Язык программирования Си

Fork - Exec

(практическое руководство по лабораторной №28)

Владимир Валерьевич Соловьёв
Huawei, НГУ
vladimir.conwor@gmail.com
t.me/conwor
vk.com/conwor

Задание №28 - посчитать **N**-ое число Фибоначчи

Задание №28 - посчитать **N**-ое число Фибоначчи

Но не всё так просто - считать нужно рекурсивным алгоритмом, в котором рекурсия осуществляется не на функциях, а на программах

Задание №28 - посчитать **N**-ое число Фибоначчи

Но не всё так просто - считать нужно рекурсивным алгоритмом, в котором рекурсия осуществляется не на функциях, а на программах

Для подсчёта **N**-го числа Фибоначчи вы должны из своей программы позвать свою же программу от параметров `N-1` и `N-2`, получить результаты, сложить их и вернуть сумму

Задание №28 - посчитать **N**-ое число Фибоначчи

Но не всё так просто - считать нужно рекурсивным алгоритмом, в котором рекурсия осуществляется не на функциях, а на программах

Для подсчёта **N**-го числа Фибоначчи вы должны из своей программы позвать свою же программу от параметров **N-1** и **N-2**, получить результаты, сложить их и вернуть сумму

Это практическое руководство строго полагается на **POSIX** окружение - в **OS Windows** , разумеется, тоже можно решить эту задачу, но тестовая система работает на **Linux**

Linux процессы

Каждая запущенная программа это ***процесс*** - сложная структура данных операционной системы, у которой есть:

1) ***PID (process identifier)*** - уникальный идентификатор (номер) процесса
2) Адресное пространство
3) Состояние регистров, включая **SP** и **PC**
4) Файловые дескрипторы
5) …

Linux процессы

Каждая запущенная программа это ***процесс*** - сложная структура данных операционной системы, у которой есть:

1) ***PID (process identifier)*** - уникальный идентификатор (номер) процесса
2) Адресное пространство
3) Состояние регистров, включая **SP** и **PC**
4) Файловые дескрипторы
5) …

У всех процессов, кроме одного, есть ***родительский процесс*** - тот, который его создал

Init

Init - первый процесс, создаваемый ядром операционной системы особым образом - именно он порождает все остальные процессы, включая командную оболочку, которая в итоге порождает процесс вашей программы, когда вы запускаете её

Init

Init - первый процесс, создаваемый ядром операционной системы особым образом - именно он порождает все остальные процессы, включая командную оболочку, которая в итоге порождает процесс вашей программы, когда вы запускаете её

У всех остальных процессов есть родительский - таким образом процессы образуют ***дерево процессов***

Порождение процессов

Каждый процесс порождается в одну или две стадии:

1) ***fork*** - операционная система создаёт копию родительского процесса с небольшими отличиями
2) ***exec*** - процесс подменяет свой образ, превращаясь из копии в другую (или ту же) программу

Порождение процессов

Каждый процесс порождается в одну или две стадии:

1) ***fork*** - операционная система создаёт копию родительского процесса с небольшими отличиями
2) ***exec*** - процесс подменяет свой образ, превращаясь из копии в другую (или ту же) программу

`fork` стадия обязательна - это и есть создание нового процесса

Порождение процессов

Каждый процесс порождается в одну или две стадии:

1) ***fork*** - операционная система создаёт копию родительского процесса с небольшими отличиями
2) ***exec*** - процесс подменяет свой образ, превращаясь из копии в другую (или ту же) программу

`fork` стадия обязательна - это и есть создание нового процесса

`exec` стадия нужна, если вы хотите, чтобы новый процесс выполнял другую программу (или ту же, но сначала или с другими аргументами)

pid_t

Целый тип данных, определённый в **sys\types.h** , использующийся для хранения **PID** процессов

`pid_t fork()`

Функция, определенная в **`unistd.h`** , делающая **`fork`** текущего процесса, создавая ***дочерний процесс (child process)***

`pid_t fork()`

Функция, определенная в **`unistd.h`** , делающая **`fork`** текущего процесса, создавая ***дочерний процесс (child process)***

Родительский и дочерний процессы совпадают практически во всем - у них одинаковое состояние всей памяти, файловых дескрипторов, регистров, **`SP`** и **`PC`**

`pid_t fork()`

Функция, определенная в **`unistd.h`** , делающая **`fork`** текущего процесса, создавая ***дочерний процесс (child process)***

Родительский и дочерний процессы совпадают практически во всем - у них одинаковое состояние всей памяти, файловых дескрипторов, регистров, **`SP`** и **`PC`**

Единственное, чем они отличаются - это **`PID`** (дочернему процессу присваивается новый) и возвращаемым значением из функции **`fork()`** - родительский процесс получает **`PID`** дочернего, а дочерний - **`0`**

pid_t fork()

