Государственное бюджетное образовательное учреждение среднего профессионального образования Новосибирской области

Новосибирский химико- 630102 г. Новосибирск, ул. Садовая, 26,

технологический колледж. Тел./факс: (383) 266-00-44, тел.: (383) 266-00-54,

nhtk@mail.ru, http://nhtk-edu.ru

им. Д.И. Менделеева

**МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ**

**ПРАКТИЧЕСКОЙ РАБОТЫ**

Разработчик:

Соломенный Ю.Ю.

Новосибирск – 2023

**Практическая работа №4 «Процессы»**

**Теоретический материал**

Основными активными сущностями в системе Linux являются процессы. Каждый процесс выполняет одну программу и изначально получает один поток управления. Иначе говоря, у процесса есть один счетчик команд, который отслеживает следующую исполняемую команду. Linux позволяет процессу создавать дополнительные потоки (после того, как он начинает выполнение).

Linux представляет собой многозадачную систему, так что несколько независимых процессов могут работать одновременно. Более того, у каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать cотни и даже тысячи процессов. Фактически на большинстве однопользовательских рабочих станций (даже когда пользователь куда-либо отлучился) работают десятки фоновых процессов, называемых демонами (daemons). Они запускаются при загрузке системы из сценария оболочки.

Типичным демоном является *cron*. Он просыпается раз в минуту, проверяя, не нужно ли ему что-то сделать. Если у него есть работа, то он ее выполняет, а затем отправляется спать дальше (до следующей проверки).

Этот демон позволяет планировать в системе Linux активность на минуты, часы, дни и даже месяцы вперед. Например, представьте, что пользователю назначено явиться во военкомат в 3 часа дня в следующий вторник. Он может создать запись в базе данных демона cron, чтобы тот просигналил ему, скажем, в 14:30. Когда наступает назначенный день и время, демон *cron* видит, что у него есть работа, и запускает в назначенное время программу звукового сигнала (в виде нового процесса).

Демон *cron* также используется для периодического запуска задач, например, ежедневного резервного копирования диска в 4 часа ночи или напоминания забывчивым пользователям каждый год за неделю до 31 декабря купить подарки для празднования нового года. Другие демоны управляют входящей и исходящей электронной почтой, очередями принтера, проверяют, достаточно ли еще осталось свободных страниц памяти и т.д. Демоны реализуются в системе Linux довольно просто, так как каждый из них представляет собой отдельный процесс, независимый от всех остальных процессов.

Процессы создаются в операционной системе Linux очень просто. Системный вызов *fork* создает точную копию исходного процесса, называемого родительским процессом (parent process). Новый процесс называется дочерним процессом (child process). У родительского и у дочернего процессов есть свои собственные (приватные) образы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, то эти изменения остаются невидимыми для дочернего процесса (и наоборот).

Открытые файлы используются родительским и дочерним процессами совместно. Это значит, что если какой-либо файл был открыт в родительском процессе до выполнения системного вызова *fork*, то он останется открытым в обоих процессах и в дальнейшем. Изменения, произведенные с этим файлом любым из процессов, будут видны другому. Такое поведение является единственно разумным, так как эти изменения будут видны также и любому другому процессу, который тоже откроет этот файл.

Тот факт, что образы памяти, переменные, регистры и все остальное и у родительского процесса, и у дочернего идентичны, приводит к небольшому затруднению: как процессам узнать, который из них должен исполнять родительский код, а который дочерний? Секрет в том, что системный вызов fork возвращает дочернему процессу число 0, а родительскому — отличный от нуля PID (Process IDentifier — идентификатор процесса) дочернего процесса.

