Лабораторные работы по курсу
Операционные системы

Лабораторная работа 3
«Процессы и межпроцессное взаимодействие»

2021

Оглавление

Список сокращений

ОС – Операционная Система

IPC – Inter Process Commutation

MMU – Memory Management Unit

PID – Process Identifier

PPID – Parent Process Identifier

FIFO – First Input First Output

1. Введение

Концепция процессов является одной из наиболее важных абстракций в ОС [1]. Именно за счет ее реализации современные вычислительные системы способны параллельно работать над разными задачам. При этом разработка многопроцессных программных комплексов, в особенности для встраиваемых систем, требует от программиста не только полного понимания поведения процессов, но и владения соответствующим инструментарием.

Целью данной лабораторной работы является освоение навыков работы с процессами и межпроцессным взаимодействием на языке *C* для встраиваемых систем на примере лабораторного стенда на базе микрокомпьютера *Raspberry Pi*.

2. Процессы в ОС *Linux*

2.1. Понятие процесса

Применение ОС позволяет хранить в памяти вычислительной системы большое количество разнообразных прикладных программ. Каждая программа является только совокупностью набора машинных инструкций и сопутствующих к ним данных, которая не может обрабатывать информацию до тех пор, пока она не будет запущена на исполнение (вызвана). При вызове программы ОС выделяет необходимую для исполнения программы память и запускает последовательное выполнение ее инструкций на микропроцессоре. Помимо выделения памяти под данные программы, ОС так же выделяет дополнительную память, необходимую для корректной работы программы и контроля ее исполнения: глобальные переменные, локальные переменные, стек, открытые программой дескрипторы файлов, регистры общего назначения, счетчик команд и т. д. Исполняемая в текущий момент времени программа называется «процессом», а вся выделенная под нее память – «контекстом процесса».

ОС *Linux* является многопроцессной, то есть поддерживает запуск большого числа процессов параллельно. Ввиду того, что количество одновременно запущенных в ОС процессов, как правило, превосходит количество доступных аппаратных устройств, способных производить вычисления, невозможно действительно исполнять инструкции всех процессов параллельно. Поэтому подсистема управления процессами ядра ОС создает видимость параллельности – она исполняет машинные инструкции каждого процесса «квазипараллельно»: небольшими порциями за короткий промежуток времени, перебирая процессы по очереди.

Каждый процесс может находиться в одном из пяти состояний, которые обозначаются латинскими буквами:

- *R* – процесс исполняется, либо ожидает очереди на исполнение;
- *D* – непрерываемый сон, процесс остановлен до возникновения определенного события;
- *S* – прерываемый сон, процесс ожидает события или сигнала;
- *T* – процесс остановлен;

- *Z* – процесс-зомби, он уже завершился, но не передал свой код завершения.

В ОС *Linux* все процессы разделяются на «системные» (являются частью ядра) и «пользовательские». Их ключевое отличие заключается в способе обращения к памяти: системные процессы могут обращаться ко всей области памяти и адресуют свои данные по ее физическим адресам («*kernelspace*»), а пользовательские процессы могут обращаться только к строго отведенным для них участкам и делать это по «виртуальным» адресам («*userspace*»). Пояснение приведено на Рисунке 1.

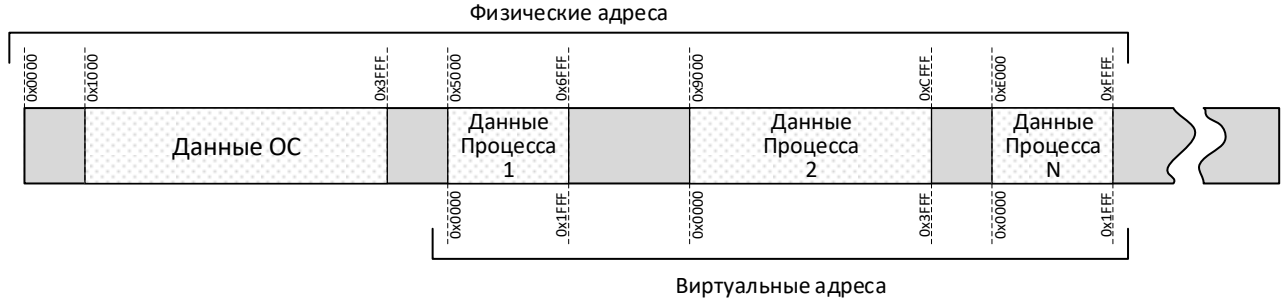

Рисунок 1. Разделение памяти

Пользовательские процессы используют одинаковые виртуальные адреса, которые преобразуются в адреса различных участков памяти. Преобразования между виртуальными и физическими адресами для каждого пользовательского процесса происходит с помощью аппаратного блока *MMU* [2], являющегося частью микропроцессора.

Организация хранения контекста пользовательских процессов через виртуальные адреса имеет несколько преимуществ. Во-первых, каждая программа всегда обращается к одним и тем же адресам вне зависимости от того, где реально расположены ее данные, что позволяет оптимально распределять память между процессами и не требует перекомпоновки программы из объектных файлов перед каждым вызовом. Во-вторых, изолированные области данных между процессами повышают стабильность системы – ни один процесс не может испортить данные другого процесса.

Но применение виртуальных адресов не позволяет процессам обмениваться данными между собой напрямую, что требует от ОС наличия механизмов межпроцессного взаимодействия (*IPC – InterProcess Communications*) [1] через ядро.

Для управления процессами ОС *Linux* присваивает каждому процессу при запуске свой уникальный номер, который называется *PID* (*Process Identifier*). Нулевой *PID* присваивается системному процессу ядра ОС (*kernel*). Остальные *PID* от 1 и до максимально возможного значения циклически присваиваются остальным процессам. При достижении максимально возможного значения счетчик *PID* сбрасывается и начинается заново с единицы. Максимально возможное значение PID хранится в файле «*/proc/sys/kernel/pid_max*».

2.2. Иерархия процессов

В отличии от ОС *Windows*, в которой процессы не имеют организованной структуры, подсистема управления процессами в ядре *Linux* построена по принципу ориентированного дерева: каждый процесс может быть запущен только одним из ранее запущенных процессов.

В вершине дерева находится процесс «*kernel*», который был запущен загрузчиком ОС. Процесс «*kernel*» после запуска порождает дочерний процесс «*init*» (ему присваивается *PID* равный 1). Процесс «*init*» проверяет конфигурационные файлы и порождает командную оболочку (по умолчанию используется «*bash*»), с помощью которой пользователь, вводя команды, может порождать новые дочерние процессы, которые, в свою очередь, тоже могу порождать новые дочерние процессы и т. д.

Для отслеживания дерева процессов у каждого процесса в системе помимо его собственного *PID* так же есть данные о его родительском процессе – *PPID* (*Parent Process Identifier*). В случае, если родительский процесс завершился раньше дочернего, *PPID* у последнего изменяется на значение «1», соответствующего процессу «*init*», поскольку процесс «*init*» не может быть завершен до тех пор, пока работает ОС.

2.3. Сигналы

Механизм сигналов в ОС *Linux* представляет собой простейшую систему оповещений процессов о каких-то событиях, которые могут повлиять на ход их исполнения [3]. Механизм сигналов по своей сути схож с механизмом аппаратных прерываний [4]: любой из процессов в ОС может сгенерировать отправку сигнала остальным процессам, а они, в свою очередь, при получении сигнала могут незамедлительно прервать свое штатное исполнение и перейти в соответствующую функцию-обработчик. Сигналы передаются асинхронно: процесс-источник сигнала не будет дожидаться перемещения в обработчик процессов-приемников.

