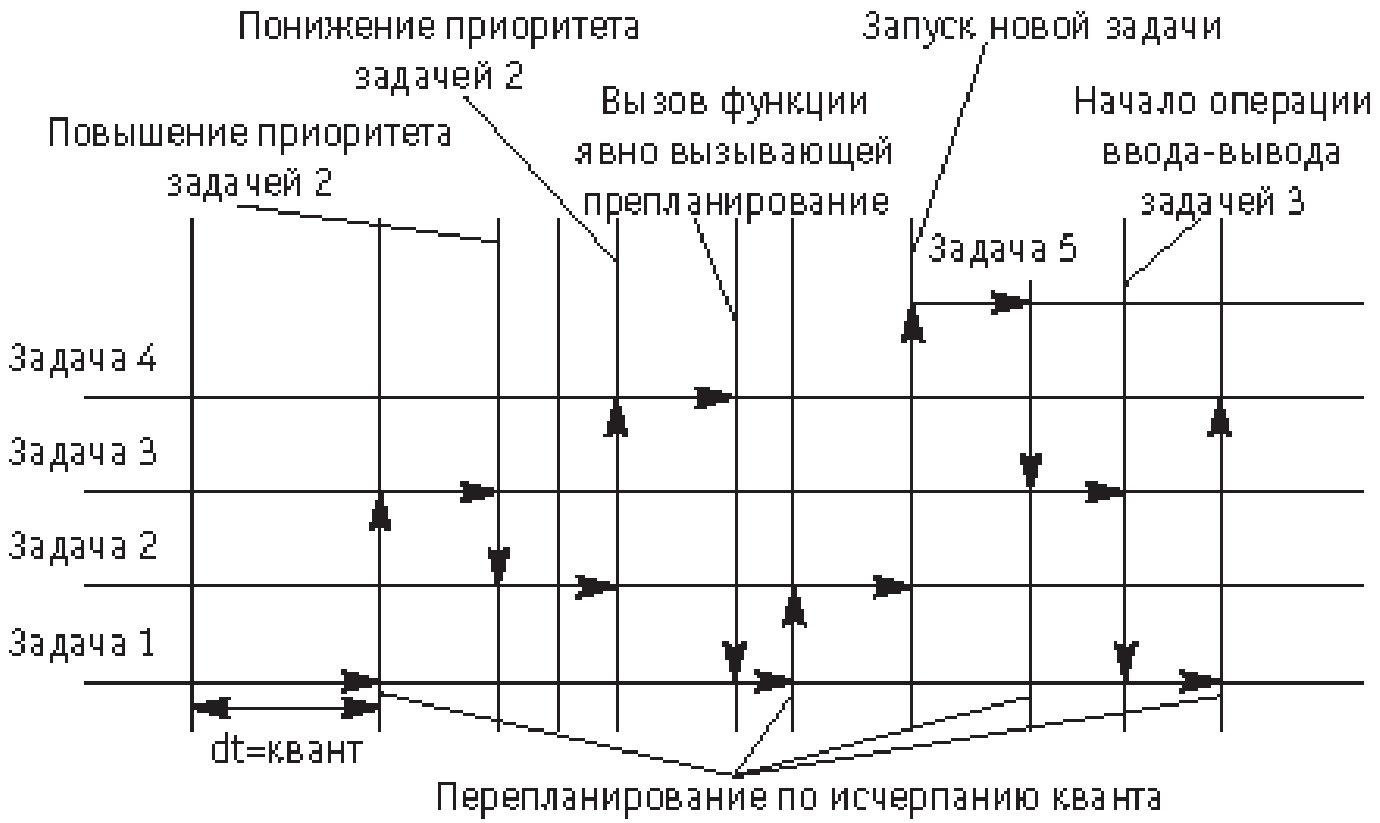
- в случае **невытесняющей** многозадачности решение о переключении принимает выполняемая в данный момент задача
- в случае **вытесняющей** многозадачности такое решение принимается операционной системой (или иным арбитром), независимо от работы активной в данный момент задачи.



