Лабораторные работы по курсу
Информационные технологии 1. Операционные системы

Лабораторная работа 2
Процессы в операционной системе

Москва, 2022

Оглавление

Теоретическая часть

1. Процессы в операционной системе

Одним из основных понятий в операционных системах является понятие **процесса** – *программа во время исполнения или объект, которому выделяются ресурсы вычислительной системы, такие как процессорное время, память и т.д*.

Каждому процессу выделено адресное пространство, в котором содержится код программы (*text section*), данные для ее работы (*data section*), свзяанные файлы, с которыми идет работа, ожидающие обработки сигналы и т.п. Помимо этого, каждый процесс имеет собственный уникальный номер – идентификатор (PID – process identifier). Этот номер присваивается операционной системой при старте процесса, при этом значение PID нового процесса будет больше, чем у ранее созданного процесса, до тех пор, пока не будет присвоен максимально возможный номер для типа данных PID, который хранится в файле */proc/sys/kernel/pid_max*. После этого ОС назначает новым процессам номера от 1, если данные значения были освобождены другими процессами. Самому первому процессу, отвечающему за планирование остальных процессов, присваивается 0 и данный PID не может быть передан другому процессу.

Любой процесс порождается другим процессом, таким образом, в ОС всегда функционирует дерево процессов, каждый из которых является либо родительским процессом, т.е. процессом, порождающим другие, либо дочерним (порождаемым), либо и тем и другим. Чтобы можно было отследить преемственность процессов, помимо уникальных идентификаторов PID у каждого порожденного процесса есть данные о родителе – идентификатор родительского процесса (PPID – *parent process identifier*). В случае, если родительский процесс завершается раньше дочернего, то значение PPID у последнего приравнивается к 1 – это процесс *init*, который функционирует все время работы операционной системы.

1.1. Работа с процессами из консоли

Для администрирования текущих процессов операционной системы из консоли можно воспользоваться утилитой *ps*, которая позволяет просматривать такую информацию о работе запущенных в системе процессах как: PID, PPID, занимаемое процессорное время, занимаемую память и т.д. Данную информацию утилита берет из каталога /proc, где в виде файлов хранится состояние ядра операционной системы и запущенных процессов.

Пример запуска:

```
$ ps -u
```

Запустим утилиту с наиболее интересными нам параметрами. Для этого выполним команду:

```
$ ps -ao user,pid,ppid,vsz,stat,cmd
```

Результат работы приведен на рисунке 1.

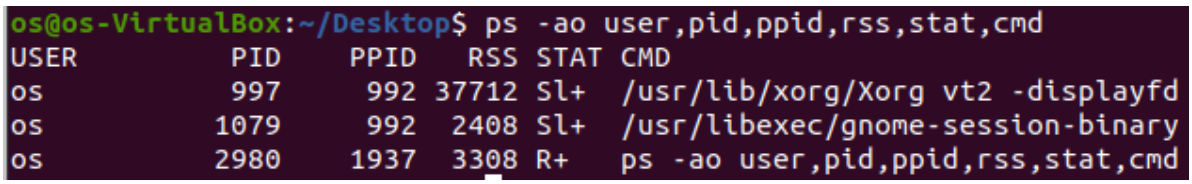

Рисунок 1 Результат работы ps

Поясним результат работы утилиты. В столбце USER указывается имя пользователя, запустившего данный процесс. В данном случае это пользователь os. PID – идентификатор запущенного процесса. PPID – идентификатор процесса-родителя. RSS — это размер резидентной памяти, указывающий сколько памяти выделено для этого процесса и находится в ОЗУ в момент вызова команды. Поле STAT обозначает текущее состояние процесса. Наиболее часто встречаются следующие значения:

R - процесс исполняется (или находится в очереди исполнения)

S - процесс ожидает завершения события

Z - процесс «зомби»

Также к данным значениям могут добавляться специальные символы. Например, «*l*» обозначает, что данное приложение является многопоточным, а «+» - процесс относится к группе переднего плана. Поле COMMAND содержит название запущенной команды.

Любой процесс может быть завершен с помощью утилиты kill, если известен его идентификатор – PID и пользователь обладает на это достаточными правами.

Аналогичную информацию возможно увидеть с помощью утилиты top.

```
$ top
```

1.2. Работа с процессами на языке программирования C

Порождение дочернего процесса родительским возможно осуществить с помощью системного вызова *fork()* и семейства функций *exec()*.

Процесс, созданный с помощью системного вызова *fork()* (от англ. «вилка»*)* является точной копией родительского процесса, за исключением таких параметров как PID и PPID.