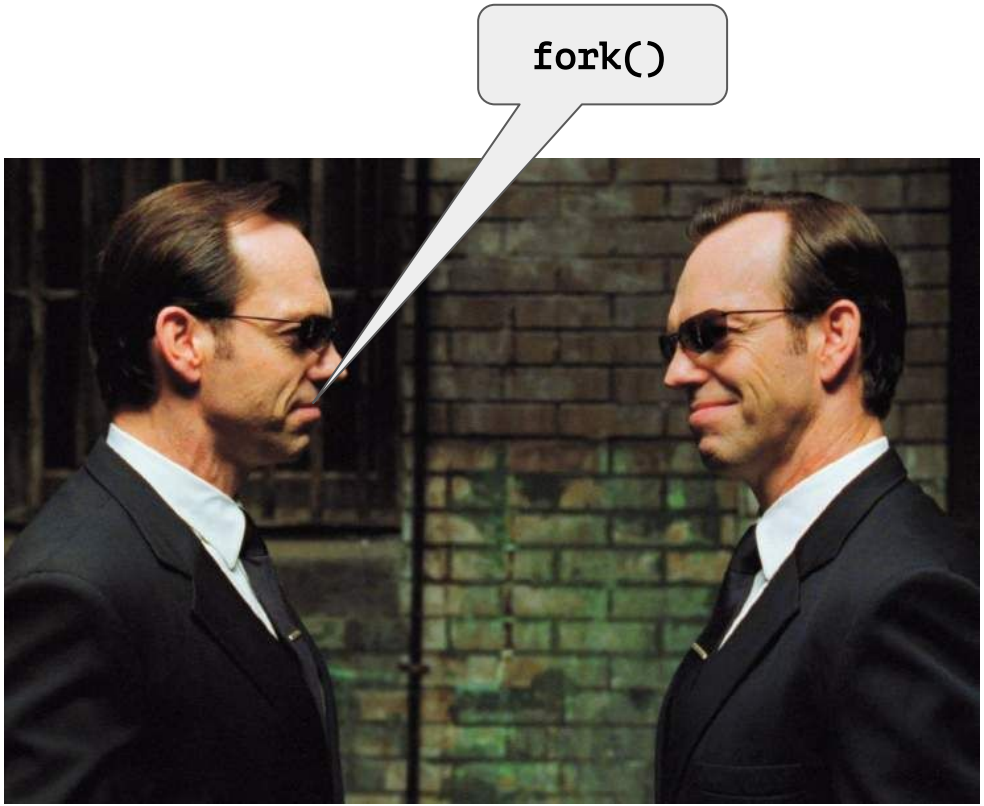
```
pid_t child = fork();
if (child == 0) {
    // Child process will start from this point
} else {
    // Parent process will continue from this point
}
```

pid_t fork()

```
pid_t child = fork();
if (child == 0) {
    // Child process will start from this point
} else {
    // Parent process will continue from this point
}
```

Механика очень похожа на **setjmp/longjmp**


fork()


fork()

exec

Вторая фаза создания нового процесса состоит в том, что дочерний процесс, узнав, что он дочерний, подменяет себя на новую программу вызовом одной из функций семейства **exec**

exec

Вторая фаза создания нового процесса состоит в том, что дочерний процесс, узнав, что он дочерний, подменяет себя на новую программу вызовом одной из функций семейства **exec**

В этот момент процесс получает новое адресное пространство и начинает исполнение новой программы в нём

Часть файловых дескрипторов при этом сохраняется, позволяя взаимодействовать родительскому и дочернему процессам

```
int execl(const char* pathname, const char* arg, ... /*, NULL */)
```

```
int execl(const char* pathname, const char* arg, ... /*, NULL */)
```

`pathname` - путь до исполняемого файла запускаемой программы

```
int execl(const char* pathname, const char* arg, ... /*, NULL */)
```

pathname - путь до исполняемого файла запускаемой программы

arg - нулевой аргумент, который практически всегда должен совпадать с **pathname**

```
int execl(const char* pathname, const char* arg, ... /*, NULL */)
```

pathname - путь до исполняемого файла запускаемой программы

arg - нулевой аргумент, который практически всегда должен совпадать с **pathname**

Далее идёт произвольное количество аргументов строкового типа, обязательно заканчивающиеся **NULL**, которые будут переданы запускаемой программе в **argv**

```
int execl(const char* pathname, const char* arg, ... /*, NULL */)
```

pathname - путь до исполняемого файла запускаемой программы

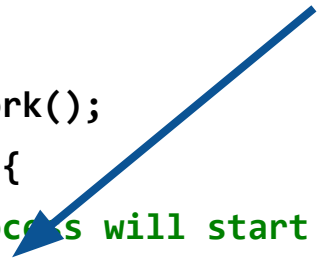
arg - нулевой аргумент, который практически всегда должен совпадать с **pathname**

Далее идёт произвольное количество аргументов строкового типа, обязательно заканчивающиеся **NULL**, которые будут переданы запускаемой программе в **argv**