Планировщик задач

- Вытесняющая многозадачность предполагает наличие некоторого арбитра, принадлежащего обычно операционной системе, который принимает решение о вытеснении текущей выполняемой задачи какой-либо другой, готовой к выполнению, асинхронно с работой текущей задачи.
- В качестве некоторого обобщения можно выделить понятие "**момент перепланирования**", когда активируется планировщик задач и принимает решение о том, какую именно задачу в следующий момент времени надо начать выполнять. Принципы, по которым назначаются моменты перепланирования, и критерии, по которым осуществляется выбор задачи, определяют способ реализации многозадачности и его сильные и слабые стороны.
- **Перепланирование**:
  - Если смена приоритета вызывает перепланирование - значит, это система с **абсолютными приоритетами** и вытесняющей многозадачностью.
  - Если моменты перепланирования наступают по исчерпанию временных квантов (возможно постоянного размера, а возможно и переменного), то система поддерживает **вытесняющую многозадачность с квантованием**.

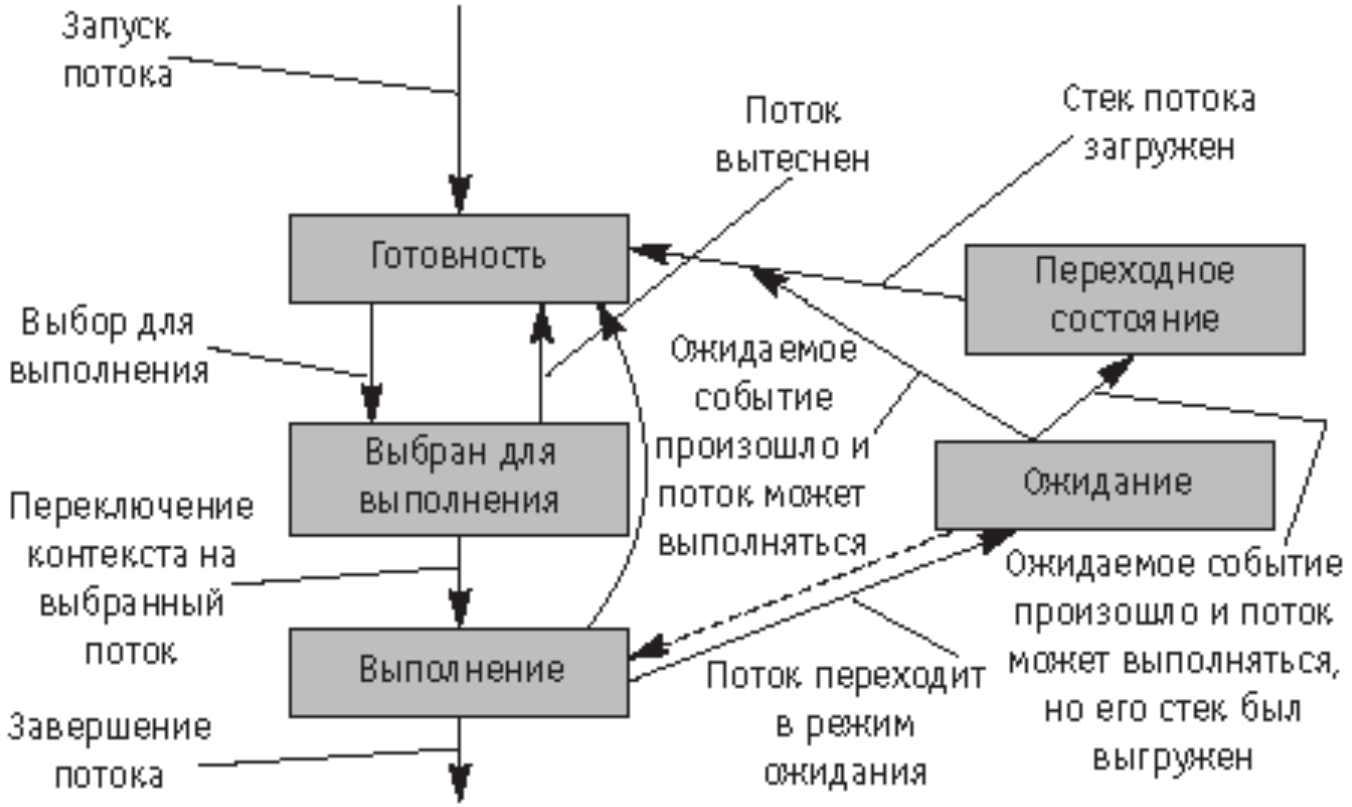
Комбинированное перепланирование задач



Причины изменения приоритета

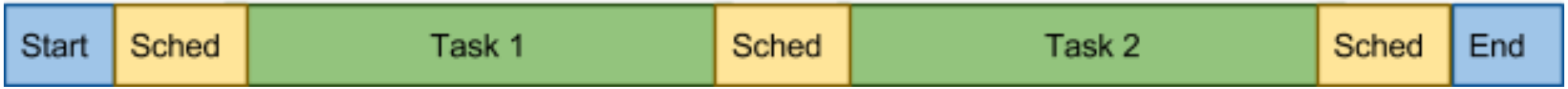
- запуск задачи (для возможно скорейшего начала исполнения);