Чтобы при выполнении кода различать родительский и дочерний процессы, функция *fork()* возвращает разные для них значения. Так, при успешном создании нового процесса в родительский процесс возвращается PID дочернего, а в дочерний процесс возвращается – 0. За счет проверки возвращаемого значения можно по-разному организовать дальнейшую работу родственных процессов.

```
pid = fork();
if(pid == -1){
    ...
    /* ошибка */
    ...
} else if (pid == 0){
    ...
    /* дочерний процесс */
    ...
} else {
    ...
    /* родительский процесс */
    ...
}
```

После окончания работы дочерний процесс должен отправить сигнал о своем завершении родительскому процессу. Для ожидания данного сигнала существует системный вызов *wait()*.

Рассмотрим следующий пример:

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

int main()
{
pid_t PID=fork();
if(PID==0)
{
        printf("Hello ");
}
else
{
         wait(0);
         printf("world!\n");
}
return 0;
}
```

В приведенном примере показана работа системного вызова *fork()*. После его вызова создается копия текущего процесса. Далее код начинает выполняться со следующей строчки уже в **двух** процессах. В зависимости от возвращенного значения fork, в процессах выполнится первое или второе условие ветвления. Дочерний процесс выведет “Hello!” и затем завершиться. В то же время родительский процесс остановится в системном вызове *wait(0)* и после получения кода возврата от дочернего процесса выведет строку world и завершится.

Приведенная в приложении анимация иллюстрирует данный пример.

В случае некорректной работы программ возможны две ситуации:

1. Дочерний процесс завершил свою работу, отправил сигнал об окончании, однако родительский процесс еще не выполнил вызов *wait()*, ожидающий данный сигнал. Таким образом, дочерний процесс уже «умер» (работа выполнена, память освобождена), но не «погребен» (информация об окончании работы не получена родителем, номер дочернего процесса все еще в списке существующих). Процесс в таком состоянии называется «зомби-процессом».
2. Родительский процесс завершил свою работу, но процесс потомок все еще выполняется. Такой процесс «без родителя» называется «процессом-сиротой». В данном случае, система назначает его родителем процесс *init*.

Вторым способом запуска процесса в Unix является вызов функции из семейства *exec*. Функция *exec*() (*execute*) загружает и запускает другую программу. Таким образом, новая программа полностью замещает текущий процесс. Новая программа начинает свое выполнение с функции *main*.

Если говорить об управлении процессами из программ, написанных на языке С, то для получения информации об идентификаторе исполняемого процесса используется системный вызов getpid(), а для получения значения идентификатора родительского процесса – getppid().

Прототипы системных вызовов getpid(), getppid(), fork() и соответствующие им типы данных описаны в системных файлах <sys/types.h> и <unistd.h>.

2. Организация взаимодействия процессов

Все процессы так или иначе взаимодействуют между собой. Самым простым способом обмена информацией как между родственными процессами, так и в рамках одного процесса, является неименованные каналы (pipe, труба, конвейер). Данные механизм реализует потоковую модель передачи данных только в одну сторону и представляет из себя область памяти, которая недоступна пользовательским процессам напрямую, а доступ ко «входу» и «выходу» осуществляется через специально объявленные дескрипторы. Каналы удобны тем, что позволяют обмениваться данными программам, которые первоначально не были написаны для этого. Например, в команда в терминале:

```
ls | grep x
```

запускает две утилиты ls и grep одновременно и при этом за счет использования спецсимвола | соединяет вывод первой утилиты со входом втором. В приведенном примере ls выводит список файлов в каталоге, а grep ищет строки из списка, которые содержат символ x, таким образом операционная система создает неименованный канал, который недоступен пользователю и существует только в рамках двух запущенных утилит.

Один канал может быть использовать для **одного** направления передачи данных и только в рамках взаимодействия родственных процессов. Для организации двунаправленной связи между родственными процессами необходимо использовать два неименованных канала.

Взаимодействие между процессами, созданными в рамках одной программы можно организовать с помощью системного вызова *pipe()*. Поясним его принцип работы на примере:

```
1.      ///***///
2.      int fd[2];
3.      pipe(fd);
4.      pid = fork();
5.      if(pid == 0)
6.              {
7.               close(fd[0]); /* Child process closes up input side of pipe */
8.               write(fd[1], string, (strlen(string)+1));
9.                        // working
10.             }
11.             else
12.             {
13.              close(fd[1]);               /* Parent process closes up output side of pipe */
14.              nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
15.                       // working
```

```
16.             }
17.      ///***///
```

Общая идея данной программы – передача сообщения от дочернего процесса к родительскому.

В строчке 2 создается файловый дескриптор, представляющий из себя массив из двух элементов. При инициализации канала (строка 3) первый элемент дескриптора открывается на чтение, а второй – на запись. После вызова fork (строка 4), каждый из процессов стал обладателем значения fd. В строках 7-9 дочерний процесс закрывает дескриптор чтения (т.к. мы будем вести запись данных) и с помощью системного вызова *write* записывает некоторую информацию. В родительском процессе (строки 13-15) наоборот, происходит закрытие дескриптора записи и вызов *read* для приема сообщения.

Практическое задание.

Следующие задания рекомендуется выполнять последовательно. Результаты работы программ и ответы на вопросы занесите в отчет.

3.1. Скомпилируйте и запустите следующий код.

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

int main()
{
        printf("PID= %d\n", getpid());
        printf("PPID= %d\n", getppid());
        return 0;
}
```

Данная программа возвращает значение PID и PPID процесса.

