ЛАБОРАТОРНАЯ РАБОТА №1

Тема: Базовая работа с процессами в ОС Linux. Сигналы

Цель работы: Изучить основы работы с процессами в ОС Linux, а также освоить сигналы как универсальное средство межпроцессного взаимодействия.

Цель работы обуславливает постановку и решение следующих **задач**:

1. Изучить основные понятия о процессах в ОС Linux, относящиеся к их созданию и управлению.
2. Ознакомиться с базовыми системными вызовами POSIX для работы с процессами: fork(), wait(), waitpid().
3. Изучить назначение и возможности сигналов в Linux, их классификацию (системные, пользовательские) и обработку.
4. Освоить установку обработчиков сигналов с помощью signal() и sigaction(), а также способы отправки сигналов с помощью kill() и sigqueue().
5. Познакомиться с соответствующими консольными инструментами для мониторинга и управления процессами и сигналами.
5.1 Изучение команд ps, top, htop — просмотр списка процессов, их состояний и ресурсов.
5.2 Изучение команд jobs, bg, fg — управление фоновыми и приостановленными задачами в оболочке.
5.3 Изучение команд kill, pkill, killall — отправка сигналов процессам по PID или имени.
5.4 Изучение команды ulimit — просмотр и установка ресурсных ограничений (например, число процессов, размер стека).
5.5 Изучение команды strace — отслеживание системных вызовов процесса, включая работу с сигналами.
6. Выполнить индивидуальное задание для закрепления знаний на практике, подготовив соответствующую программу на языке C (номер задания соответствует номеру студента по журналу; если этот номер больше, чем максимальное число заданий, тогда вариант задания вычисляется по формуле: номер по журналу % максимальный номер задания, где % — остаток от деления; если остаток равен 0, студенту назначается задание с максимальным номером).
7. Подготовить отчёт по работе, который должен включать: краткое описание всех использованных системных вызовов и команд; текст программы; протоколы; скриншоты с демонстрацией поведения ОС Linux; выводы.

Краткие теоретические сведения

Процесс в ОС Linux — это единица планирования и управления в ОС, имеющая собственное адресное пространство, ресурсы и состояние.

Вызов fork() создаёт новый процесс-потомок, который продолжает выполнение программы с того же места, где был вызван fork(). После этого вызова родитель и потомок продолжают выполняться независимо, при этом их данные и стек находятся в разных адресных пространствах, поэтому изменения в памяти одного процесса не влияют на память другого. У процесса-потомка после fork() совпадают код программы, контекст выполнения, открытые файловые дескрипторы и окружение, но отличаются такие характеристики, как PID, PPID, время выполнения, а также содержимое некоторых счётчиков.

Вызов wait() приостанавливает выполнение родительского процесса до тех пор, пока один из его дочерних процессов не завершится, после чего возвращает его PID и статус завершения.

Вызов waitpid() позволяет родительскому процессу ожидать завершения конкретного дочернего процесса (по его PID) или группы процессов, при этом можно задать режимы работы (например, неблокирующий режим ожидания, когда при использовании флага WNOHANG функция немедленно возвращает управление родительскому процессу, если ни один из ожидаемых процессов ещё не завершился).

Сигналы в Linux являются одним из наиболее простых и эффективных средств межпроцессного взаимодействия. Они представляют собой асинхронные уведомления, которые ОС или процессы могут посылать друг другу для информирования о наступлении определённых событий. Парадигма сигналов строится на механизме передачи коротких сообщений-идентификаторов (номеров сигналов) между процессами, что позволяет:

1. Управлять жизненным циклом процессов (останов, продолжение, завершение).

2. Синхронизировать их работу (например, «пинг-понг» обмен сообщениями).

3. Реализовывать простые протоколы взаимодействия без использования сложных каналов или сокетов.

Команда kill -l выводит список всех доступных сигналов в Linux с их номерами и именами.

Системные сигналы автоматически посылаются ядром для уведомления процесса о событии (например, SIGSEGV), а пользовательские сигналы (SIGUSR1, SIGUSR2) могут использоваться программистом для организации собственного взаимодействия между процессами.

