Лабораторная работа по курсу "Операционные системы"

Процессы в ОС Linux (II)

Цель работы: знакомство с системными вызовами для синхронизации процессов и обработки сигналов; изучение распределения виртуальной памяти процесса.

I. Синхронизация процессов и обработка сигналов

Ожидание завершения процесса.

Функция **wait**. **pid_t wait (int *status)**

Блокирует вызывающий процесс до тех пор, пока не завершится один из его дочерних процессов. Если к моменту выдачи **wait** дочерний процесс уже завершен, то возвращает управление немедленно. Если ***status** не равен NULL, то в переменную **status** записывается код завершения дочернего процесса (см. подробнее в лаб. работе 3). Функция **wait** возвращает PID завершившегося процесса или -1, если дочерние процессы отсутствуют (в последнем случае устанавливает переменную **errno**).

Функция **waitpid**. **pid_t waitpid (pid_t pid,int *status, int options)**

Блокирует вызывающий процесс до тех пор, пока не завершится дочерний процесс с PID, заданным в первом параметре. Если **pid=-1**, то ожидает завершения любого дочернего процесса (как и **wait**). Если задать в **options** значение **WNOHANG**, то **waitpid** работает в неблокирующем режиме (асинхронно): если нет завершившихся дочерних процессов, то управление немедленно возвращается в вызывающий процесс, при этом **waitpid** возвращает 0 и устанавливает **errno**.

Состояние "зомби". Когда процесс завершается, информация о нем сохраняется в системных таблицах для того, чтобы родительский процесс мог получить информацию о нем, в частности, его код возврата при помощи функции **wait**. Если же родительский процесс не вызывает **wait**, то дочерний процесс продолжает существовать в так называемом состоянии "зомби". Наличие таких процессов нежелательно, т.к. они занимают место в системной таблице процессов. Если родительский процесс завершится, не вызвав **wait**, то процесс-потомок удаляется из памяти процессом **init**.

Задание 1.

1.1. В программе **pr1.c** основной процесс создает три дочерних процесса и ожидает их завершения. *Выполните программу и запишите выходные данные. Нарисуйте приблизительную временную диаграмму работы всех четырех процессов.*

```
/* pr1.c */
#include <stdio.h>
#include <unistd.h>
int main()
{int i, pid, status, w;
for (i=0; i<3; ++i)
    {
    pid = fork();
    if (pid ==0)
        exit(getpid() % 256);
    }
while ((w = wait(&status))&& w!= -1)
    printf ("Child %x status %x\n", w, status);
return 0;
}
```

1.2. Замените **exit** на **kill(pid_дочернего_процесса, SIGKILL)**. *Поясните в отчете полученные в 1.1, 1.2 коды возврата дочернего процесса* (см. лаб. работу 3).

1.3. Напишите программу **pr2.c**, где основной процесс создает дочерний процесс, завершающийся немедленно. Затем основной процесс "засыпает" на 30 секунд. Запустите программу, и, пока она выполняется, из другого окна введите команду

% ps -mAl

Выпишите и поясните информацию о состоянии обоих процессов. Повторите эту команду после завершения работы программы. *Объясните результаты в отчете*.

Сигналы. Сигнал можно послать двумя способами:

- командой оболочки **$kill [-s номер_сигнала] pid**
- из программы функцией **kill (pid, номер_сигнала);**

Если включить **<signal.h>**, то вместо номеров можно использовать имена сигналов. Более подробную информацию о сигналах можно получить по команде

$man 7 signal

Для завершения процесса используются сигналы **SIGTERM** или **SIGKILL**. **SIGTERM** может быть проигнорирован процессом, в то время как **SIGKILL** всегда вызывает немедленное завершение процесса. Получить *асинхронное уведомление о завершении дочернего процесса* можно при помощи сигнала **SIGCHLD**, который ОС посылает родительскому процессу при завершении его потомка.

Задание 2.

