


Раздел «Язык Си» . OOP-Instrumental_3sem1 :

- [Дочерние процессы.](#)
 -  [Задачи](#)
 - [Задача 1.](#)
 - [Задача 2.](#)
 - [Задача 3.](#)

Объекты могут состоять их других объектов, как уже реализованных ранее (например *string*, *sstream*, *istream*, *ofstream* и др.), так и из тех, которые были описаны нами.

Простые и сложные объекты могут порождаться используя стековую память функции, в которой они локальные или используя динамическую память ("кучу"). В первом случае при завершении функции объект уничтожается (с использованием деструктора). Во втором необходимо освобождать память "вручную" с использованием оператора *delete*.

Дочерние процессы.

Каждый процесс может породить свой дочерний процесс. При этом дочерний процесс наследует от родительского все значения переменных, открытые дескрипторы, терминал. Но этот дочерний процесс – совершенно самостоятельный: у него свой *pid*, своя память и т.д.

Пример использования порожденных процессов.

```
#include <unistd.h>
#include <iostream>
#include <cstdlib>
#include <sstream>
#include <ctime> // для clock()
#include <fstream>

using namespace std;
int main(){
    // строковый поток (можно пользоваться операторами ввода/вывода
    stringstream sp;
    // строка
    string st;
    int pid; // pid текущего процесса
    int ppid; // pid родительского процесса
    // получить время начала работы родительского процесса
    unsigned int start_time = clock();
    // в цикле получаем запрос для исполнения: w - запись в файл
    // r - читать из файла, d - удалить
    while(1){
        // получаем строку
        cin>>st;
        // если ввели *, то выходим из программы
        if(st == "*") exit(1);

        // порождаем детский процесс
        // родительский процесс при этом получает pid детского процесса,
        // а в детском pid будет равен 0
        // На каждой итерации цикла порождается
        // НОВЫЙ детский процесс
        pid = fork();
        // если это детский процесс, выходим из цикла,
        // а родитель остается в цикле получать команды
        // каждую команду будет обрабатывать свой детский процесс
        if(pid == 0) break;
    }

    // получаем порождения детского процесса
```

Поиск

Раздел «Язык Си»

[Главная](#)
[Зачем учить C?](#)
[Определения](#)

Инструменты:

[Поиск](#)
[Изменения](#)
[Index](#)
[Статистика](#)

Разделы

[Информация](#)
[Алгоритмы](#)
[Язык Си](#)
[Язык Ruby](#)
[Язык](#)
[Ассемблера](#)
[Ei Judge](#)
[Парадигмы](#)
[Образование](#)
[Сети](#)
[Objective C](#)

[Login>>](#)

```

    unsigned int run_time = clock();
    // генерим имя файла для записи действий
    sp<<"log_"<<getpid();
    // открываем файл на ввод
    ofstream fo;
    // sp.str() возвращает строку, а c_str() от строки возвращает C-строку (массив символов)
    fo.open((sp.str()).c_str());
    // записываем информацию в файл
    fo<<"fork pid: "<<getpid()<<" at time: "<< run_time -start_time<<endl;

    if(st == "w"){
        fo<<" пишем\n";
    }
    if(st == "s"){
        fo<<" ищем\n";
    }
    if(st == "d"){
        fo<<" удаляем\n";
    }
    fo.close();
}

```

Дан пример реализации объектов типа **Actor**. Объекты **Actor** нужны нам для моделирования взаимодействия независимых процессов в борьбе за разделяемый ресурс. **Actor** в данной реализации только сообщает о своем наличии путем увеличения или уменьшения общего количества объектов в системе используя очередь сообщений.

Самый первый объект создает очередь сообщений и сразу записывает в нее число **1**. Все последующие объекты уже могут читать сообщения из очереди. Таким образом каждый объект при создании проверяет есть ли очередь, читает сообщение, увеличивает число на 1 и записывает сообщение в очередь снова. При удалении объекта, он также читает сообщение. Если количество объектов больше 1, то число уменьшается на 1 и сообщение отправляется в очередь.

Для реализации составим программу, которая порождает дочерние процессы, и меняет содержимое каждого дочернего процесса на код программы, реализующей запуск одного объекта

Заголовочный файл actor.h

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <stdlib.h>
#include <sys/msg.h>
#include <time.h>
#include <errno.h>
#include <string.h>

#include <iostream>
#include <cstdlib>
#include <fstream>

#define PERM 0666
using namespace std;

// Структура для записи и чтения в/из очереди сообщений
struct Mess{
    long type; // тип сообщения
    int number; // количество процессов
    int pid; // pid каждого процесса
};

// Процесс, он же действующее лицо для других задач
class Actor{
    Mess mSend, mRead; // сообщения для послыки и чтения
    key_t key; // ключ для создания очереди сообщений
    int mesid; // идентификатор очереди сообщений
    int lng;
    int n;
public:

```

```
// Конструктор
// Создаем или получаем mesid
Actor();
// Удаляем очередь
~Actor();
// что-то делаем
void act(int);
};
```

Реализация класса **Actor**:

```
#include "actor.h"