Сигналы реального времени — это особый диапазон POSIX-сигналов (от SIGRTMIN до SIGRTMAX), которые поддерживают очередь доставки и могут нести вместе с собой дополнительное значение (sigval), что делает их более надёжным и информативным средством межпроцессного взаимодействия.

Для того, чтобы процесс мог перехватывать сигналы и управлять своим поведением при их получении, обеспечивая корректную обработку событий и устойчивость работы программы используются различные инструменты.

Так вызов signal() позволяет связать определённый сигнал с пользовательской функцией-обработчиком, которая будет автоматически выполняться при поступлении этого сигнала в процесс.

Вызов sigaction() предоставляет более гибкий и надёжный способ установки обработчиков сигналов, позволяя задавать дополнительные флаги и маски блокируемых сигналов во время их обработки.

Функция kill() используется для отправки сигналов процессу или группе процессов по их PID, что позволяет завершать их работу или инициировать выполнение заданных обработчиков.

Функция sigqueue() используется для отправки процессу указанного сигнала вместе с дополнительным целочисленным значением, что позволяет организовывать более информативное взаимодействие между процессами.

Для того, чтобы программист, администратор или пользователь мог в реальном времени наблюдать состояние процессов, их ресурсы и сигналы, а также управлять их выполнением и завершением используются консольные инструменты.

Для интерактивного управления процессами из терминала используются следующие сочетания клавиш:

1. Нажатие Ctrl+C отправляет процессу сигнал SIGINT (2), который по умолчанию завершает его работу, но может быть перехвачен обработчиком.

2. Нажатие Ctrl+Z отправляет процессу сигнал SIGTSTP (20), который по умолчанию приостанавливает его выполнение и может быть перехвачен обработчиком.

3. Нажатие Ctrl+\ отправляет процессу сигнал SIGQUIT (3), который по умолчанию завершает его работу и формирует core dump, но может быть перехвачен обработчиком. В Linux

возможно разрешить создание дампа памяти, установив лимит размера core-файла командой ulimit -c unlimited (для текущей сессии).

Команда ps отображает список активных процессов в системе и позволяет получить информацию об их идентификаторах, состоянии, использовании ресурсов и связанной командной строке.

Примеры:

ps -ef — показать полный список всех процессов в системе с указанием PID, PPID, пользователя, времени запуска и команды.

ps -p 1234 -o pid,ppid,cmd,stat — вывести для процесса 1234 его PID, PID родителя, команду запуска и текущий статус.

Команда pgrep ищет процессы по имени или шаблону и выводит их PID, например pgrep chrome покажет идентификаторы всех запущенных процессов браузера Google Chrome.

Команда cat /proc/<PID>/status отображает подробную информацию о процессе с данным PID, включая состояние, права, идентификаторы, ресурсы и лимиты.

Команда cat /proc/<PID>/maps показывает карту отображённых в память областей процесса, включая адреса, права доступа, размер и соответствующие файлы или библиотеки. Отображённые в память области — это фрагменты файлов, библиотек или анонимной памяти, которые ядро связало с адресным пространством процесса для его работы.

Команда top интерактивно отображает в реальном времени список процессов с их загрузкой процессора, памяти и другими параметрами, позволяя отслеживать и управлять ресурсами системы.

Команда jobs показывает список задач, запущенных из текущей оболочки, с их номерами и состояниями.

Команда bg переводит приостановленную задачу в выполнение в фоновом режиме.

Команда fg возвращает задачу из фонового режима для выполнение на передний план.

Команда kill отправляет сигнал процессу или группе процессов по их PID.

Например:

kill -SIGTERM 1234 — послать процессу с PID 1234 запрос на завершение (по умолчанию).

kill -SIGKILL 1234 — немедленно завершить процесс 1234 без возможности обработки сигнала.

kill -SIGSTOP 1234 — приостановить выполнение процесса 1234.

kill -SIGCONT 1234 — возобновить выполнение ранее остановленного процесса 1234.

kill -SIGHUP 1234 — послать сигнал «разрыв соединения» (часто используется для перезапуска демонов).

kill -SIGUSR1 1234 — послать пользовательский сигнал 1 для обработки внутри программы.