- досрочное освобождение процессора до исчерпания отведенного кванта (велик шанс, что задача и в этот раз так же быстро отдаст управление);
- частый вызов операций ввода-вывода (при этом задача чаще находится в ожидании завершения операции, нежели занимает процессорное время);
- продолжительное ожидание в очереди (приоритет ожидающей задачи часто постепенно начинают увеличивать);

Жизненный цикл потока в Windows

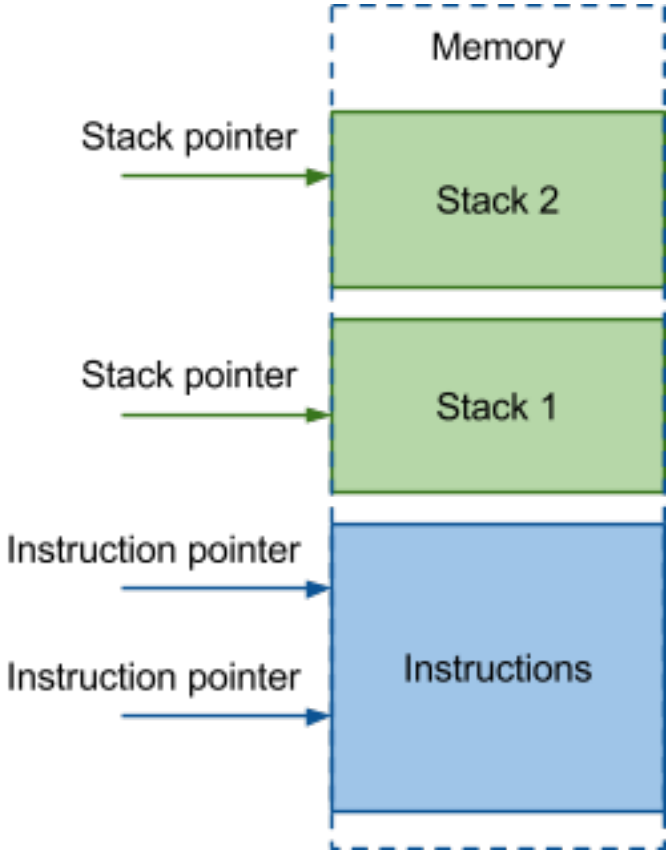


Невытесняющая (кооперативная) многозадачность

Планировщик (scheduler) не может забрать время у вычислительного потока, пока тот сам его не отдаст.



Вытесняющий планировщик

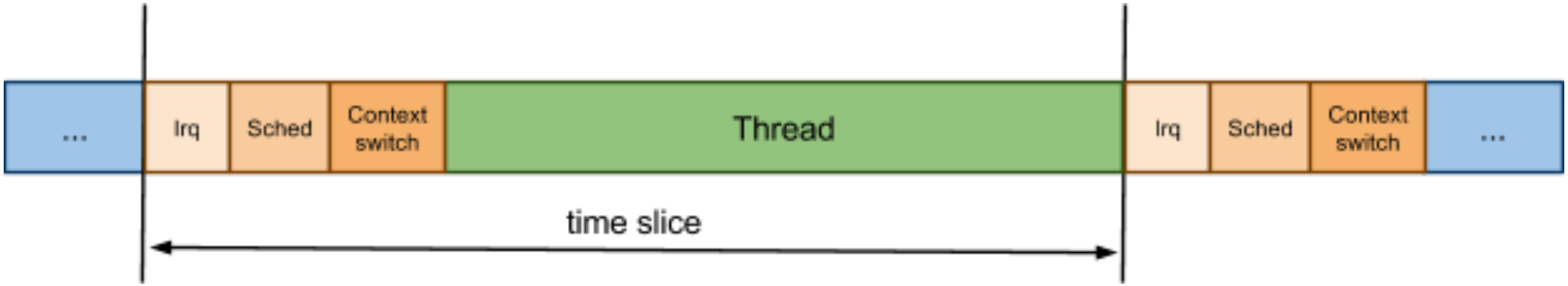


- У нас есть два вычислительных потока, выполняющих одну и ту же программу, и есть планировщик, который в произвольный момент времени перед выполнением любой инструкции может прервать активный поток и активировать другой.
- instruction pointer и stack pointer — хранятся в регистрах процессора. Кроме них для корректной работы необходима и другая информация, содержащаяся в регистрах: флаги состояния, различные регистры общего назначения, в которых содержатся временные переменные, и так далее. Все это называется контекстом процессора.