Оба процесса обычно проверяют возвращаемое значение и действуют соответственно:  
  
pid = fork( ); /\* если fork завершился успешно, pid > 0 в родительском процессе \*/  
if (pid < 0) {  
handle\_error(); /\* fork потерпел неудачу (например, память или какая-  
либо таблица переполнена) \*/  
} else if (pid > 0) {  
/\* здесь располагается родительский код \*/  
} else {  
/\* здесь располагается дочерний код \*/  
}

Если дочерний процесс желает узнать свой PID, то он может воспользоваться системным вызовом *getpid*. Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его родитель получает PID только что завершившегося дочернего процесса. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, то исходный процесс может создать целое дерево детей, внуков, правнуков и более дальних потомков.

В системе Linux процессы могут общаться друг с другом с помощью некой формы передачи сообщений. Можно создать канал между двумя процессами, в который один процесс сможет писать поток байтов, а другой процесс сможет его читать. Эти каналы иногда называют трубами (pipes). Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется.

При помощи каналов организуются конвейеры оболочки. Когда оболочка видит строку вроде:

*sort <f | head*

то она создает два процесса, *sort* и *head*, а также устанавливает между ними канал таким образом, что стандартный поток вывода программы sort соединяется со стандартным потоком ввода программы *head*. При этом все данные, которые пишет *sort*, попадают напрямую к head, для чего не требуется временного файла. Если канал переполняется, то система приостанавливает работу *sort* до тех пор, пока *head* не удалит из него хоть сколько-нибудь данных.

Процессы также могут общаться и другим способом — при помощи программных прерываний. Один процесс может послать другому так называемый сигнал (signal). Процессы могут сообщить системе, какие действия следует предпринимать, когда придет сигнал. Варианты такие: проигнорировать сигнал, перехватить его, позволить сигналу убить процесс (действие по умолчанию для большинства сигналов).

Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуру обработки сигналов. Когда сигнал прибывает, управление сразу же передается обработчику. Когда процедура обработки сигнала завершает свою работу, то управление снова передается в то место, в котором оно находилось, когда пришел сигнал (это аналогично обработке аппаратных прерываний ввода-вывода).

Процесс может посылать сигналы только членам своей группы процессов (process group), состоящей из его прямого родителя (и других предков), братьев и сестер, а также детей (и прочих потомков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

Сигналы используются и для других целей. Например, если процесс выполняет вычисления с плавающей точкой и непреднамеренно делит на 0, то он получает сигнал SIGFPE (Floating-Point Exception SIGnal — сигнал исключения при выполнении операции с плавающей точкой).

[Создание процессов с помощью вызова fork(). (opennet.ru)](https://www.opennet.ru/docs/RUS/linux_parallel/node7.html)

[Работа с процессами в Linux. Команды для управления системными процессами в системе Линукс (dmosk.ru)](https://www.dmosk.ru/miniinstruktions.php?mini=processes-linux&ysclid=lmhg1q14q56929685)

**Порядок выполнения практической работы**

1. Практическая работа выполняется в ***отдельном*** создаваемом студентом ***документе***!
2. Титульный лист с указанием наименования организации, ФИО студента и номер группы
3. Выполнить задания, в отчете написать номер задания, добавить скриншот, описать результат.
4. Ответить на контрольные вопросы
5. Вывод

**Практическое задание**

1. Создайте скрипт (программу) на языке C, который будет выводить PID текущего процесса и PID родительского процесса.
2. Используя команду sleep, добавьте задержку в скрипт.
3. Создайте новый процесс внутри текущего скрипта с помощью оператора fork.

4. Внутри нового процесса, выведите его PID и PID родительского процесса.

5. Используя команду ps, проверьте иерархию процессов, чтобы убедиться, что новый процесс является дочерним для родительского процесса.

6. В основном процессе, дождитесь завершения нового процесса с помощью оператора wait.

7. Выведите сообщение о завершении нового процесса.

8. Используя команду pstree или ps, проверьте иерархию процессов после завершения нового процесса.

9. Прокомментируйте в письменном виде результаты выполнения скрипта.

**Примечания:**

- Проверьте результаты выполнения каждого этапа с помощью вывода в консоль.

- Обратите внимание на правильный порядок использования системных вызовов fork() и wait().

- Обратите внимание на отслеживание PID процессов во время выполнения задания.