Команда pkill отправляет сигнал процессам, отбираемым по имени или другим критериям.

Команда killall отправляет сигнал всем процессам с указанным именем.

Команда ulimit используется для просмотра и задания ограничений на ресурсы, доступные процессам текущей оболочки, например на количество создаваемых процессов или максимальный размер стека.

Вот несколько кратких примеров с ulimit:

ulimit -a — показать все текущие ограничения.

ulimit -s 8192 — задать размер стека в 8192 КБ.

ulimit -u 200 — ограничить число одновременно создаваемых процессов до 200.

Команда strace используется для запуска или подключения к процессу с целью отслеживания выполняемых им системных вызовов и сигналов, что помогает анализировать его поведение и отладку.

Пример:

strace ls — показать все системные вызовы, выполняемые при запуске команды ls.

strace -e open,close cat file.txt — отследить только вызовы open и close при чтении файла.

strace -p 1234 — подключиться к уже работающему процессу с PID 1234 и наблюдать его системные вызовы.

strace -e signal ./prog — отследить только работу программы prog с сигналами.

strace -o log.txt ./a.out — сохранить трассировку выполнения программы a.out в файл log.txt.

Команда taskset используется для задания или просмотра привязки процесса к определённым CPU-ядрам, что позволяет контролировать распределение нагрузки по процессорам.

taskset -cp 0 1234 — задать процессу с PID 1234 привязку к первому ядру процессора (ядро с номером 0).

Команда nohup запускает процесс так, чтобы он игнорировал сигнал SIGHUP и продолжал работать в фоне даже после выхода пользователя из системы.

Для порождения процесса-демона можно запустить процесс с помощью nohup ./myprogram & disown, чтобы он продолжал работать в фоне независимо от терминала. В этой команде nohup защищает процесс от завершения при выходе из терминала, ./myprogram запускает программу, & отправляет её в фон, a disown отвязывает от текущей оболочки.

Индивидуальные задания

Для всех заданий предусмотреть корректное завершение работы (gracefully) с освобождением всех занятых ресурсов. Выполнить текущий мониторинг процессов с использованием команд ps, top. Для своего задания провести эксперимент, вначале разрешив планировщику ОС распределять процессы самостоятельно по ядрам. Затем вручную задать всем процессам привязку к одному ядру (системный вызов sched_setaffinity или утилита taskset). Изменились ли полученные результаты? Почему?

0. Реализовать программу, в которой родительский процесс порождает два дочерних процесса с помощью fork(). Каждый дочерний процесс в бесконечном цикле раз в секунду печатает сообщение со своим порядковым номером и PID. Родительский процесс в тоже время в собственном бесконечном цикле выводит сообщение о своей работе. Корректное завершение программы осуществляется по Ctrl+C: родитель перехватывает сигнал SIGINT, отправляет всем дочерним процессам сигнал SIGTERM, ожидает их завершения с помощью wait() и после этого завершает свою работу.

1. Породить цепочку из N процессов «родитель ➡ потомок 1 ➡ потомок 2 ➡ … ➡ потомок N-1», образовав линейную иерархию. То есть «родитель» порождает «потомка 1», «потомок 1» порождает «потомка 2» и т.д. В бесконечном цикле следует последовательно по цепочке (0-й, 1-й, … N-1-й) печатать PID этих процессов (причем, каждый процесс должен печатать свой PID). Использовать сигналы для синхронизации и завершения работы. Инициализацию цепочки выполняет родитель. Корректное завершение по Ctrl+C (обрабатывается родителем): родитель посылает всей группе SIGTERM.

2. Породить N процессов-демонов (фоновых процессов) с разными статическими приоритетами планировщика. В Linux приоритет задаётся через значение nice (от -20 до 19, чем меньше значение, тем выше приоритет). Процессы-демоны должны периодически (через каждые S секунд) записывать информацию (PID и текущее время) в специальный log-файл (например, /var/log/my_daemon.log). Проанализировать log-файл, выяснить в каких пределах можно эффективно изменять приоритеты процессов, как приоритет влияет на частотность записей в log-файле.