При запуске процесса ОС *Linux* автоматически создает для него набор обработчиков сигналов по умолчанию. Стандарт *POSIX.1* определяет 28 сигналов, которые могут быть сформированы в ОС. Они приведены в Таблице Таблица 1. Для каждого сигнала обозначены его уникальный код, имя, краткое описание причины возникновения и основная функция обработчика по умолчанию.

Таблица 1. Сигналы по стандарту POSIX.1

Код	Название	Причины возникновения	Функция обработчика процесса по умолчанию
1	*SIGHUP*	Закрытие терминала	Завершение
2	*SIGINT*	Сигнал прерывания с терминала Ctrl+C	Завершение
3	*SIGQUIT*	Сигнал выхода с терминала Ctrl+\	Завершение с дампом памяти
4	*SIGILL*	Недопустимая инструкция процессора	Завершение с дампом памяти
5	*SIGTRAP*	Ловушка трассировки или брейкпоинт	Завершение с дампом памяти
6	*SIGABRT*	Сигнал, создаваемый функцией abort()	Завершение с дампом памяти
8	*SIGFPE*	Ошибка арифметической операции	Завершение с дампом памяти

9	*SIGKILL*	Безусловное завершение	Завершение
10	*SIGBUS*	Неправильное обращение в память	Завершение с дампом памяти
11	*SIGSEGV*	Нарушение при обращении в память	Завершение с дампом памяти
12	*SIGSYS*	Неправильный системный вызов	Завершение с дампом памяти
13	*SIGPIPE*	Запись в разорванное соединение (пайп, сокет)	Завершение
14	*SIGALRM*	Сигнал истечения времени, заданного alarm()	Завершение
15	*SIGTERM*	Сигнал завершения	Завершение
16	*SIGUSR1*	Пользовательский сигнал №1	Завершение
17	*SIGUSR2*	Пользовательский сигнал №2	Завершение
18	*SIGCHLD*	Дочерний процесс завершен или остановлен	Игнорируется
20	*SIGTSTP*	Сигнал остановки с терминала Ctrl+Z	Остановка процесса
21	*SIGURG*	На сокете получены срочные данные	Игнорируется
22	*SIGPOLL*	Событие, отслеживаемое функцией poll()	Завершение
23	*SIGSTOP*	Остановка выполнения процесса	Остановка процесса
25	*SIGCONT*	Продолжение выполнения процесса	Продолжить выполнение
26	*SIGTTIN*	Попытка чтения с терминала фоновым процессом	Остановка процесса
27	*SIGTTOU*	Попытка записи на терминал фоновым процессом	Остановка процесса
28	*SIGVTALRM*	Истечение «виртуального таймера»	Завершение
29	*SIGPROF*	Истечение таймера профилирования	Завершение
30	*SIGXCPU*	Процесс превысил лимит процессорного времени	Завершение с дампом памяти
31	*SIGXFSZ*	Процесс превысил допустимый размер файла	Завершение с дампом памяти

Все сигналы могут быть проигнорированы или перехвачены собственной функций-обработчиком, за исключением сигналов *SIGKILL* и *SIGSTOP*.

Большинство сигналов возникает из-за различного рода ошибок, после которых дальнейшее выполнение процесса невозможно. Например, сигнал *SIGILL* формируется при попадании в микропроцессор несуществующей аппаратной инструкции от процесса, сигнал *SIGFPE* формируется при ошибке вычисления с плавающей запятой и т. д. Некоторые сигналы служат для управления поведением процесса. Например, с помощью сигналов *SIGSTOP* и *SIGCONT* можно приостанавливать (состояние «*T*») и возобновлять (состояние «*R*») выполнение процесса, а сигнал *SIGINT* предназначен для вызова нештатного прерывания процесса. Так же в *POSIX.1* имеются два свободных пользовательских сигнала *SIGUSR1* и *SIGUSR2*, назначение которых неопределенно, и они могут использоваться в любых целях.

На практике существует расхождение между теми сигналами, которые описаны в стандарте *POSIX.1* и теми, которые реально реализованы в дистрибутивах ОС *Linux*. Причем состав сигналов может изменяться в зависимости от дистрибутива. Реализованный в дистрибутиве список сигналов можно посмотреть в терминале с помощью вызова команды

```
kill -l
```

3. Управление процессами в командной оболочке

3.1. Мониторинг процессов

Информация, формируемая ОС Linux относительно процессов, расположена в виртуальной директории «*/proc*». Для каждого запущенного процесса создается поддиректория с именем *PID*, в которой располагаются виртуальные файлы, содержащие служебную информацию о процессе: текущее состояние, объем выделенной памяти, использованные системные вызовы и т. д.

Информация из «*/proc*» может быть получена с помощью утилиты «*cat*», позволяющей выводить содержимое файлов на экран. Однако более удобным способом ее получения является стандартизированная *POSIX* утилита «*ps*». Она самостоятельно разбирает файлы из каталога «*/proc*» и преобразует информацию в удобном для пользователя формате. В Таблице Таблица 2 представлены наиболее часто используемые опции для данной утилиты.

Таблица 2. Опции утилиты ps

Опция	Значение
-e	отобразить информацию обо всех процессах
-a	отобразить информацию обо всех процессах, связанных с терминалом
-d	отобразить информацию обо всех процессах, кроме главных системных процессов
-f	расширение информации
-x	отобразить информацию о процессах, отсоединенных от терминала
-u	фильтрация по имени пользователя, который запустил утилиту

Утилита «*ps*» может быть вызвана как без опций вообще, так и с одной или несколькими:

```
ps          # отобразить процессы текущего пользователя, привязанные к терминалу
ps -e       # отобразить все процессы
ps -aux     # отобразить все процессы в расширенном формате
ps -fax             # отобразить процессы с деревом вызовов
```

Большее количество опций утилиты «*ps*» можно узнать через команду

```
man ps
```

Для просмотра статистики по запущенным процессам в режиме реального времени существует утилита «*top*». После ее вызова в командной оболочке будет отображаться таблица с запущенными процессами и сводная информация по ним в схожем формате с «Диспетчером задач» в ОС Windows. Выход из утилиты *top* осуществляется с помощью нажатия клавиши «*q*».

3.2. Создание процессов

В Лабораторной работе №1 был показан ввод команд и запуск процессов в командной оболочке «на переднем плане». При таком вводе после порождения дочернего процесса командная оболочка блокируется и недоступна для ввода новых команд до тех пор, пока не будет завершен текущий дочерний процесс. При этом стандартные потоки

ввода-вывода командной оболочки перенаправляются в стандартные потоки ввода-вывода дочернего процесса. Очевидно, что при таком режиме в одной командной оболочке невозможно запустить сразу несколько процессов, работающих квазипараллельно, что не позволяет в полной мере воспользоваться поддержкой многопроцессности ОС.

Одним из очевидных способов решения данной проблемы является порождение в ОС еще одного экземпляра командной оболочки с помощью еще одного терминала, однако это не всегда бывает возможно или удобно для пользователя. Наиболее удобным вариантом будет запуск процессов «в фоновом режиме», который так же поддерживается командной оболочкой ОС *Linux*. Для запуска любой команды в фоновом режиме необходимо в конце команды поставить знак «&»:

```
<command> &
```

Например,

```
cp -r /home/pi/data /home/ivan/data &  # копирование данных
```

После запуска процесса в фоновом режиме командная оболочка будет сразу доступна для ввода новых команд – стандартный поток ввода *stdin* дочернего процесса будет откреплен от командной оболочки. При этом стандартные выводы *stdout* и *stderr* дочернего процесса будут по-прежнему выводиться в терминал.