*Запустите программу несколько раз. Какие значение изменились? С помощью утилиты **ps** определите родителя запущенных вами процессов.*

3.2. Скомпилируйте и запустите следующую программу.

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

int main()
{
pid_t PID=fork();
if(PID==0)
{
        printf("Hello ");
        sleep(30);
}
else
{
         wait(0);
         printf("world!\n");
}
return 0;
}
```

*Запустите в соседнем окне еще один терминал и выполните команду “ ps -ao user,pid,ppid,rss,stat,cmd”. Сколько процессов было порождено запуском программы? Объясните значения в полях **PID** и **PPID**.*

3.3. Скомпилируйте две программы – *Parent.c* и *Child.c*. Обратите внимание на то, что название программы *Child.c* должно быть *child*.

Child.c

```
#include<unistd.h>
#include<stdio.h>

int main()
{
         printf("I am Child!\n");
         sleep(30);
         return 0;
}
```

Parent.c

```
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>

int main()
{
        execl("child","child", NULL);
        printf("I am Parent!\n");
        sleep(30);
         return 0;
}
```

*Запустите программу **Parent**. Обратите внимание на сообщение, выведенное на экран. Аналогично предыдущему пункту проанализируйте результат вывода **ps**.*

3.4. Скомпилируйте и запустите следующий код.

```
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>

int main()
{
    pid_t pid = fork();
    if (pid==0)
    {
        printf("PID = %d, PPID = %d\n", getpid(), getppid());
        sleep(2);
        printf("PID = %d, PPID = %d\n", getpid(), getppid());
    }
    else
    {
        sleep(1);
        exit(0);
    }
}
```

*Проанализируйте возвращенные значения **PID** и **PPID** в первой и второй строчке. Объясните смоделированную ситуацию.*

3.5. Скомпилируйте и запустите следующий код.

```
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>

int main()
{
    pid_t pid = fork();
    if (pid==0)
    {
        printf("I am alive!!\n");
    }
    else
    {
        sleep(30);
    }
     return 0;
}
```

Аналогично проанализируйте результат вывода ps. Обратите внимание на столбцы STAT, RSS. Объясните их значения в запущенном вами процессе и порожденным им дочернем. Дождитесь завершения процесса и запустите утилиту ps еще раз. Объясните полученное значение.

3.6. Скомпилируйте и запустите следующий код.

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

int main(void)
{
        int     fd[2];
        pid_t   pid;
        char    string[] = "Hello, Parent!\n";
        char    readbuffer[80];
        pipe(fd);
        pid = fork();
        if(pid == 0)
        {
                close(fd[0]);
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                close(fd[1]);
                read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Parent received: %s", readbuffer);
        }

        return(0);
}
```

Проанализируйте полученный результат. Измените программу так, чтобы происходил двойной обмен сообщениями – после получения строки от дочернего процесса, он возвращал строку «Hello, Child».

3.7. Напишите программу, порождающую три процесса, которые по кругу обмениваются сообщением приветствия.

Контрольные вопросы

1. В чем отличие процесса от программы?
2. В чем отличие между системными вызовами fork и exec?
3. Что такое каналы в ОС? Для чего они предназначены?
4. Что будет выведено на экран в результате работы следующей программы:

```
1. //***//
2. fork();
3. fork();
4. fork();
5. printf(“Hello world!”);
6. //***//
```

Список рекомендованной литературы

1. Э. Таненбаум и Х. Бос, Современные операционные системы, Санкт-Петербург: Питер, 2021, с. 1119.
2. В. Е. Карпов и К. А. Коньков, Основы операционных систем, Москва: Физматкнига, 2019, с. 326.
3. Р. Лав, Ядро Linux. Описание процесса разработки, Москва, Санкт-Петербург, Киев: Вильямс, 2013, с. 51 - 73.
4. В. Столлингс, Операционные системы. Внутренняя структура и принципы проектирования, Москва: Диалектика, 2020, с. 1261.
5. Р. Арпачи-Дюссо и А. Арпачи-Дюссо, Операционные системы. Три простых элемента, Москва: ДМК, 2021, с. 730.