**Лекция 15**

Вместо t.join() можно написать **t.detach(),** тогда тред будет работать независимо и выход из main не приведет к terminate.

Тот тред, который в main — leader thread.

Если сделать lock(), но забыть unlock(), программа зависнет.

Решение: сделать RAII над mutex.

**std::lock\_guard<std::mutex>** - этот класс в конструкторе сам залочит mutex, а в деструкторе разлочит.

#include <thread>  
#include <iostream>  
#include <vector>  
  
std::vector<int> v;  
std::mutex m;  
  
void job() {  
 std::lock\_guard<std::mutex> lg(m);  
 // Critical section - этот кусок кода нельзя делать параллельно  
 // из разных тредов  
 for (int i = 0; i < 1000000; ++i) {  
 v.push\_back(i);  
 }  
}  
  
int main() {  
  
 std::thread t(job);  
  
 job(); // параллельное выполнение с тредом  
  
 t.join(); // остановись и подожди, пока тред закончится  
}

**Зачем вообще нужны треды?** Для какого-то параллельного вычисления — так программа работает эффективнее.

Один процессор одномоментно выполняет один тред, тогда если тредов сильно больше, чем ядер процессора, то это будет не очень эффективно.

**Пример:** сделать QuickSort массива на миллиард чисел. Или перемножение больших матриц.

Также треды нужны, когда обрабатываем запросы (при написании сервера). Один запрос обрабатывается в отдельном треде.

Для Linux процессы и треды практически неотличимые сущности. Однако начиная с более новых версий была введена абстракция thread group.

Процессы, имеющие общую тред группу (одинаковый TGID), являются разными тредами одного и того же процесса. У main thread PID = TGID.

Приоритет треда может отличаться от приоритета главного треда.

**clone** - сискол, который позволяет создать дочерний процесс (тред)

(fork с кучей настраиваемых параметров)

*int clone(int (\*fn)(void \*), void \*stack, int flags, void \*arg, ...*

*/\* pid\_t \*parent\_tid, void \*tls, pid\_t \*child\_tid \*/ );*

Первым параметром передаем, какую функцию дочерним процессом хотим исполнять, затем передаем указатель на начало стека (в std::thread стек выделяется mmapом), затем флаги (набор этих флагов определяет, запустим мы тред или новый полноценный процесс).

getpid() - возвращает TGID

gettid() - возвращает TID (в htop это PID)

Если в clone проставить флаг CLONE\_PARENT, то getppid будет одинаковым у всех тредов, равным getppid главного процесса, а не самому этому процессу.

**Разница поведения треда и самостоятельного процесса:**

* Если какой-то тред из тред группы делает execve, то все треды из этой группы завершаются, и новая программа выполняется в лидерском треде.
* Если один из тредов делает форк, то любой тред из этой группы может делать wait на этого ребенка.
* Сигналы могут быть направлены процессу или отдельному треду. Когда делаем kill, он направляется процессу, но с помощью tgkill можно направить сигнал отдельному треду.

**tgkill** — направить сигнал треду

*int tgkill(int tgid, int tid, int sig);*

kill же направлял сигнал всем тредам в тред группе, и какой-то один перехватывал их (не определено).

**Попробуем реализовать std::thread**

Нам нужно: сделать clone с правильными параметрами, выделить руками стек, поставить указатель на функцию, которую планируем выполнять в треде, поставить указатель на стек и пустить исполняться (выход из конструктора).

Join — это waitpid. Pid нужно будет сохранить, когда clone его вернет.

Проблема: тред умеет принимать функции с произвольным набором аргументов. Как передавать их в конструктор? Пока что сделаем так, чтобы тред принимал функции без аргументов.

Выделение стека происходит mmapом, выделяется память сначала без прав, а потом отступается одна страница и на оставшуюся часть выдаются права read и write. Это сделано для безопасности: чтобы начальная страница была защищена, и обращение к ней (за границу стека) приводило бы к сегфолту.

Также нужно вызывать mmap с дополнительным флагом MAP\_STACK: флаг, который указывает, что мы аллоцируем стек.