При запуске нового процесса в фоновом режиме в терминале появится сообщения формата

```
[jobspec] PID,
```

где *jobspec* – номер запущенного фонового процесса по порядку начиная с 1, а *PID* – идентификационный номер фонового процесса. При штатном завершении фонового процесса в терминале появится сообщение формата

```
[jobspec]+ Done                 <command>
```

Просмотреть все запущенные фоновые процессы в данной командной оболочке можно с помощью команды

```
jobs -l
```

Вернуть исполнение процесса из фонового режима на передний план можно с помощью команды

```
fg %jobspec
```

Если в терминале запущен всего один фоновый процесс, то аргумент указывать не обязательно.

3.3. Завершение процессов

В каждом процессе предусмотрено его самостоятельное завершение либо при его попадании в «*return*» в функции «*main*», либо при использовании им библиотечного вызова «*exit*» (более подробно об этом рассказано в п. 3.1 Лабораторной работы №2). Однако иногда возникает необходимость принудительного завершения процесса, не дожидаясь его штатного успешного или неуспешного окончания. Например, если процесс завис в процессе работы или потерял актуальность. Для принудительного завершения процесса в командной оболочке может быть использована утилита «*kill*». Она отправляет заданный в одном аргументе сигнал (см. п. 2.3) в заданный в другом аргументе процесс:

```
kill [-s sigspec | -n signum | -sigspec] pid | jobspec
```

При использовании «*kill*» сигнал может быть указан по его названию (ключ «-s») или номеру (ключ «-n»). Допускается указывать сразу после «-» номер или имя сигнала. Если сигнал не указан, то по умолчанию посылается *SIGTERM* (-15). Большинство сигналов приводят к завершению процесса, если они не перехвачены собственными функциями-обработчиками, в которых завершение будет отключено. Сигнал *SIGKILL* всегда гарантированно завершит процесс.

Процесс-получатель сигнала указывается через его *PID* или через уникальный номер «*jobspec*», присваиваемый процессу, запущенному в фоновом режиме. Примеры использования утилиты «*kill*»:

```
kill 1234              # отправка SIGTERM в процесс с PID=1234 (мягкое завершение)
kill -SIGKILL 1234     # отправка SIGKILL в процесс с PID=1234 (принудительное завершение)
kill -9 %1             # отправка SIGKILL в фоновый процесс с jobspec = 1
```

Важно понимать, что в общем случае с помощью утилиты «*kill*» возможно не только посылать сигналы на завершения процесса, а отправлять любые сигналы, которые могут обрабатываться в процессе в штатном режиме (например, *SIGUSR1* или *SIGCHLD*), то есть с помощью «*kill*» возможно осуществлять примитивный информационный обмен между процессами.

4. Обмен данными между процессами в командной оболочке

Обмен сигналами позволяет кодировать передаваемую информацию от процесса к процессу только типом сигнала. Но для эффективного взаимодействия процессов требуется наличие *IPC* (см. п. 2.1): зачастую выходные данные одного процесса являются входными данными другого или других процессов, а одному процессу могут потребоваться выходные данные от одного или нескольких других процессов.

В ОС *Linux IPC* представляет собой системные и библиотечные вызовы, с помощью которых у ядра системы можно «заказать» копирование данных из одной виртуальной памяти в другую. Для этого существует два основных механизма:

- неименованные каналы (так же называемые «*pipe*», «пайпы», «труба») и
- именованные каналы (так же называемые «*fifo*», «*mkfifo*»).

Оба механизма могут использоваться как внутри приложений, так и для процессов, вызываемых в командной оболочке. Несмотря на единую цель этих механизмов, существуют некоторые отличия в их использовании, что делает каждый из них более или менее удобным в зависимости от ситуации.

4.1. Неименованные каналы

Про механизм неименованных каналов ранее частично было упомянуто в пункте 3.2.2 Лабораторной работы №1.

Он представляет собой замыкание стандартного вывода *stdout* одного процесса на стандартный ввод *stdin* другого процесса. В командной оболочке неименованные каналы между процессами создаются с помощью символа «|», например:

```
program1 | program2 | program3 | … | programN
```

Данная команда интерпретируется следующим образом: отправленные процессом «*program1*» данные в *stdout* (с помощь, например, функции «*printf*»), автоматически появятся в *stdin* у процесса «*program2*» (которые могут быть считаны с помощью, например, функции «*fgets*»). В свою очередь данные из *stdout* процесса «*program2*» попадут в *stdin* процесса «*program3*» и т. д.

Такая организация передачи данных чрезвычайно удобна, когда необходима конвейерная обработка данных, в которой первый процесс выступает источником, а каждый последующий процесс обрабатывает данные предыдущего. Например, если требуется удалить все поддиректории с пометкой «*tmp*» в рабочей директории, то при вызове команды

```
ls | grep "tmp" | xargs rm -r
```

программа «*ls*» сформирует список существующих файлов и подкаталогов в текущем каталоге, затем программа «*grep*» оставит из них только те, что содержат в названии подстроку «*tmp*», а затем программа «*xargs*» сформирует из них список аргументов для программы «*rm*», которая их удалит.

Преимуществом применения неименованных каналов в командной оболочке является чрезвычайно скромный синтаксис и легкость организации конвейерной обработки. Однако при применении неименованных каналов невозможно:

- распараллелить *stdout* процесса-источника на несколько *stdin* процессов-приемников,
- отправить в *stdin* процесса-приемника данные от нескольких процессов-источников,
- обратиться к промежуточным результатам внутри конвейера: каждое соединение процессов через «|» создают дескрипторы файлов, указывающих на промежуточный буфер, которые доступны только соединенным через них процессам.

4.2. Именованные каналы

В случаях, когда требуется доступ к промежуточным данным или требуется передача в процесс-приемник данных от нескольких процессов-источников целесообразно использовать механизм именованных каналов.

Именованный канал представляет собой некоторый псевдо-файл в файловой системе ОС, который может быть создан в каталоге, в него могут быть записаны данные, из него могут быть считаны данные, и он может быть удален. Однако у данного псевдо-файла есть несколько отличий от обычного файла.

Во-первых, физически он не располагается на диске и в файловой системе. Путь к данному файлу представляет собой ссылку к внутреннему буферу в оперативной памяти, созданному ядром ОС и ассоциированным с именем этого файла.

Во-вторых, данные в именованном канале записываются и считываются по принципу *FIFO* (*First Input First Output*), т. е. первыми считываются те данные, которые были записаны первыми, чтение из произвольного места невозможно. После считывания

данных любым из процессов они удаляются из именованного канала средствами ОС. Повторно считать данные в именованном канале невозможно.

В-третьих, ввиду того, что именованный канал располагается в оперативной памяти, существует риск его переполнения, если несколько процессов будут записывать данные в именованный канал, но ни один из процессов не будет их считывать. Поэтому операция записи в именованный канал является блокируемой: ОС не даст процессу записать данные, если тот же именованный канал не открыт на чтение хотя бы в одном процессе. Стоит оговориться, что «под открыт на чтение» подразумевается именно системный вызов «*open*» с атрибутами «O_*RDONLY*» или «O_*RDWR*», использование системного вызова «*read*» необязательно (см. п. 3.4.1 Лабораторной работы №2). Так же стоит отметить, что блокировка является поведением ОС по умолчанию, но такое поведение может быть изменено.

