**Лекция 14**

**Shared memory** — третий вид межпроцессного взаимодействия

**shmat** — сискол, который позволяет воспользоваться разделяемой памятью

**Разделяемая память** — общий участок памяти у нескольких процессов. То есть можно попросить ОС какие-то участки памяти сделать общедоступными.

*void \*shmat(int shmid, const void \*shmaddr, int shmflg);*

*int shmdt(const void \*shmaddr);*

Адрес присоединения определяется shmaddr с помощью одного из следующих критериев:

• Если значение shmaddr равно NULL, система выбирает подходящий (неиспользуемый) адрес, выровненный по странице, для присоединения сегмента.

• Если значение shmaddr не равно NULL, а в shmflg указано значение SHM\_RND, то подключение происходит по адресу, равному shmaddr, округленному в меньшую сторону до ближайшего кратного SHMLBA.

• В противном случае shmaddr должен быть адресом, выровненным по странице, по которому происходит прикрепление.

**shmget** — сискол, позволяющий выделить разделяемую память

*int shmget(key\_t key, size\_t size, int shmflg);*

key — ключ, который однозначно идентифицирует кусок разделяемой памяти. Он должен быть уникальным на всю ОС (нужно, чтобы он не пересекался с каким-то другим ключом).

**Пример: пишем в SM**

#include <sys/stat.h>  
#include <unistd.h>  
#include <string>  
#include <string.h>  
#include <fcntl.h>  
#include <sys/shm.h>  
  
#define SHM\_SIZE 1000000  
  
int main() {  
 int shm\_id;  
 char\* shm\_ptr;  
  
 // create the shared memory segment  
 shm\_id = shmget(12222, SHM\_SIZE, IPC\_CREAT | 0666);  
 if (shm\_id == -1) {  
 perror("shmget");  
 exit(1);  
 }  
  
 // sttach the shared memory segment to the process address space  
 shm\_ptr = (char\*) shmat(shm\_id, NULL, 0);  
 if (shm\_ptr == (char\*)-1) {  
 perror("shmat");  
 exit(1);  
 }  
  
 // write tp shared memory  
 const char\* message = "Hello from writer!";  
 sprintf(shm\_ptr, "%s", message);  
 printf("Writer: wrote to shared memory: %s\n", message);  
  
 getchar();  
  
 // detach from the shared memory  
 shmdt(shm\_ptr);  
}

Если посмотрим на maps этого процесса, увидим новую строчку:

78dba26c7000-78dba27bc000 rw-s 00000000 00:01 32803 /SYSV00002fbe (deleted)

rw-s — последний флаг здесь означает, что этот кусок памяти shared (p — private)

При завершении процесса память не исчезла, а осталась, зарезервированная операционкой, и она дожидается, пока кто-то еще ей воспользуется.

**shmctl** — сискол, который позволяет выполнять разные операции над shared memory, в том числе, например, удаление разделяемого сегмента памяти

*int shmctl(int shmid, int cmd, struct shmid\_ds \*buf);*

**Пример: читаем из SM**

#include <sys/stat.h>  
#include <unistd.h>  
#include <string>  
#include <string.h>  
#include <fcntl.h>  
#include <sys/shm.h>  
  
#define SHM\_SIZE 1000000  
  
int main() {  
 int shm\_id;  
 char\* shm\_ptr;  
  
 // create the shared memory segment  
 shm\_id = shmget(12222, SHM\_SIZE, 0666);  
 if (shm\_id == -1) {  
 perror("shmget");  
 exit(1);  
 }  
  
 // sttach the shared memory segment to the process address space  
 shm\_ptr = (char\*) shmat(shm\_id, NULL, 0);  
 if (shm\_ptr == (char\*)-1) {  
 perror("shmat");  
 exit(1);  
 }  
  
 // Read from shared memory  
 printf("Reader: read from shared memory: %s\n", shm\_ptr);  
  
 // detach from the shared memory  
 shmdt(shm\_ptr);  
  
 // Optionally, remove the shared memory segment  
 shmctl(shm\_id, IPC\_RMID, NULL);  
}

**Как получить уникальный ключ?**

**ftok** — функция, которая конвертирует строку с названием имени файла в уникальный ключ (файл должен существовать, например это может быть путь к исполняемому файлу).

*key\_t ftok(const char \*pathname, int proj\_id);*

proj\_id нужен для той ситуации, когда для одного и того же файла хотим сгенерировать разные ключи.

**Использование:**

key\_t key = ftok("shmfile", 65);  
if (key == -1) {  
 perror("ftok");  
 exit(1);  
}

Нет гарантий, что в результате получим уникальный ключ.