3. Породить кольцо из N процессов «родитель 0 ➡ потомок 1 ➡ потомок 2 ➡ … ➡ потомок N-1 ➡ родитель 0». Осуществить передачу по кольцу целого числа S, которое печатается в консоль текущим процессом вместе со своим PID и PPID сразу после получения S. Перед передачей S далее по кольцу значение S увеличивается текущим процессом на единицу. Для передачи S можно воспользоваться, например, библиотечной функцией стандарта POSIX sigqueue.

4. Задание «Фабрика и дракон». Фабрика (процесс 1) каждые T1 миллисекунд производит продукт (случайное число от 1 до 9) и отправляет его в процесс-склад (процесс 2), где значения суммируются. Если эта сумма больше M процесс-склад заполнен, фабрика засыпает и ждёт, пока освободится место. Дракон (процесс 3) каждые T2 миллисекунд берёт один продукт со склада (вычитает случайное число от 1 до E) и «съедает» его (печатает в консоль). Если дракону нечего есть, то дракон засыпает, пока не появится новый продукт на складе. Подобрать параметры T1, T2, M, E так, чтобы обеспечить устойчивую работу фабрики по сбыту продукции.

5. Реализовать map-reduce вычисление количества простых чисел на отрезке [A, B], используя пул из N рабочих процессов. Процесс-мастер делит отрезок на блоки длиной S и раздаёт их через сигналы, а процессы-рабочие считают частичные суммы и возвращают результаты, мастер выполняет редукцию (суммирование) и печатает итог. Предусмотреть корректное завершение по Ctrl+C.

6. Реализовать процесс «watch-dog» (сторожевой таймер). Создать основной процесс и K дочерних процессов-исполнителей. Каждый исполнитель периодически (раз в T секунд) посылает родителю сигнал «я жив». Родительский процесс («watch-dog») отслеживает поступление таких сигналов от всех исполнителей. Если в течение интервала M от какого-либо процесса сигнал не поступил, «watch-dog» фиксирует сбой (печатает в консоль предупреждение с PID проблемного процесса) и предпринимает меры: завершает зависший процесс сигналом SIGKILL и порождает новый процесс-исполнитель взамен. Предусмотреть возможность корректного завершения всей системы по Ctrl+C. Смоделировать зависание процесса, например, посредством принудительной его остановки извне сигналом.

7. Смоделировать ситуацию «зомби-апокалипсиса», то есть ситуацию накопления зомби-процессов и их последующую «утилизацию». Родитель порождает пул из N дочерних процессов. Каждый дочерний процесс мгновенно завершается с кодом выхода exit(code) (например, code = getpid() % 100) и тем самым превращается в зомби до тех пор, пока родитель не соберёт их через wait/waitpid. Родитель должен собрать их пачкой и подсчитать сумму кодов выхода. Необходимо подобрать эмпирически максимально возможное N для данной конфигурации ОС за пределами которого возникает отказ в обслуживании ОС.

8. Построить полное бинарное дерево процессов глубины D, в котором любой лист по специальному сигналу может инициировать корректное завершение всех процессов дерева, причём оповещения (сигналы) допускается распространять только вдоль рёбер дерева между

смежными процессами (ребёнок ⟷ родитель). Пересылка между несмежными вершинами запрещена.

9. Реализовать «пинг-понг» сигналов между родительским процессом и дочерним процессом. Родитель первым генерирует целое число X и отправляет его дочернему процессу («пинг»). Дочерний процесс, получив сигнал, печатает на экран X с указанием своего PID и отсылает родителю новое целое число X' («понг»). Родитель, получив X', печатает его со своим PID, генерирует новое целое число X" и снова отсылает дочернему процессу. Таким образом процессы обмениваются сигналами в бесконечном цикле. Корректное завершение осуществляется по Ctrl+C: родитель перехватывает SIGINT, печатает сообщение об остановке и посылает дочернему процессу SIGTERM.