В командной оболочке именованный канал создается с помощью команды «mkfifo»:

```
mkfifo <namedpipe>
```

где «namedpipe» – имя создаваемого именованного канала.

Дальнейшая работа с именованным каналом в командной оболочке происходит точно так же, как и с любым другим файлом: в него могут быть перенаправлены потоки для записи, и он может быть считан. Для примера рассмотрим следующий *bash*-скрипт:

```
mkfifo gpiostate

while [ true ]; do
date | tr -s '\n' ' ' > gpiostate;
echo -n "PINS STATE: " > gpiostate;
cat /sys/class/gpio/gpio18/value | tr -d '\n' > gpiostate;
cat /sys/class/gpio/gpio4/value > gpiostate;
sleep 1;
done &

tail -f gpiostate
Sat 7 Aug 06:28:31 BST 2021 PINS STATE: 01
Sat 7 Aug 06:28:32 BST 2021 PINS STATE: 01
Sat 7 Aug 06:28:33 BST 2021 PINS STATE: 01
Sat 7 Aug 06:28:34 BST 2021 PINS STATE: 01
Sat 7 Aug 06:28:35 BST 2021 PINS STATE: 01
…^C

rm gpiostate
kill %1
```

В начале скрипта с помощью «*mkfifo*» создается именованный канал в текущем рабочем каталоге с именем «*gpiostate*». Затем в цикле «*while*» раз в секунду в именованный канал записываются:

- текущая дата и время, в которой последний символ в строке «*n*» (перенос строки) заменяется на пробел с помощью программы «*tr*»,
- строка «*PINS STATE:* » без переноса строки (опция «*-n*» у «*echo*»)
- текущее значение пина 18 *GPIO*, ранее настроенного на вход (символ переноса строки удаляется),

- текущее значение пина 4 *GPIO*, так же ранее настроенного на вход.

Описанный цикл запускается в фоновом режиме. Его работа будет заблокирована до тех пор, пока в терминале не будет вызвана команда «*tail*», непрерывно считывающая данные из именованного канала (после вызова команды «*tail*» показан ее вывод в терминал). После прерывания команды «*tail*» сочетанием клавиш «*Ctrl+C*» («*^C*») удаляется именованный канал и завершается фоновый процесс формирования данных.

Создаваемый ядром буфер памяти как для неименованных, так и для именованных каналов не может существовать дольше, чем последний из использовавших его процессов. Когда все процессы, работающие с именованным каналом, закрывают все файловые дескрипторы, ассоциированные с ним, система освобождает ресурсы. Вся непрочитанная информация теряется. В то же время псевдо-файл именованного канала остается в файловой системе, и, если он не был специально удален, то его можно в дальнейшем повторно использовать для нового обмена данными.

5. Работа с процессами в приложениях

Так же, как и скрипты в командной оболочке, приложения на языке *C* способны управлять процессами и использовать механизмы межпроцессного обмена данными. Для этого в «*C*» необходимо использовать системные и библиотечные вызовы.

5.1. Создание процессов

5.1.1. Системный вызов *fork*

Системный вызов «*fork*» используется для создания дочернего процесса. При получении вызова *fork* от одного из процессов ОС создаст его дубликат: выделит новую область виртуальной памяти и перенесет весь пользовательский контекст исходного процесса в дочерний, за исключением:

- идентификатора процесса (дочернему процессу будет присвоен новый идентификатор);
- израсходованное время процессора (у дочернего процесса оно будет равно нулю);
- требующих обработки сигналов у процесса-родителя;
- блокированных файлов.

Вся остальная информация (глобальные и локальные переменные, счетчик команд, открытые дескрипторы файлов) будут идентичны. В дочернем процессе продолжится исполнение инструкций с момента вызова *fork* параллельно с родительским.

Системный вызов *fork* имеет следующую семантику:

```
#include <sys/types>
#include <unistd.h>

pid_t fork(void);
```

У него отсутствуют входные аргументы, а возвращаемое значение типа «*pid_t*» является *PID* процесса. Его конкретное значение зависит от того, в каком процессе значение было

возвращено: в родительском процессе возвращается *PID* дочернего процесса, а в дочернем процессе возвращается 0. Если произошла ошибка, то в обоих процессах возвращаемое значение будет равно «-1».

Если дочерний процесс завершится раньше родительского, а в родительском процессе не будет получен статус его завершения, то дочерний процесс перейдет в состояние «зомби»: фактически он не будет исполняться, но ОС не высвободит ресурсы, выделенные под него. Для того, чтобы окончательно завершить дочерний процесс в процессе-родителе должен быть использован системный вызов «*wait*» или «*waitpid*»:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Каждый из этих системный вызов приостанавливает вызов родительского процесса до фактического завершения дочернего процесса. Системный вызов «*wait*» ожидает завершение любого из дочерних процессов, передавая в «*int *status*» статус его завершения и возвращая его *PID*. Системный вызов «*waitpid*» позволяет конкретизировать дочерний процесс для ожидания с помощью указания его *PID* и опции ожидания через входные аргументы «*pid_t pid*» и «*option*».

Пример использования системного вызова «*fork*» приведен в Листинге 1. На основе возвращаемого значения «*pid_t*» вызовом «*fork*» разделена логика поведения родительского и дочернего процессов.

Листинг 1 – Пример функции fork

```
1.   pid_t pid = fork();
2.
3.   switch (pid) {
4.            case -1:
5.                     /* обработчик ошибки вызова fork */
6.            break;
7.            case 0:
8.                     /* код дочернего процесса */
9.            break;
10.           default:
11.                    /* код родительского процесса */
12.                    wait();
13.           break;
14.  }
```

Для обмена данными между родительским и дочерним процессами возможно создание неименованного канала с помощью системного вызова «*pipe*»:

```
#include <unistd.h>
int pipe(int filedes[2]);
```

При удачном открытии системный вызов «*pipe*» возвращает ноль. При ошибке он возвращает «-1», а код ошибки сохраняет в глобальную переменную «*errno*».

Входной аргумент «*int filedes[2]*» представляет собой указатель на массив типа «int» из двух элементов, в которых будут возвращены открытые дескрипторы для обмена данными. Дескриптор «*filedes[0]*» предназначен для чтения данных, а дескриптор «*filedes[1]*» для записи. Для работы с дескрипторами используются те же системные вызовы «*read*», «*write*» и «*close*», что и при работе с файлами.

По окончанию использования входного или/и выходного потока данных нужно закрыть соответствующий дескриптор с помощью системного вызова «*close*» для освобождения ресурсов ОС. При закрытии всех файловых дескрипторов ОС ликвидирует внутренний буфер, то есть время существования канала определяется временем работы с ним процессов. В Листинге 2 приведен пример создания неименованного канала при использовании системного вызова *fork*:

Листинг 2 – Пример создания неименованного канала

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {

int fd[2];
    if(pipe(fd) < 0){
        printf("Can\'t create pipe\n");
        exit(-1);
    }

    pid_t pid = fork();
         if (pid == -1) {
                  printf("Can\'t create fork\n");
        exit(-1);
         }

         switch (pid) {

                  // Дочерний процесс:
                  case 0:
                           close(fd[1]);
                           char resstring[14];
                           if(read(fd[0], resstring, 14) < 0){
                                    printf("Can\'t read string\n");
                                    exit(-1);
                           }
                           printf("%s\n",resstring);
                           close(fd[0]);
                           printf("Child exit\n");
                  break;