2.1. Завершите **bash** сигналом **SIGTERM**, затем - сигналом **SIGKILL**. *Поясните результаты и запишите Ваши команды. Определите и выпишите номера этих двух сигналов.*

2.2. Напишите программу **pr3.c**, которая содержит функцию-обработчик для сигнала **SIGCHLD**, обработчик вызывает функцию **wait** и записывает код возврата дочернего процесса в некоторую глобальную переменную. Основной процесс создает дочерний процесс (который завершается немедленно и возвращает какой-нибудь код возврата, например, 2), затем основной процесс в цикле выполняет некоторую работу и проверяет значение глобальной переменной. После завершения дочернего процесса распечатывает его код возврата. Используйте следующий код для функции-обработчика сигнала и для задания реакции на сигнал **SIGCHLD**:

```
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
sig_atomic_t exstatus;
void chld_hand (int snumber)
{
int status;
wait (&status);
exstatus = status;
}
int main ()
{struct sigaction sigact;
memset (&sigact, 0, sizeof (sigact));
sigact.sa_handler = chld_hand;
sigaction (SIGCHLD, &sigact, NULL);
...
return 0;
}
```

ПРИМЕЧАНИЕ. Так как обработчик может быть прерван в любом месте, например, поступлением другого сигнала, то присвоение значения глобальной переменной должно быть неделимой (атомарной) операцией. Это обеспечивается использованием переменной типа **sig_atomic_t.**

2.3. Реализация тайм-аута при помощи сигнала-"будильника" SIGALRM. Напишите программу **pr4.c**, которая ожидает ввода имени с клавиатуры, печатает "*Привет, <имя>*" и завершается. Если имя не было введено в течение пяти секунд, то программа печатает "*Время истекло*" и завершается.

ПРИМЕЧАНИЯ.

1. Задайте обработчик для сигнала **SIGALRM** с пустым телом функции-обработчика.
2. Перед вызовом функции **read** запустите будильник функцией **alarm**.
3. Если **read** завершилась с ошибкой, то следует проверить значение **errno**. Если **errno** равно **EINTR**, то истек тайм-аут.

*Поясните в отчете, почему возникает ошибка **EINTR**. Повлияет ли на работу программы установка флага **SA_RESTART** в поле **sa_flags** структуры **sigaction**? Почему?*

2.4.* **Одновременное поступление сигналов одного типа.** Напишите программу **pr5.c**.

Основной процесс: создает пять потомков и затем выполняет какой-либо бесконечный цикл.

Код потомка: **sleep(1); exit(…);**

Код обработчика сигнала SIGCHLD: **sleep(3); wait(0); write(1, "process terminated\n", 19);**

Объясните в отчете полученные результаты. Сколько сигналов обработано и почему? Модифицируйте программу, попытавшись обработать все сигналы следующими способами:

a) (**pr5a**) используя в обработчике вместо **wait** функцию **waitpid** в цикле.

b) (**pr5b**) используя флаг **SA_NOMASK** в поле **sa_flags** структуры **sigaction** (см. **man**).

c) (**pr5c**) используя вместо обработчика сигнала функцию **sigwait**.

Тексты программ и результаты объясните в отчете.

* Необязательное задание. Оценивается дополнительно, если студент полностью выполнил обязательные задания.

II. Распределение виртуальной памяти

В ОС Linux адреса виртуальной памяти каждого процесса могут быть получены из глобальных переменных **etext**, **edata**, **end** (рис. 1).