Для задания №28 вам будет достаточно этой функции; для самостоятельного изучения - оставшееся семейство **exec** функций: **execlp**, **execle**, **execv**, **execvp**, **execve**

```c
pid_t child = fork();
if (child == 0) {
    // Child process will start from this point
    execl("foo", "foo", "bar", "baz", NULL);
    exit(1);
}
// Parent process will continue from this point
```

Вызов программы **foo** с аргументами **foo**, **bar** и **baz**



```
pid_t child = fork();
if (child == 0) {
    // Child process will start from this point
    execl("foo", "foo", "bar", "baz", NULL);
    exit(1);
}
// Parent process will continue from this point
```

Вызов программы **foo** с аргументами **foo**, **bar** и **baz**

```
pid_t child = fork();
if (child == 0) {
    // Child process will start from this point
    execl("foo", "foo", "bar", "baz", NULL);
    exit(1);
}
// Parent process will continue from this point
```

Это код достижим только в случае ошибки (её можно узнать из возвращаемого значения **execl**)


execl(...)

Создание процесса в два шага

Новый процесс не всегда делает **`exec`**, в некоторых сценариях межпроцессного взаимодействия копия процесса выполняет какую-то отдельную подзадачу

Создание процесса в два шага

Новый процесс не всегда делает **`exec`**, в некоторых сценариях межпроцессного взаимодействия копия процесса выполняет какую-то отдельную подзадачу

Во время **`fork`** память не копируется - одни и те же участки памяти одновременно принадлежат двум процессам в режиме **`read-only`** ; участок скопируется только при попытке изменить его

Такая механика называется ***copy-on-write***

Создание процесса в два шага

Новый процесс не всегда делает **`exec`**, в некоторых сценариях межпроцессного взаимодействия копия процесса выполняет какую-то отдельную подзадачу

Во время **`fork`** память не копируется - одни и те же участки памяти одновременно принадлежат двум процессам в режиме **`read-only`** ; участок скопируется только при попытке изменить его

Такая механика называется ***copy-on-write***

Между **`fork`** и **`exec`** также можно манипулировать файловыми дескрипторами для того, чтобы процессы могли взаимодействовать

Межпроцессное взаимодействие

Адресные пространства процессов изолированы друг от друга, поэтому любое взаимодействие нужно совершать через объекты операционной системы: файлы, pipe (каналы), аргументы и возвращаемое значение, сигналы, …

Межпроцессное взаимодействие

Адресные пространства процессов изолированы друг от друга, поэтому любое взаимодействие нужно совершать через объекты операционной системы: файлы, pipe (каналы), аргументы и возвращаемое значение, сигналы, …

Использование любого из этих инструментов потребует дальнейшего погружения в **POSIX** специфику, поэтому предлагается использовать простой и известный вам механизм аргументов и возвращаемого значения

Возвращаемое значение ограничено байтом, тесты рассчитаны на это (не больше 12-го числа)

Два режима запуска

Ваша программа становится зависимой от аргументов и работает в двух режимах:

1) **`argc == 1`** - запуск из системы тестирования, нужно прочитать число из входящего потока, сделать рекурсивный шаг и вывести результат в выходящий поток

Два режима запуска

Ваша программа становится зависимой от аргументов и работает в двух режимах:

1) **`argc == 1`** - запуск из системы тестирования, нужно прочитать число из входящего потока, сделать рекурсивный шаг и вывести результат в выходящий поток

2) **`argc == 2`** - запуск из рекурсии, нужно взять аргумент, сконвертировать его в число, сделать рекурсивные шаги и вернуть результат в возвращаемое значение

Получение возвращаемого значения

Во многих сценариях, включая эту задачу, родительскому процессу нужно дождаться результата от дочернего процесса и прочитать его возвращаемое значение

Получение возвращаемого значения

Во многих сценариях, включая эту задачу, родительскому процессу нужно дождаться результата от дочернего процесса и прочитать его возвращаемое значение

Для этого вам нужно использовать функцию **`waitpid`** и макрос **`WEXITSTATUS`** (самостоятельное изучение)

Альтернативные способы решения

Возвращаемое значение - очень ограниченный способ взаимодействия; для этой задачи его достаточно, но большинство других сценариев взаимодействия процессов требуют привлечения более сложных механизмов

Альтернативные способы решения

Возвращаемое значение - очень ограниченный способ взаимодействия; для этой задачи его достаточно, но большинство других сценариев взаимодействия процессов требуют привлечения более сложных механизмов

Рекомендуется решить задачу, используя ***каналы (pipe)***

Альтернативные способы решения

Возвращаемое значение - очень ограниченный способ взаимодействия; для этой задачи его достаточно, но большинство других сценариев взаимодействия процессов требуют привлечения более сложных механизмов

Рекомендуется решить задачу, используя ***каналы (pipe)***

Также можете поискать решение, не использующее **`exec`**, работающее только на **`fork`**