                  // Родительский процесс:
                  default:
                           close(fd[0]);
                           if(write(fd[1], "Hello, world!", 14) != 14){
                                    printf("Can\'t write all string\n");
                                    exit(-1);
                           }
                           close(fd[1]);
                           printf("Parent exit\n");
                  break;
         }

    return 0;
}
```

Вначале кода с помощью «*pipe*» создаются два дескриптора и сохраняются в массиве «*fd*». Затем с помощью системного вызова «*fork*» происходит порождение дочернего процесса и по значению возвращаемого номера процесса «*pid*» происходит разделение логики поведения между материнским и дочерним процессами.

В материнском процессе закрывается дескриптор на чтение «*fd[0]*» и происходит запись строки «*Hello, world!*» в дескриптор на запись «*fd[1]*». После записи в материнском процессе так же закрывается дескриптор на запись и выводится информационное сообщение о завершении работы в *stdout*.

В дочернем процессе закрывается дескриптор на запись «*fd[1]*» и считываются данные из дескриптора на чтение «*fd[0]*». После этого принятая строка от материнского процесса выводится в *stdout* и дочерний процесс завершает работу.

Из примера видно, что один канал используется только для одного направления передачи данных (в примере – от материнского к дочернему). Для создания двунаправленного потока обмена необходимо создание двух неименованных каналов.

5.1.2. Системные вызовы семейства *exec*

Системные вызовы семейства *exec* описаны в *POSIX.1* и позволяют запустить новый процесс поверх старого. При вызове в исполняемом процессе одной из функций семейства *exec* контекст исполняемого процесса будет полностью замещен контекстом вызываемого процесса, за исключением ранее открытых файлов, они останутся доступны для вызываемого процесса. Вызываемый процесс начнет исполнение с функции «*main*».

Семейство вызовов *exec* состоит из следующих функций:

```
#include <unistd.h>
int execlp(const char *file, const char *arg0, ... const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path, const char *arg0, ... const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path, const char *arg0, ... const char *argN,(char *)NULL, char *
envp[])
int execve(const char *path, char *argv[], char *envp[])
```

Возвращаемое значение типа «*int*» у каждой функции имеет смысл только при ошибке и равно «-1», а код ошибки будет сохранен в глобальную переменную «*errno*». При успешном вызове возвращаемое значение не имеет смысла ввиду уничтожения вызывающего процесса.

В качестве аргументов каждая из функций принимает на вход имя исполняемого файла и набор аргументов для его вызова. Различается формат аргументов: «*const char *file*» – это указатель на строку с именем исполняемого файла, «*const char *path*» – указатель на строку с полным путем до исполняемого файла, «*const char *arg0…N*» – указатели на строки с аргументами, «*const char *argv[]*» – указатель на массив передаваемых аргументов.

Вызовы *exec* могут использоваться в сочетании с системными вызовами *fork* для порождения дочерних процессов с контекстом, отличным от родительского. Пример приведен в Листинге 3 :

Листинг 3 – Пример вызова exec и fork

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(){

        pid_t pid = fork();
        if (result > 0) {
                printf("Parent process\n");
        } else {
                printf("Child process\n");
                        if (execle("/bin/cat", "/bin/cat",
"textfile.txt", 0, envp)== -1) {
                        printf("cannot run exec\n");
                        exit(-1);
                }
        }

        printf("Parent exit\n");
        return 0;

}
```

В данном примере контекст дочернего процесса замещается порождаемым с помощью вызова «*execle*» процессом «*/bin/cat*».

5.1.3. Библиотечный вызов popen

Библиотечный вызов «*popen*» является комбинацией системных вызов *fork*, *pipe* и *exec*. Он позволяет в исполняемом процессе породить дочерний процесс и связать с ним неименованный канал потока ввода или вывода:

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
```

Входные аргументы: «*const char *command*» – указатель на строку с вызываемым процессом, «*const char *type*» – указатель на строку с типом открытия канала обмена информацией. Канал является однонаправленным, поэтому «*const char *type*» может принимать только значения «*r*» (канал открыт на чтение) или «*w*» (канал открыт на запись).

Возвращаемое библиотечным вызовом «*popen*» значение типа «*FILE **» является стандартным потоком ввода-вывода, работа с которым может вестись функциями «*fgetc*», «*fscanf*», «*putc*», «*fputc*», «*fputs*», «*fprintf*», более подробно описанными в п. 3.5.1 Лабораторной работы №2.

Для закрытия потока необходимо использовать библиотечный вызов «*pclose*»:

```
#include <stdio.h>
int pclose(FILE *stream);
```

В случае ошибки его возвращаемое значение типа «*int*» равно «-1».

В Листинге 4 приведен пример использования библиотечных вызовов *popen* и *pclose*:

Листинг 4 – Пример вызова *popen* и *pclose*

```
        #include <stdio.h>
        int main () {

                // run subprocess:
                FILE *fp = popen("/bin/ls -a", "r");
                if (fp == NULL) {
```

```
7.                              fprintf(stderr, "error: cannot popen ls command.
8.          Sorry");
9.                              return -1;
10.                     }
11.
12.                     // read subprocess output:
13.                     char buf[256];
14.                     while (fgets(256, sizeof(256), fp) != NULL) printf("%s",
15.          buf);
16.
17.                     // close:
18.                     pclose(fp);
19.
20.                     return 0;
21.
22.         }
```

В приведенном выше примере вызывается дочерний процесс «*/bin/ls*», результат работы которого построчно считывается с помощью функции «*fgets*».

5.1.4. Библиотечный вызов *system*

Библиотечный вызов «*system*» имеет сигнатуру

```
#include <stdlib.h>

int system(const char * string);
```

Он запускает командную оболочку, вызывая в ней команду «*string*» *(«/bin/sh -c string»)*. До тех пор, пока командная оболочка не завершит работу вызывающий процесс будет приостановлен.

Библиотечный вызов *system* возвращает значение типа «*int*», которое при ошибках равно «-1». В противном случае он возвращает статус выполнения команды.

Примеры использования библиотечного вызова *system*:

```
int res = system("script.sh");
int res = system("pwd");
```

5.2. Работа с сигналами

5.2.1. Отправка сигналов

Для отправки сигналов используется системный вызов «*kill*»:

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

В случае успешной отправки сигнала «*kill*» в возвращаемом значении типа «*int*» будет 0, в случае неуспешной отправки «-1», а код ошибки будет записан в «*errno*».

Сигнала передается в процесс с *PID*, указанным в аргументе «*pid_t pid*». Если вместо *PID* указан «0», то сигнал передается каждом процессу, входящему в группу процесса-отправителя. Если в *PID* указан «-1», то сигнал будет послан всем процессам, кроме «*init*». С помощью значений меньше «-1» возможна отправка сигналов в группы процессов.

Аргумент «*int sig*» указывает тип отправляемого сигнала. Если «*sig*» равно нулю, то системный вызов «*kill*» не отправляет сигнал, а проверяет ошибки отправки.

Процесс так же может отправить сигнал самому себе с помощью библиотечного вызова «*raise*»:

```
#include <signal.h>

int raise(int sig);
```

в котором возвращаемое значение и входной аргумент «*int sig*» аналогичны системному вызову «*kill*».

5.2.2. Прием сигналов

Для замещения в процессе штатных обработчиков сигналов может быть использован интерфейс «*sigaction*». Созданный с его помощью обработчик будет присвоен сигналу однократно и будет вызываться до тех пор, пока не будет целенаправленно отменен.

Для прикрепления нового или получения информации о старом обработчике используется системный вызов «sigaction»:

```
#include <signal.h>
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

Входной аргумент «*signum*» – номер или название сигнала, для которого прикрепляется обработчик. Он может быть равен любому сигналу, за исключением «*SIGKILL*» и «*SIGSTOP*». Входные аргументы «*const struct sigaction *act*» и «*const struct sigaction *oldact*» используются для прикрепления нового или получения информации о старом обработчике. Если аргумент «*act*» ссылается не на нулевой адрес, то сигналу присваивается обработчик, который им описывается. Если аргумент «*oldact*» ссылается не на нулевой адрес, то в его адрес записывается информация о ранее присвоенном обработчике.