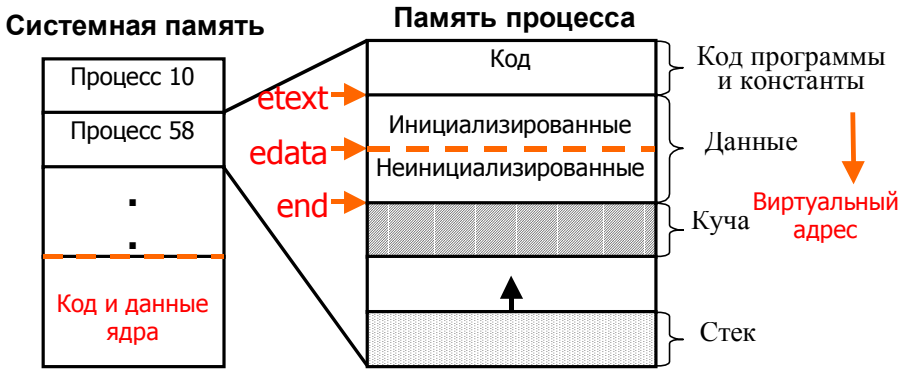

Рис. 1. Схема распределения виртуальной памяти

Задание 3. **3.1.** Используя программу **procmemory.c** , *зарисуйте в отчете распределение памяти процесса*, как это сделано на рис. 1. *Покажите на рисунке расположение и адреса переменных* ***main***, ***showit***, ***cptr***, ***buffer1***, ***buffer2***, ***i*** *и память, на которую указывает* ***buffer2***.

```
/* procmemory.c */
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
/* Below is a macro definition */
#define SHW_ADR(ID,I) (printf("ID %s \t is at virt.address: %8X\n", ID,&I))
extern int etext, edata, end; /* Global variables for process memory */
char   *cptr = "This message is output by showit()\n"; /* Static */
char   buffer1[25];
int showit (); /* Function prototype */

main ()
{int i = 0;            /* Automatic variable */
/* Printing addressing information */
printf ("\nAddress etext: %8X \n", &etext);
printf ("Address edata: %8X \n", &edata);
printf ("Address end  : %8X \n", &end);
SHW_ADR ("main", main);
SHW_ADR ("showit", showit);
SHW_ADR ("cptr", cptr);
SHW_ADR ("buffer1", buffer1);
SHW_ADR ("i", i);
strcpy (buffer1, "A demonstration\n");   /* Library function */
write (1, buffer1, strlen(buffer1) + 1); /* System call */
showit(cptr);
} /* end of main function */

int showit (p)          /* A function follows */
char * p;
{char *buffer2;
SHW_ADR("buffer2", buffer2);
if ((buffer2 = (char *)malloc ((unsigned) (strlen(p) + 1))) != NULL)
   {printf("Alocated memory at %X\n", buffer2);
   strcpy (buffer2, p);    /* copy the string */
   printf ("%s", buffer2); /* Didplay the string */
   free (buffer2);         /* Release location */
   }
else
   {printf ("Allocation error\n");
   exit (1);
   }
}
```

3.2. Вставьте в программу системный вызов **fork** после оператора **int i=0**. *Что будет напечатано? Одинаковы ли виртуальные адреса переменных в обоих процессах?*

Порядок выполнения лабораторной работы

1. Выполните задания 1-3.
2. Занесите в отчет описание заданий со всеми требуемыми пояснениями.

Требования

1. При подготовке к лабораторной работе (дома) занесите в конспект: а) описания системных вызовов для управления процессами и сигналами, которые используются в лабораторной работе; тексты всех программ (кроме **procmemory.c**) с комментариями.
2. Студент должен знать ответы на следующие вопросы:

Вопросы

1. Каково назначение системных вызовов **wait** и **waitpid**?
2. Что такое состояние "зомби" процесса?
3. Как в родительском процессе можно узнать о завершении дочернего процесса, не прерывая основной работы?
4. Как в программе задать реакцию на сигнал определенного типа?
5. Все ли функции можно использовать в обработчике сигналов?
6. Как послать сигнал определенному процессу? группе процессов? всем процессам?
7. Может ли обработчик сигналов быть прерван сигналом другого типа? аналогичного типа?
8. Как в обработчике сигналов блокировать поступление сигналов определенных типов?
9. Что происходит, если сигнал поступает во время блокировки процесса на системном вызове, например, когда процесс ожидает ввода с клавиатуры?
10. Каково назначение директивы **#define** в программе **procmemory.c**?
11. Каков смысл переменных **etext**, **edata** и **end** в программе **procmenory.c**? Почему эти переменные объявлены с атрибутом **extern**?
12. В родительском процессе некоторой переменной было присвоено значение до создания дочернего процесса. Будет ли эта переменная доступна в дочернем процессе? Каково будет ее значение в начальный момент времени? Если родительский процесс будет изменять значение переменной, то будет ли это изменение доступно дочернему процессу? А наоборот?