**Как посмотреть, какие сейчас в ОС существуют сегменты разделяемой памяти?**

**ipcs** — команда, которая показывает эту информацию

**ipcs -p** — дополнительно показывает, какой процесс создал этот сегмент, и какой процесс последний раз к нему обращался

Это также можно увидеть в файле: **cat /proc/sysvipc/shm**

**ps -ef | grep pid** — узнать по pid, что это за процесс.

**System V** — набор функций ОС, который включат в себя управление разделяемой памятью, сигналы, другие механизмы IPC

**POSIX** — современный стандарт

**Пример с функциями стандарта POSIX:**

#include <sys/stat.h>  
#include <unistd.h>  
#include <sys/mman.h>  
#include <fcntl.h>  
#include <string>  
  
#define SHM\_NAME "/my\_shm"  
#define SHM\_SIZE 256  
  
int main() {  
 int shm\_fd;  
 char\* ptr;  
  
 // create shared memory object  
 shm\_fd = shm\_open(SHM\_NAME, O\_CREAT | O\_RDWR, 0666);  
 if (shm\_fd == -1) {  
 perror("shm open");  
 exit(1);  
 }  
  
 // configure the size of the shared memory object  
 ftruncate(shm\_fd, SHM\_SIZE);  
  
 // map the shared memory object in the address space  
 ptr = (char\*)mmap(0, SHM\_SIZE, PROT\_WRITE, MAP\_SHARED, shm\_fd, 0);  
  
 // write to shared memory  
 const char\* message = "Hello from writer!";  
 sprintf(ptr, "%s", message);  
 printf("Writer: wrote to shared memory: %s\n", message);  
  
 getchar();  
  
 // clean up  
 munmap(ptr, SHM\_SIZE);  
 close(shm\_fd);  
}

После завершения программы файл не удалится (он не на диске хранится, а в оперативной памяти), какой-нибудь другой процесс сможет открыть его и прочитать, что в нем написано. Этот файл будет храниться в директории /dev/shm

Еще есть два вида IPC: очереди сообщений и семафоры.

**msgget** - get a System V message queue identifier

*int msgget(key\_t key, int msgflg);*

key — такой же уникальный ключ, как и для shared memory был. То есть по нему и разделяемую память можно получить, и очередь сообщений создать, и семафор тоже.

**Проблемы разделяемой памяти:**

Проблема синхронизации. Если один процесс что-то пишет в память, а другой читает и это происходит несинхронизированно, получим UB.

**Многопоточка**

Хотим создать новый процесс, но который не копирует всю память, как при fork, а переиспользует ее. Fork с сохранением адресного пространства. Это **threads**.

**Чем отличается процесс от треда?** С точки зрения ОС это почти одно и то же, отличие в том, что у тредов общие ресурсы, одинаковое виртуальное пространство.

**Как создать тред?**

Треды пришли в плюсы с C++11.

**std::thread** — класс, которому в конструктор передается функциональный объект: указатель на функцию, либо класс с оператором (), либо лямбда.

Тред начинает работать, как только завершился конструктор, т.е. вызов конструктора треда и есть его запуск.

Конкурентно использовать cout — UB.

#include <thread>  
#include <iostream>  
  
void job() {  
 for (int i = 0; i < 1000000; ++i) {  
 std::cout << i << ' ';  
 }  
}  
  
int main() {  
  
 std::thread t(job);  
  
 job(); // параллельное выполнение с тредом  
  
}

Этот код выдаст *terminate called without an active exception*.

Это произошло, потому что для треда должно было выполняться требование: все треды должны быть завершены (заджоинены или задетачены) перед завершением main.

Если мы вышли из main, а тред продолжает выполнение, то в деструкторе треда вызывается abort.

**Как в тред передать параметры функции?** Через запятую в конструкторе.

#include <thread>  
#include <iostream>  
  
void job(char c) {  
 for (int i = 0; i < 1000000; ++i) {  
 std::cout << c;  
 }  
}  
  
int main() {  
  
 std::thread t(job, 'a');  
  
 job('b'); // параллельное выполнение с тредом  
  
 t.join(); // остановись и подожди, пока тред закончится  
}

**Что, если тред выполняет операцию, которую нельзя делать параллельно?**

#include <thread>  
#include <iostream>  
#include <vector>  
  
std::vector<int> v;  
  
void job() {  
 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  
}

Будет Aborted с разными вердиктами. При реаллокации могли прервать малок, другой процесс снова стал делать пушбек ну и проблема.

Если векторы будут разные для тредов, то все будет работать нормально, потому что малок устойчив к многопоточным вызовам на разные области памяти.

**Как избежать проблему конкурентного обращения к одной и той же памяти?**

**std::mutex** — объект, который позволяет получить эксклюзивный доступ к какому-то участку кода.

Когда вызываем lock(), блокируемся до тех пор, пока не сделаем unlock(), никто другой тогда не сможет сделать lock() в это время — он зависнет до тех пор, пока mutex не освободится.

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