Переключение контекста

Переключение контекста — замена контекста одного потока другим. Планировщик сохраняет текущий контекст и загружает в регистры процессора другой.

Текущая программа прерывается процессором в результате реакции на внешнее событие — аппаратное прерывание — и передает управление планировщику



За чем применять многозадачность?

1. Разделение программы на независимые части. Один процесс выполняет одну задачу (например, взаимодействие с пользователем), а другой – другую (например, вычисления).
2. Для увеличения производительности.

Увеличение числа параллельных процессов не всегда приводит к ускорению программы.

Hello World!

Example99_Thread

```
1. #include <iostream>
2. #include <thread>
3. void hello() {
4. std::cout<<"Hello Concurrent World\n";
5. }

6. int main(int argc,char * argv[]){
7. std::thread t(hello); // launch new thread
8. t.join(); //wait for finish of threads

9. cin.get();
10.return 0;
11.}
```

std::thread
стандарт C++11

Конструктор

```
template <class Fn, class... Args>  explicit thread (Fn&& fn, Args&&... args);
```

Принимает в качестве аргумента функтор и его параметры.

```
void join();
```

Ожидает когда выполнение потока закончится.

Как передать объект?
Example100_ThreadFunctor

```
1.  #include <thread>
2.  #include <iostream>
3.  class MyClass{
4.  public:
5.      // thread вызовет переопределенный оператор
6.      void operator()(const char* param){
7.      std::cout << param << std::endl;
8.     }
9.  };
10. int main() {
11.     // передается копия объекта!
12.     std::thread my_thread(MyClass(),"Hello world!");
13.     my_thread.join();
14.     return 0;
15. }
```

Передача параметров в поток
Example102_ThreadMoveSemantics

С помощью семантики перемещения можно передавать данные в поток, избегая накладных расходов на копирование большого объема данных в стек потока.

С семантикой перемещения мы 100% уверены что к данным нашего потока не будет доступа из других потоков.

```
class MyClass {
protected:
 std::unique_ptr<Param> ptr;
public:
 MyClass( std::unique_ptr<Param> &&param) {
 ptr = std::move(param);}
 MyClass(MyClass &&other) { // конструктор перемещения
 ptr = std::move(other.ptr);}
 MyClass(const MyClass &) = delete;
 MyClass & operator=(const MyClass &)= delete;
 void operator()() { } // тут код потока
};
```

Полезные функции
std::this_thread

std::thread::hardware_concurrency()

Возвращает количество Thread которые могут выполняться параллельно для данного приложения.

std::this_thread::get_id()

Возвращает идентификатор потока.

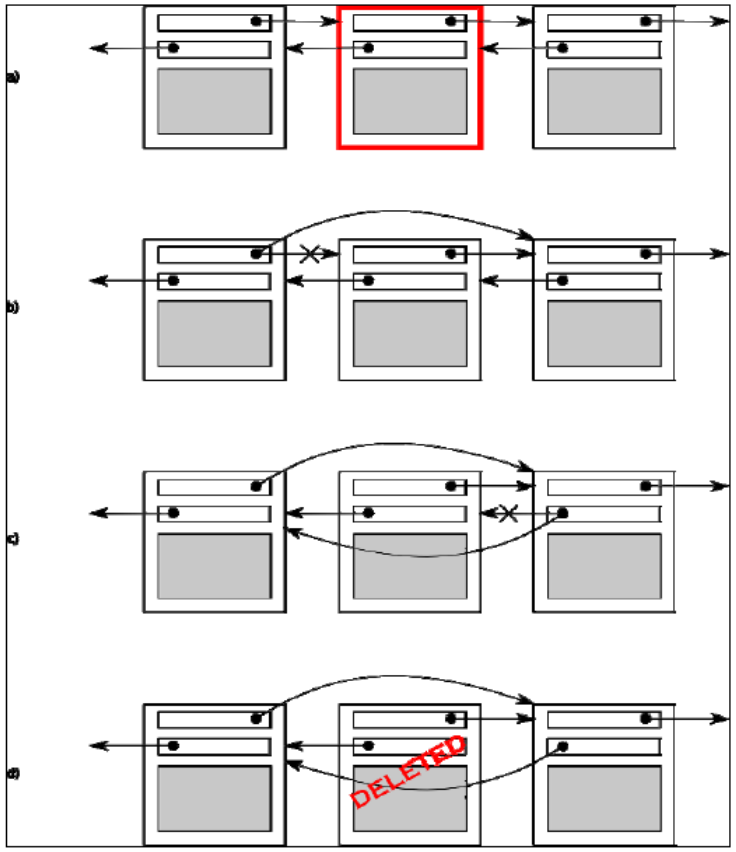
std::this_thread::sleep_for(std::chrono::milliseconds)