**Версия с неработающим join:**

#include <thread>  
#include <iostream>  
#include <vector>  
#include <sys/mman.h>  
#include <sys/wait.h>  
#include <linux/futex.h>  
  
class Thread {  
private:  
 using Callable = void(\*)();  
 Callable func;  
 static int *threadRoutine*(void\* arg) {  
 Callable routine = reinterpret\_cast<Callable>(arg);  
 routine();  
 return 0;  
 }  
 void\* stack;  
 int pid = -1;  
  
public:  
 Thread(Callable func) : func(func) {  
 stack = mmap(NULL, 8 \* 1024 \* 1024, PROT\_READ | PROT\_WRITE,  
 MAP\_PRIVATE | MAP\_ANONYMOUS | MAP\_STACK, -1, 0);  
 void\* stackTop = static\_cast<char\*>(stack) + 8 \* 1024 \* 1024;  
  
 pid = clone(*threadRoutine*, stackTop,  
 CLONE\_THREAD | CLONE\_VM | CLONE\_FS | CLONE\_FILES |  
 CLONE\_SIGHAND, (void\*)func);  
 }  
  
 void join() {  
 int status;  
 waitpid(pid, &status, 0);  
 pid = -1;  
 }  
  
 ~Thread() {  
 if (pid > 0) {  
 std::terminate();  
 }  
 munmap(stack, 8\*1024\*1024);  
 }  
};  
  
void f() {  
 for (int i = 0; i < 1000; ++i) {  
 std::cout << i << ' ';  
 }  
 std::cout << std::endl;  
}  
  
int main() {  
 Thread t(&f);  
  
 f();  
  
 t.join();  
}

Здесь, в отличие от реализации на лекции, указатель stack, передаваемый в clone, указывает на конец выделенного стека (не на его начало). Тогда segfaultа нет.

Но работает слишком UBшно, то завершается, то не может (проблема с waitpid).

Можно попробовать через futex.

**Как передать аргументы в функцию через конструктор?**

Через type erasure. Есть базовый класс и шаблонный наследник, в котором хранится объект Callable. Когда создаем тред: thread(Callable&& f, Args&&… args), первым делом вызываем std::bind(f, args…). Таким образом создаем функциональный объект, у которого уже подставлены аргументы, и его можно просто вызвать без аргументов.

Также мы должны создать shared\_ptr на этого наследника (это делается через функцию make\_routine), и сохраняем его у себя как shared ptr на базовый класс.

Поэтому когда просят вызвать тред, по shared\_ptr вызываем метод Run из базового класса, а так как он виртуальный, то вызовется то что нужно.

Ну вот эта версия с горем пополам работает получше:

#include <thread>  
#include <iostream>  
#include <vector>  
#include <sys/mman.h>  
#include <sys/wait.h>  
#include <linux/futex.h>

#define SYS\_futex 202  
  
class Thread {  
private:  
 using Callable = void(\*)();  
 Callable func;  
  
 int threadState = 0; // 0 - поток выполняется, 1 - завершен  
  
 static int *futexWait*(int\* addr, int expected) {  
 return syscall(SYS\_futex, addr, FUTEX\_WAIT, expected, nullptr, nullptr, 0);  
 }  
  
 static int *futexWake*(int\* addr, int count) {  
 return syscall(SYS\_futex, addr, FUTEX\_WAKE, count);  
 }  
  
 static int *threadRoutine*(void\* arg) {  
 Thread\* thread = reinterpret\_cast<Thread\*>(arg);  
 thread->func();  
 \_\_sync\_synchronize(); // Барьер памяти для гарантии обновления threadState  
 thread->threadState = 1;  
 *futexWake*(&thread->threadState, 1);  
 return 0;  
 }  
  
  
  
 void\* stack;  
 int pid = -1;  
  
public:  
 Thread(Callable func) : func(func) {  
 stack = mmap(NULL, 8 \* 1024 \* 1024, PROT\_READ | PROT\_WRITE,  
 MAP\_PRIVATE | MAP\_ANONYMOUS | MAP\_STACK, -1, 0);  
 void\* stackTop = static\_cast<char\*>(stack) + 8 \* 1024 \* 1024;  
  
 pid = clone(*threadRoutine*, stackTop,  
 CLONE\_THREAD | CLONE\_VM | CLONE\_FS | CLONE\_FILES |  
 CLONE\_SIGHAND, this);  
 }  
  
 void join() {  
 while (true) {  
 if (threadState == 1) {  
 break;  
 }  
 *futexWait*(&threadState, 0);  
 }  
 pid = -1;  
 }  
  
 ~Thread() {  
 if (pid > 0) {  
 join();  
 }  
 munmap(stack, 8 \* 1024 \* 1024);  
 }  
};  
  
void f() {  
 for (int i = 0; i < 1000; ++i) {  
 std::cout << i << ' ';  
 }  
 std::cout << std::endl;  
}  
  
int main() {  
 Thread t(&f);  
 f();  
 t.join();  
}

Сам сискол как то плохо отрабатывает. Ну в целом сойдет...