Actor::Actor(){
// печать для иллюстрации
cout<<"Конструктор Actor: ";
// получение ключа
if ((key = ftok("act2", 'A'))<0){
    printf("Can't get key\n");
    exit(1);
}
// для сохранения кода ошибки
int err = 0;
// пытаемся создать очередь сообщений
if((mesid = msgget(key,PERM|IPC_CREAT|IPC_EXCL))<0){
// сохраним код ошибки
    err = errno;
    if(errno == EEXIST){
        cout <<"Очередь уже создана\n";
// пытаемся получить mesid
        if ((mesid = msgget(key,0))< 0){
            printf("Can't create message's queue\n");
            exit(1);
        }
    }else{
        printf("Can't create message's queue\n");
        exit(1);
    }
}

mSend.pid = getpid();
// Читаем-пишем сообщения типа 1
mSend.type = mRead.type = 1L;

if (err == 0){
// Очередь создал именно этот процесс
cout<<"Посылаем\n";
mSend.number = 1;
if (msgsnd(mesid,(void*)&mSend,sizeof(Mess),0) < 0){
    cout<<"Can't write message\n";
    exit(1);
}
    cout<<"1\n";
}
else{
// Очередь создана другим процессом
    mRead.type = 1L;
    cout<<"Читаем:";
// Сначала читаем сообщение и увеличиваем число
    n=msgrcv(mesid,&mRead, sizeof(Mess), mRead.type,0);

    cout<<mRead.number<<endl;
    mSend.number = mRead.number + 1;
    cout<<"New number: "<<mSend.number<<endl;

// отсылаем сообщение обратно с новым числом
    if (msgsnd(mesid,(void*)&mSend,sizeof(Mess),0) < 0){
```

```

        cout<<"Can't write message\n";
        exit(1);
    }

    cout<<" Я тоже пришел: "<< mSend.number<<endl;
}
};

// Деструктор с проверкой возможности удаления очереди
Actor::~Actor(){

    cout<<"Desctuctor: читаем сообщение \n";

    // Проверяем сколько прцессов еще не удалено
    n=msgrcv(mesid,&mRead, sizeof(Mess), mRead.type,0);
    cout<<mRead.number<<endl;
    // Если число больше 1, то уменьшаем и посылаем новое сообщений
    if(mRead.number > 1){
        mSend.number = mRead.number - 1;
        cout<<"Посылаем новый номер: "<<mSend.number<<endl;
        if (msgsnd(mesid,(void*)&mSend,sizeof(Mess),0) < 0){
            cout<<"Can't write message\n";
            exit(1);
        }
        cout<<"Ушел: "<< mRead.number<<endl;
    }else{
        // Остался последний. Удаляем очередь
        if(msgctl(mesid,IPC_RMID,0)<0){
            printf("Can't delete queue\n");
            exit(1);
        }
        cout<<"Последний удалил очередь: "<< mRead.number<<endl;
    }
};

// Действие
void Actor::act(int tm){
    cout<<"Работаем, работаем: "<<tm<<endl;
    sleep(tm);
};

```

Для работы одного Actor пишем программу (после компиляции она получит имя **act2**):

```

#include "actor.h"

int main(int argc, char **argv){
    int t;
    // ofstream ff("l1.dat");
    // параметр - время работы (ожидания)
    t = atoi(argv[1]);
    cout<< t<< endl;
    // ff<<"Actor "<<getpid()<<endl;
    // ff.close();
    // создаем объект
    Actor a;
    // действие
    a.act(t);
    // деструктор отработает когда main завершает работу
    return 0;
}

```

Программа (**t_fork**) для запуска нескольких процессов как дочерних. Содержание дочернего процесса заменяется кодом **act2**

```

#include "actor.h"

int main(){
    int pid;
    // запустить дочерний процесс
    pid = fork();
    // если процесс детский:

```

```
if(pid == 0){
    cout<<"Дите: "<<getpid()<<endl;
    // время "работы"
    int timeP = rand()%100;
    cout<<"timeR: "<<timeP<<endl;
    // формируем строку параметра для запуска act2
    char buf[20];
    sprintf(buf,"%d",timeP);
    // Замещение содержимого этого процесса кодом act2
    // со строкой параметров buf
    execl("act2",buf,NULL);

}

return 0;
}
```

Компилируем и линкуем **act20**, **t_fork**.

Задачи

Задача 1.

Запустить в цикле **n** процессов. **n** – параметр запуска **t_fork**.

Задача 2.

Изменить программный код так, чтобы каждый процесс записывал информацию не на экран, а в файл с именем **log**

Задача 3.

В маленькой мастерской три рабочих осуществляют окончательную сборку некоторого устройства из полуфабриката, установленного в тисках, закрепляя на нем две одинаковые гайки и один винт. Два рабочих умеют обращаться только с гаечным ключом, а один – только с отверткой. Действия рабочих схематически описываются следующим образом: взять элемент крепежа и, при наличии возможности, установить его на устройство; если все три элемента крепежа установлены, то вынуть из тисков готовое устройство и закрепить в них очередной полуфабрикат. Размеры устройства позволяют в данный момент времени работать только одному рабочему. Каждый рабочий, сделав одну свою операцию, уходит от устройства. Рабочий, который закончил обрабатывать деталь (готова), ставит на ее место новую и выполняет одну свою операцию. Самую первую деталь ставит рабочий с отверткой.

Написать две программы: для рабочего с отверткой и для рабочего с гайкой, используя класс **Actor** для моделирования процесса в течении 1 минуты. Время, затраченное на операцию каждым рабочим соответствующая программа получает как параметр. Всю информацию о своей работе каждый процесс выводит в файл с названием **log**.

-- TatyanaOvsyannikova2011 – 20 Oct 2016

(с) Материалы раздела "Язык Си" публикуются под лицензией [GNU Free Documentation License](#).