Позволяет усыпить поток на время

Как дождаться завершения потока красиво? Example104_ScopedThread

```
class Scoped_Thread {
 std::thread t;
public:
 Scoped_Thread(std::thread&& t_) : t(std::move(t_)) {
  if (!t.joinable()) throw std::logic_error("No thread");
 };
 Scoped_Thread(std::thread& t_) : t(std::move(t_)) {
  if (!t.joinable()) throw std::logic_error("No thread");
 };
 Scoped_Thread(Scoped_Thread & other) : t(std::move(other.t)) { };
 Scoped_Thread(Scoped_Thread && other) : t(std::move(other.t)) {};
 Scoped_Thread& operator=(Scoped_Thread &&other) {
  t = std::move(other.t);        return *this;}
 ~Scoped_Thread() {
  if (t.joinable()) t.join();
  };
};
```

Проблемы работы с динамическими структурами данных в многопоточной среде



При удалении элемента из связанного списка производится несколько операций:

- удаление связи с предыдущим элементом

- удаление связи со следующим элементом

- удаление самого элемента списка

Во время выполнения этих операций к этими элементами обращаться из других потоков нельзя!

Потоковая безопасность

Свойство кода, предполагающее его корректное функционирование при одновременном исполнении несколькими потоками.

Основные методы достижения:

- Реентрабельность;
- Локальное хранилище потока;
- Взаимное исключение;
- Атомарные операции;

Реентерабельность

Свойство функции, предполагающее её корректное исполнение во время повторного вызова (например, рекурсивного).

- Не должна работать со статическими (глобальными данными);
- Не должна возвращать адрес статических (глобальных данных);
- Должна работать только с данными, переданными вызывающей стороной;
- Не должна модифицировать своего кода;
- Не должна вызывать нереентерабельных программ и процедур.

Потоконебезопасный код
Example105_RaceCondition

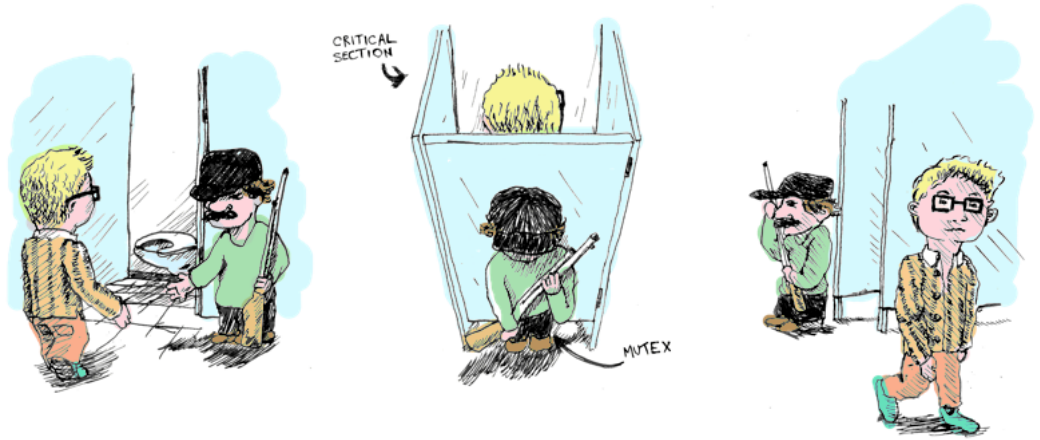
```
void add_function( long * number){
 for( long i=0;i<1000000000L;i++) (*number)++;}

void subst_function( long * number){
 for( long i=0;i<1000000000L;i++) (*number)--;}

int main() {
long number = 0;
{
 Scoped_Thread th1(std::move(std::thread(add_function,&number)));
Scoped_Thread th2(std::move(std::thread(subst_function,&number)));
}
// Результат неопределен!
std::cout << "Result:" << number << std::endl;
return 0;
}
```