Возвращаемое значение *sigaction* типа «*int*» равно «0» в случае успешного завершения функции или «-1» в случае возникновения ошибки.

Структурой «*struct sigaction*», описывающая обработчик, имеет следующие основные поля:

```
#include <signal.h>
struct sigaction {
	void (*sa_handler) (int);
	sigset_t sa_mask;
	int sa_flags;
}
```

В поле «*sa_handler*» записывается указатель на функцию-обработчик с входным аргументом типа «int» для номера сигнала. В «*sa_handler*» может быть записан макрос «*SIG_DFL*» для установки обработчика по умолчанию или макрос «*SIG_IGN*» для игнорирования сигнала. Поле «*sigset_t sa_mask*» представляет собой битовую маску для тех сигналов, которые должны быть заблокированы во время вызова обработчика. Поле «*int sa_flags*» позволяет процессу модифицировать поведение сигнала и может принимать один или несколько флагов, объеденных логическим «ИЛИ»:

- *SA_NOCLDSTOP* – если сигнал равен *SIGCHLD*, то уведомление об остановке дочернего процесса получено не будет;

- *SA_NODEFER* или *SA_NOMASK* – не препятствовать повторному получению сигнала при его обработке;
- *SA_RESTHAND* – после прихода сигнала его обработчик сбрасывается в *SIG_DFL*;
- *SA_RESTART* – перезапуск системного вызова после возврата из обработчика сигнала. Если флаг не установлен, то системный вызов возвращает ошибку *EINTR*.

При необходимости модификации значения битовой маски «*sigset_t sa_mask*» может быть использован следующий набор библиотечных функций:

```
#include <signal.h>
int sigemptyset(sigset_t *set);                   // очистка битовой маски
int sigfillset(sigset_t *set);                    // установка блокировки всех сигналов
int sigaddset(sigset_t *set, int signum);         // добавление сигнала signum
int sigdelset(sigset_t *set, int signum);         // удаление сигнала signum
int sigismember(const sigset_t *set, int signum); // проверка наличия сигнала signum в set
```

Системный вызов «*sigprocmask*» позволяет проводить групповое редактирование сигналов в маске:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Входной аргумент «*const sigset_t *set*» является набором сигналов для модификации. Если он равен «*NULL*», то «*sigprocmask*» возвращает ранее установленное значение маски. В аргумент «*sigset_t *oldset*» возвращается ранее установленная маска сигналов. Если ее значения не требуются, то можно указать «*NULL*». Входной аргумент «*int how*» описывает действие для «*sigprocmask*»:

- *SIG_BLOCK* – сигналы из набора *set* блокируются,
- *SIG_UNBLOCK* – сигналы из набора *set* разблокируются,
- *SIG_SETMASK* – сигналы из набора *set* блокируются, остальные разблокируются.

Для просмотра сигналов, которые поступили в процесс, но не были обработаны можно использовать системный вызов «*sigpending*»:

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Поступившие и необработанные сигналы возвращаются в «*sigset_t *set*». В случае успеха возвращаемое значение типа «*int*» равно 0, в случае ошибки – «-1», а код ошибки записывается в «*errno*».

5.3. Обмен данными между процессами

5.3.1. Именованные каналы

Помимо командной оболочки, именованные каналы так же могут создаваться и использоваться в процессах. Для создания именованных каналов существует системный вызов «*mkfifo*»:

```
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>

int mkfifo(char *path, int mode);
```

Его входные аргументы: «*char *path*» – указатель на строку, содержащий путь и имя создаваемого именованного канала, «*int mode*» уровень доступа к каналу, который может быть логическим «ИЛИ» следующих значений:

- 0400 – разрешено чтение для пользователя, создавшего *FIFO*;
- 0200 – разрешена запись для пользователя, создавшего *FIFO*;
- 0040 – разрешено чтение для группы пользователя, создавшего *FIFO*;
- 0020 – разрешена запись для группы пользователя, создавшего *FIFO*;
- 0004 – разрешено чтение для всех остальных пользователей;
- 0002 – разрешена запись для всех остальных пользователей.

Системный вызов «*mkfifo*» возвращает значение типа «*int*», которое равно 0 в случае успеха. В случае неудачи возвращаемое значение равно -1, а код ошибки записывается в глобальную переменную «*errno*».

Удаление именованного канала можно осуществить с помощью системного вызова «*unlink*»:

```
#include <unistd.h>

int unlink(const char *pathname);
```

Входной аргумент «*const char *pathname*» является указателем на строку, содержащую путь и имя именованного канала. В случае успешного удаления «*unlink*» возвращает «0», в случае неуспешного «-1», а в глобальную переменную «*errno*» записывается код ошибки.

Дальнейшая работа с именованным каналом происходит так же, как и с остальными файлами – с помощью системных вызовов «*open*», «*read*», «*write*» и «*close*» или соответствующих библиотечных вызовов.

Для того, чтобы открыть именованный канал без блокировки на чтение у системного вызова «open» есть атрибут «*O_NONBLOCK*»:

```
int fd = open("namedpipe", O_RDONLY | O_NONBLOCK);
```

Для того, чтобы открыть именованный канал без блокировки на запись возможен вызов «open» с атрибутом *O_RDWR*:

```
int fd = open("namedpipe", O_RDWR);
```

5.3.2. Разделяемая память

Механизмы неименованных и именованных каналов крайне удобны в использовании, однако они имеют ряд недостатков.

- Для передачи данных от одного процесса к другому ОС осуществляет, как минимум, два копирования данных: первый раз из адресного пространства передающего процесса в буфер, и второй раз из буфера в адресное пространство принимающего процесса.
- Процессы, обменивающиеся информацией, должны одновременно существовать в ОС. Если хотя бы один из процессов непредвиденно завершится, то все данные будут потеряны.
- В именованных канала ОС не анализирует записываемый поток данных. Если в один именованный канал записывают данные несколько процессов-источников, то процесс-потребитель не сможет определить каким именно

процессом какие данные были записаны без специально добавляемых процессами меток.

Все эти факторы могут быть критичны или некритичны в зависимости от конкретных функций прикладных программ. Однако стоит упомянуть, что в ОС *Linux* есть механизм *IPC*, лишенный всех вышеперечисленных недостатков. Он представляет собой создание областей разделяемой памяти между процессами.

Разделяемая память создается с помощью системного вызова «*shmget*» и позволяет одному процессу напрямую обращаться к области памяти другого процесса и наоборот. В дальнейшем применение механизма разделяемой памяти в рамках данного лабораторного практикума производиться не будет. Самостоятельно более подробно можно ознакомиться с ним в [5].

5.4. Вспомогательные вызовы

При разработке приложений, порождающих процессы или участвующих в межпроцессном взаимодействии, часто применяются различные вспомогательные системные и библиотечные вызовы.