Пример кода на языке C (задание 0)

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

#define NUM 2
pid_t children[NUM] = { [0 ... NUM-1] = -1 };

static volatile sig_atomic_t stop_flag = 0;
//безопасный тип для сигналов

void gf_handler(int signo) {
     for (size_t ind = 0; ind < NUM; ind++)
          if (children[ind] > 0)
               kill(children[ind], SIGTERM);
     stop_flag = 1;
}

int main() {

     setvbuf(stdout, NULL, _IONBF, 0);
     signal(SIGINT, gf_handler);

     for (size_t ind = 0; ind < NUM && !stop_flag; ind++) {
          children[ind] = fork();
          if (children[ind] == 0) {
               signal(SIGINT, SIG_DFL);
               while (1) {
                    printf("[Child%zu %d] Я тоже работаю\n",
                              ind + 1,
                              (int)getpid());
                    sleep(1);
               }
          } else if (children[ind] < 0) {
               perror("Ошибка порождения процесса!");
               stop_flag = 1;
          }
     }

     while (!stop_flag) {
          printf("[Parent %d] Я работаю\n", (int)getpid());
```

```
            sleep(1);
        }

        while (wait(NULL) > 0);

        return 0;
}
```

Как работает программа (задание 0)

Программа создаёт два дочерних процесса с помощью fork() и организует одновременную работу родителя и процессов-детей. Каждый дочерний процесс сбрасывает унаследованный обработчик SIGINT и в бесконечном цикле каждую секунду печатает своё сообщение с номером и PID. Родительский процесс также в бесконечном цикле выводит строку о своей работе, пока не придёт сигнал прерывания. При нажатии Ctrl+C срабатывает обработчик gf_handler, который посылает всем живым дочерним процессам сигнал SIGTERM и устанавливает флаг остановки. После этого родитель выходит из основного цикла, дожидается завершения всех детей с помощью wait, и программа корректно завершается.

Контрольные вопросы

1. Что такое процесс в ОС Linux, какие основные характеристики (PID, PPID, состояние, приоритет) ему присущи?
2. Каким образом в Linux создаются новые процессы? В чем суть системного вызова fork()?
3. Как работают системные вызовы wait() и waitpid()? Чем они отличаются?
4. Что такое зомби-процессы и каким образом их можно «утилизировать» (два сценария)?
5. Что представляют собой сигналы в Linux и как они классифицируются (системные, пользовательские, реального времени)?
6. Какими средствами можно установить обработчик сигнала в программе на языке C (signal(), sigaction())?
7. Как можно отправить сигнал процессу программно (kill(), sigqueue()) и с помощью консольных утилит (kill, pkill, killall)?
8. Чем отличаются методы отправки сигналов kill() и sigqueue()? В каких случаях удобно использовать второй вариант?
9. Как можно организовать взаимодействие процессов с помощью сигналов (например, схема «пинг-понг»)?
10. Каким образом происходит корректное завершение процессов по сигналам (SIGINT, SIGTERM)? Чем отличается SIGKILL?
11. Какие консольные команды позволяют просматривать список процессов и их характеристики (ps, top, htop)?
12. Как управлять фоновыми и приостановленными задачами в оболочке (jobs, bg, fg)?
13. Для чего используется команда ulimit, какие ограничения ресурсов процессов можно просматривать и задавать?
14. Как команда strace помогает анализировать работу процессов и их взаимодействие с ОС (в том числе обработку сигналов)?
15. Что такое приоритеты процессов в Linux, как работает механизм nice и как он влияет на распределение процессорного времени?
16. В чем заключается назначение процесса-демона? Как он отличается от обычного процесса?
17. Как можно привязать процесс к конкретному ядру процессора (sched_setaffinity, taskset) и зачем это делать?

18. В чем заключается отличие сигналов от других механизмов межпроцессного взаимодействия (каналов, сокетов, очередей сообщений)?
19. Как правильно организовать корректное завершение группы процессов при нажатии Ctrl+C?
20. Почему сигналы можно рассматривать как событийную модель межпроцессного взаимодействия?