Взаимное исключение
`Example108_Mutex1`

Мьютекс — базовый элемент синхронизации и в C++11 представлен в 4 формах в заголовочном файле <mutex>:

mutex
обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock()

timed_mutex
в отличие от обычного мьютекса, имеет еще два метода: try_lock_for() и try_lock_until() позволяющих контролировать время ожидания вхождения в mutex



Рекурсивное вхождение Mutex
Example109_Mutex2

1. **recursive_mutex**
   может войти «сам в себя», т.е. Поддерживает многократный вызов lock из одного потока. Содержит все функции mutex.
2. **recursive_timed_mutex**
   это комбинация timed_mutex и recursive_mutex

Замки

lock_guard
`Example110_LockGuard`

1. Захват в конструкторе
2. Освобождение в деструкторе
3. Используются методы мьютексов
   - Захват: void lock();
   - Освободить: void unlock();

unique_guard
`Example111_UniqueLock`

1. То же, что lock_guard
2. Используются методы мьютексов
   - Попытаться захватить: bool try_lock();
   - Захват с ограничением: void timed_lock(...);
3. + Дополнительные функции получения мьютекса, проверки «захваченности»...

Опасности в многопоточное среде Example101_RaceCondition

```
int array[100];
 …
std::thread s1(sort, array, 100, less);
std::thread s2(sort, array, 100, more);
s1.join();
s2.join();
```

Когда несколько потоков имеют доступ к одним и тем же данным (бесконтрольный), то результат работы алгоритма не определен!

Локальное хранилище потока `Example107_TLC`

- Набор статических/глобальных переменных, локальных по отношению к данному потоку.

- Отличие от реентерабельности в том, что используются не только «локальные переменные потока» и его параметры,, но и специальный менеджер ресурсов.

Вообще говоря, в данном примере TLS не совсем потокобезопасный.

Потоко-безопасный Stack `Example112_ThreadSafeStack`

Классы «обертки» позволяют непротиворечиво использовать мьютекс в RAII-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

lock_guard
когда объект создан, он пытается получить мьютекс (вызывая lock()), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая unlock())

Печать на экран это то же разделяемый ресурс – так что печать то же защищаем мьютексом.

Deadlock
Example113_Deadlock

```
std::lock_guard<std::mutex> lock(a);
```

```
std::lock_guard<std::mutex> lock(b);
```

```
std::lock_guard<std::mutex> lock(b);
```

```
std::lock_guard<std::mutex> lock(a);
```

Возникает когда несколько потоков пытаются получить доступ к нескольким ресурсам в разной последовательности.

Пример опасной ситуации
Example114_PassOut

Передача данных из за границу lock.

Правило:

Ни когда не передавайте во вне указатели или ссылки на защищаемые данные. Это касается сохранение данных в видимой области памяти или передачи данных как параметра в пользовательскую функцию.

Exceptions в многопоточной среде
`Example115_Exception`

1. Исключения между потоками не передаются!
2. Нужно устроить хранилище исключений, для того что бы их потом обработать!

future + async Example117_Future

future – служит для получения результата вычислений из другого потока.

Т.е. Функция выполняемая thread теперь может возвращать значение.

– это шаблон (параметр – тип возвращаемого значения)

– конструируется с помощью std::async

— результат выполнения получается методом get()

— async запускает поток и синхронизирует результат с возвращаемым future

```
T function( ) {
 return T( );
}
int main ( ){
std::future<T> fut = std::async (function);
T x = fut.get( );
}
```

std::future<T>
ограничения

1. Нельзя вызывать сразу несколько get() из разных std::thread. Если нужно синхронизировать сразу несколько потоков – то лучше использовать условные переменные (будет расмотренно далее).
2. Нельзя копировать (только передача по ссылке).

future работает на promise `Example118_Promise`

template <class T> promise – шаблон который сохраняет значение типа T, которое может быть получено с помощью future

get_future – запрос future, связанного со значением внутри promise

set_value – устанавливает значение (и передает его в вызов get_future)

set_exception – передача исключения

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