5.4.1. Системные вызов *getpid*

Системный вызов «*getpid*» может использоваться в процессе для того, чтобы узнать свой текущий *PID*:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
```

Он не имеет входных аргументов, а выходное значение «*pid_t*» является текущим *PID* процесса.

5.4.2. Системный вызов *getppid*

Системный вызов «*getppid*» может использоваться в процессе для того, чтобы узнать *PID* родительского процесса:

```
#include <sys/types.h>
#include <unistd.h>

pid_t getppid(void);
```

Он не имеет входных аргументов, а выходное значение «*pid_t*» соответствует *PID* родительского процесса (*PPID*).

5.4.3. Библиотечный вызов *sleep*

Библиотечный вызов «*sleep*» применяется для того, чтобы перевести процесс в режим ожидания (состояние «*S*») на указанное количество секунд:

```
#include <unistd.h>

unsigned int sleep(unsigned int seconds);
```

Возвращаемое значение «*unsigned int*» равно 0, если время перевода в режим ожидания истекло, или оставшееся количество секунд, если режим ожидания был прерван поступившим сигналом.

6. Упражнения

6.1. Начало работы

6.1.1. В соответствии с п. 1 – 2 из Упражнений предыдущей лабораторной (п. 1.4) работы подключитесь к командной оболочке *Raspberry Pi*.

6.1.2. Перейдите в каталог с вашими проектами, создайте в нем каталог «*lab3*» для данной лабораторной работы и перейдите в него:

```
cd IVT31_Ivanov_Ivan
mkdir lab3
cd lab3
```

6.2. Изучение процессов

6.2.1. Введите команду

```
ps
```

на экране отобразятся процессы, порожденные командной оболочкой: «*bash*» для обработки команд и «*ps*», который был вызван. В левой колонке будет отображен *PID* каждого процесса.

6.2.2. Введите команду

```
ps -aux
```

на экране отобразятся все процессы, запущенны в ОС в расширенном формате: указан пользователь, который запустил процесс, *PID* процесса, его состояние, команда его вызова и т. д.

6.2.3. Введите команду

```
ps -faxu
```

на экране отобразятся запущенные в системе процессы с указанием родительских и дочерних. Найдите дерево процессов, вызвавших используемый терминал. В корне дерева процесс *"/usr/sbin/sshd*" - программа «*SSH Daemon*», через которое произведено подключение.

6.2.4. Введите команду

```
top
```

на экране будут показаны запущенные процессы и используемые системные ресурсы. Информация будет обновляться в режиме реального времени.

6.3. Запуск и завершение процессов в командной оболочке:

6.3.1. Запустите скрипт

```
while [ true ]; do sleep 1;  done
```

который бесконечное количество раз ничего не делает раз в одну секунду.

Скрипт запущен на переднем плане и захватит командную оболочку, ввести другие команды будет невозможно.

6.3.2. Завершите выполнение процесса с помощью сочетания клавиш "*Ctrl+C*" (процессу будет отправлен сигнал *SIGINT*).

6.3.3. Запустите данный скрипт в фоновом режиме

```
while [ true ]; do sleep 1;  done &
```

на экране появятся его номер «jobspec» фоновых процессов [1] и значение *PID*.

6.3.4. Посмотрите запущенные фоновые процессы пользователя *pi* с помощью команды

```
jobs -l
```

6.3.5. Добавьте еще 2 фоновых процесса

```
while [ true ]; do sleep 1;  done &
while [ true ]; do sleep 1;  done &
```

6.3.6. Посмотрите фоновые процессы

```
jobs -l
```

6.3.7. Перевидите экземпляр фонового процесса [2] на передний план

```
fg %2
```

После перевода процесса на передний план командная строка стала недоступна для ввода новых команд. Завершите выполнение процесса с помощью сочетания клавиш «*Ctrl+C*».

6.3.8. Завершите выполнение остальных процессов в фоновом режиме с помощью утилиты *kill*:

```
kill %1
kill -9 %3
```

На экране появится сообщение для каждого процесса о том, что он был завершен. Для процесса [1] сообщение будет "*Terminated*" (отправлен сигнал *SIGTERM*), а для процесса [3] сообщение будет "*Killed*" (отправлен сигнал *SIGKILL*)

6.4. Межпроцессное взаимодействие в командной оболочке

6.4.1. Для примера применения неименованных каналов вычислите какое количество процессов в ОС запущено и находится в состоянии сна («*S*»)

6.4.1.1. Введите команду

```
ls /proc/*/stat
```

она выведет на экран все поддиректории в директории «*/proc*», которые содержат файл «*stat*», в котором указано состояние процесса.

6.4.1.2. Чтобы оставить только файлы «*stat*» в директориях, соответствующим процессам (именуются цифрами *PID*), передайте вывод «*ls*» команде «*grep*»:

```
ls /proc/*/stat | grep -e "/proc/[[:digit:]]*/stat"
```

В команде *grep* с помощью флага «*-e*» используются регулярные выражения для поиска директорий с числовым названием.

6.4.1.3. Преобразуйте полученный список файлов в аргументы с помощью «*xarg*» и прочитайте каждый файл с помощью «*cat*»:

```
ls /proc/*/stat | grep -e "/proc/[[:digit:]]*/stat" | xargs cat 2> /dev/null
```

Поскольку несколько найденных процессов соответствуют вызываемым в команде и завершаются раньше, чем будут прочитаны, «*cat*» выдаст в *stderr* сообщение о невозможности их прочитать. Чтобы убрать сообщения *stderr* «*cat*» перенаправляется в «*/dev/null*».

6.4.1.4. С помощью утилиты «*grep*» отфильтруйте содержимое каждого файла, оставив только статус «*S*»:

```
ls /proc/*/stat | grep -e "/proc/[[:digit:]]*/stat" | xargs cat 2> /dev/null | grep -o " S "
```

6.4.1.5. С помощью утилиты «*wc*» посчитайте количество строк:

```
ls /proc/*/stat | grep -e "/proc/[[:digit:]]*/stat" | xargs cat 2> /dev/null | grep -o " S " | wc -l
```

Выведенное на экран число равняется количеству запущенных процессов в системе, находящихся в состоянии сна.

6.4.2. В качестве примера использования именованных каналов разработайте *bash*-скрипты для получения значений пинов *GPIO* в режиме реального времени.

6.4.2.1. Создайте скрипт «*gpiosrc.sh*», назначьте ему права на исполнение и откройте в текстовом редакторе:

```
touch gpiosrc.sh
chmod +x gpiosrc.sh
nano gpiosrc.sh
```

6.4.2.2. Добавьте в созданный скрипт следующий код:

```
#!/bin/bash

# get arguments:
pin=$1

# export pin to userspace:
echo "$pin" > /sys/class/gpio/export 2> /dev/null

# check files of pin:
dirfile=/sys/class/gpio/gpio$pin/direction
while [ ! -f  $dirfile ] ; do sleep 1 ; done
sleep 1

# set direction as "input":
echo "in" > $dirfile


# get pin state and write to fifo:
while [ true ] ; do
        pinstate=`(cat /sys/class/gpio/gpio$pin/value)`
        echo "PIN$pin=$pinstate"
        sleep 1
done
```

Данный скрипт настраивает на вход пин *GPIO*, номер которого передается через аргумент, после чего в бесконечном цикле раз в секунду считывает логический уровень пина и отправляет в *stdout*.

6.4.2.3. Запустите скрипт и убедитесь, что раз в секунду на экране отображается логический уровень пина:

```
./gpiosrc.sh 4
```

6.4.2.4. Создайте скрипт «*gpiodst.sh*» для приема и вывода значений нескольких пинов, назначьте ему права на исполнение и откройте в текстовом редакторе:

```
touch gpiodst.sh
chmod +x gpiosrc.sh
nano gpiodst.sh
```

6.4.2.5. Добавьте в созданный скрипт следующий код:

```
#!/bin/bash

# init pins array:
for (( i=0; i < 32; i=i+1 )) ; do
        pins[$i]="x"
done

# get pin state:
while read pinmsg ; do

        # split:
        IFS='=' read -ra pinm <<< "$pinmsg"
        pinname=${pinm[0]};
        pinvalue=${pinm[1]};
        pinnum=${pinname:3};


        # update value for pin:
        pins[pinnum]=$pinvalue


        # out:
        clear
        for pin in ${pins[*]} ; do
                echo -n "$pin"
        done
        echo ""

done
```

Вначале инициализируется массив «*pins*» значением «*x*» для каждого пина, после чего в цикле ожидается появление в «*stdin*» строки «*pinmsg*», она разбирается в соответствии с форматом строки «*gpiosrc.sh*» и обновляет значение соответствующего пина, очищается терминал и выводятся новые значения всех пинов.

6.4.2.6. Создайте именованный канал для передачи данных:

```
mkfifo gpiodata
```

6.4.2.7. Запустите несколько процессов для получения состояний пинов в фоновом режиме и процесс для приема на переднем плане с перенаправлением данных через именованный канал:

```
./gpiosrc.sh 4 > gpiodata &
./gpiosrc.sh 17 > gpiodata &
./gpiosrc.sh 18 > gpiodata &
./gpiodst.sh < gpiodata
```

Убедитесь, что значения пинов обновляются.

6.4.2.8. Завершите процесс приема данных с пинов с помощью «*Ctrl+C*». Остальные процессы так же завершатся с сообщением «*Broken pipe*» из-за того, что канал закроется со стороны получателя данных.

6.4.2.9. Удалите файл именованного канала

```
rm gpiostate
```

6.4.2.10. Добавьте скрипты в *git*

```
git add gpiosrc.sh gpiodst.sh
git commit -m “Example of mkfifo for pins viewing was added”
```

6.5. Работа с процессами в приложениях

6.5.1. Для сборки программы скопируйте «*Makefile*» из Лабораторной работы №2

```
cp ../lab2/Makefile Makefile
```

6.5.2. Создайте новый файл для исходного кода и откройте его в текстовом редакторе

```
touch main.c
nano main.c
```

6.5.3. Добавьте в файл следующий код:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main() {

    printf("\n");

    pid_t pid = fork();

    if (pid != 0) {                // parent 1

        printf("PID: %d parent 1          (popen)\n", getpid());

        // call popen:
        FILE *fp = popen("/bin/bash -c \"while [ true ]; do sleep 1 ;
done\"", "r");
        pclose(fp);

        int status;
        wait(&status);

    } else {                                // child 1

        printf("PID: %d    child 1\n", getpid());

        pid = fork();

        if (pid != 0) {                // parent 2

            printf("PID: %d       parent 2    (system)\n", getpid());

            // call system:
            system("while [ true ]; do sleep 2 ; done");

            int status;
            wait(&status);

        } else {                                    // child 2

            printf("PID: %d          child 2 (exec)\n", getpid());

            // call exec:
            execl("/bin/bash", "/bin/bash", "-c", "/bin/bash -c \"while
[ true ]; do sleep 3 ; done\"", NULL);

        }

    }
```

```
    return 0;
}
```

Данный код с помощью системного вызова «*fork*» разделяет процесс на родительский (*parent 1*) и дочерний (*child 1*). В родительском процессе с помощью библиотечного вызова «*popen*» вызывается командная оболочка с бесконечным циклом.

Дочерний процесс с помощью «*fork*» разделяется на родительский (*parent 2*) и дочерний (*child 2*) процессы. В «*parent 2*» бесконечный цикл вызывается с помощью «system». В «*child 2*» бесконечный цикл вызывается с помощью «*exec*».

6.5.4. Скомпилируйте и запустите программу в фоновом режиме:

```
make
./program &
```

6.5.5. После запуска на экране появятся сообщения с соответствием между *PID* процесса и его уровнем в иерархии. Вызовете команду

```
ps -fu
```

Выведенное дерево отражает порядок разделения процессов на родительские и дочерние. Сверьте *PID*, выведенные на экран, и *PID*, отображаемые в дереве процессов. Обратите внимание, что процессы «*parent 1*» и «*child 1*»/«*parent 2*» являются программой «*./program*», которые породили «*sh*», а процесс «*child 2*» был замещен вызовом «*exec*».

6.5.6. Переведите процессы из фонового режима на передний план:

```
fg %1
```

6.5.7. Завершите процессы с помощью нажатия «*Ctrl+C*»

6.5.8. Добавьте исходные коды программы и *Makefile* в *git*:

```
git add main.c Makefile
git commit -m “Example for processes calling was added”
```

7. Индивидуальные задания

7.1. Общие требования к выполнению

7.1.1. Задания из п. 7.2 – 7.4 должны быть выполнены последовательно.

7.1.2. Каждое выполненное задание должно быть сохранено с использованием системы контроля версий git. Допускается сохранение промежуточных вариантов.

7.1.3. Коммиты в git должны иметь содержательное описание произведенных изменений.

7.1.4. Для каждого выполненного задания должен быть создан файл «README.MD», располагающийся в одном каталоге с исходными кодами и сохраненный с помощью git. Данный файл должен описывать поведение работы программы и показывать примеры запуска.

7.1.5. Разработанные программы должны иметь режим справки по вводимому ключу «-h» в качестве первого аргумента.

7.1.6. Конкретное задание основывается на двоичном представлении номера обучающегося в общем списке в восьмибитном формате *N*.

7.1.7. Для ознакомления принципов работы с GPIO изучите Приложение 1.

7.1.8. Продемонстрируйте преподавателю выполненное задание перед тем, как приступить к следующему.

7.2. Задание №1

7.2.1. Разработайте программу, которая порождает отдельный процесс для переключения состояния светодиода из активного в неактивный и наоборот с заданной периодичностью (мигание). Период мигания задается через входной аргумент при вызове программы.

7.2.2. $N_0 = 0$: дочерний процесс порождается с помощью системного вызова «fork»

7.2.3. $N_0 = 1$: дочерний процесс порождается с помощью библиотечного вызова «popen»

7.3. Задание №2

7.3.1. Усовершенствуйте программу, добавив в родительский процесс прием аргумента через «stdin» для изменения частоты мигания.

7.4. Задание №3

7.4.1. Усовершенствуйте программу, добавив в нее обработчик сигнала SIGINT, который всегда устанавливает светодиод в неактивное состояние перед завершением процесса.

8. Контрольные вопросы

1. Что такое процесс?
2. Для чего предназначен механизм IPC?
3. Как работает механизм сигналов?
4. В чем отличие между неименованных и именованных каналов?
5. Какие действия осуществляет ОС при получении вызова fork?
6. Чем вызов exec отличается от вызова popen?

9. Список литературы

1. Таненбаум Э. Б.Х. Современные операционные системы. СПб.: Питер, 2015. 1120 pp.
2. Memory management unit (MMU) [Электронный ресурс] URL: https://en.wikipedia.org/wiki/Memory_management_unit
3. OpenNET: Понятие о сигналах [Электронный ресурс] URL: https://www.opennet.ru/docs/RUS/linux_parallel/node10.html
4. Embedded Systems - Interrupts [Электронный ресурс] URL: https://www.tutorialspoint.com/embedded_systems/es_interrupts.htm
5. Проект OpenNET: MAN shmget [Электронный ресурс] URL: https://www.opennet.ru/man.shtml?topic=shmget&category=2&russian=0