Министерство науки и образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
"Московский институт электронной техники"
(МИЭТ)

Отчет по лабораторной работе № 4

Операционные системы

Выполнил: студент ПМ - 31

Мартынова Мария Олеговна

2023 г.

Задание 1

Дополнить программу parallel_min_max.c из ***лабораторной работы №3***, так чтобы после заданного таймаута родительский процесс посылал дочерним сигнал SIGKILL. Таймаут должен быть задан, как именной необязательный параметр командной строки (--timeout 10). Если таймаут не задан, то выполнение программы не должно меняться.

Команда kill в Unix/Linux | linux-notes.org
man kill (2): послать сигнал процессу (manpages.org)
Команда KILL в Linux. Описание и примеры (pingvinus.ru)

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

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

Системный вызов **kill**() может быть использован для посылки какого-либо сигнала какому-либо процессу или группе процессов.

Если значение *pid* является положительным, то сигнал *sig* посылается процессу с идентификатором *pid*.

Если значение *pid* равно 0, то *sig* посылается каждому процессу, который входит в группу вызывающего процесса.

Если значение *pid* равно -1, то *sig* посылается каждому процессу, которым вызывающий процесс имеет право отправлять сигналы, за исключением процесса с номером 1 (*init*), подробности смотрите далее.

Если значение *pid* меньше -1, то *sig* посылается каждому процессу, который входит в группу процессов, чей ID равен *-pid*.

Если значение *sig* равно 0, то никакой сигнал не посылается, а только выполняется проверка; это можно использовать для проверки существования процесса или группы процессов с заданным ID.

man waitpid (2): ожидает смену состояния процесса (manpages.org)
Проект OpenNet: MAN waitpid (2) Системные вызовы (FreeBSD и Linux)
Подробное объяснение функций wait () и waitpid () в Linux - Русские Блоги (russianblogs.com)

#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int ****status*);**
pid_t waitpid(pid_t *pid***, int ****status***, int** *options***);**

Данные системные вызовы используются для ожидания изменения состояния процесса-потомка вызвавшего процесса и получения информации о потомке, чьё состояние изменилось. Сменой состояния считается: прекращение работы потомка, останов потомка по сигналу, продолжение работы потомка по сигналу. Ожидание прекращения работы потомка позволяет системе освободить ресурсы, использовавшиеся потомком; если ожидание не выполняется, то прекративший работу потомок остаётся в системе в состоянии "зомби (zombie)"

Если состояние потомка уже изменилось, то вызов сразу возвращает результат. В противном случае, работа приостанавливается до тех пор, пока не произойдёт изменение состояния потомка или обработчик сигналов не прервёт. В оставшейся части страницы потомок, чьё состояние ещё не было получено одним из этих системных вызовов, называется *ожидаемым (waitable)*.

wait() и waitpid()

Системный вызов **wait**() приостанавливает выполнение вызвавшего процесса до тех пор, пока не прекратит выполнение один из его потомков. Вызов *wait(&status)* эквивалентен:

```
waitpid(-1, &status, 0);
```

Системный вызов **waitpid**() приостанавливает выполнение вызвавшего процесса до тех пор, пока не изменится состояние потомка, заданного аргументом *pid*. По умолчанию **waitpid**() ожидает только прекращения работы потомка, но это можно изменить через аргумент *options* как описано далее.

Значением *pid* может быть:

< -1

: означает, что нужно ждать любого потомка, чей идентификатор группы процессов равен абсолютному значению *pid*.

-1

: означает, что нужно ждать любого потомка.

0

: означает, что нужно ждать любого потомка, чей идентификатор группы процессов равен таковому у вызвавшего процесса.

> 0

: означает, что нужно ждать любого потомка, чей идентификатор процесса равен *pid*.

Проект OpenNet: MAN alarm (2) Системные вызовы (FreeBSD и Linux)
man signal (7): обзор сигналов (manpages.org)

#include <unistd.h>

unsigned int alarm(unsigned int *seconds***);**

Системный вызов **alarm** выполняет в вызвавший его процесс доставку сигнала " SIGALRM через *seconds* секунд.

Если *seconds* равно нулю, то никаких новых тревожных сигналов в очередь поставлено не будет.

Если случиться какое-либо событие (интересно какое? -- прим. пер.), любые предыдущие установки **alarm** отменяются.

```
~/oslab2019$ cd lab3/src
~/.../lab3/src$ make all
gcc -o sequential_min_max find_min_max.o utils.o sequential_min_max.c -I.
gcc -o parallel_min_max utils.o find_min_max.o parallel_min_max.c -I.
~/.../lab3/src$ ./parallel_min_max --seed=51 --array_size=200000000 --pnu
m=3 --timeout=1
Killed
~/.../lab3/src$ ./parallel_min_max --seed=51 --array_size=200000000 --pnu
m=3 --timeout=2
Killed
~/.../lab3/src$ ./parallel_min_max --seed=51 --array_size=200000000 --pnu
m=3 --timeout=3
Min: 59
Max: 2147483630
Elapsed time: 2106.029000ms
~/.../lab3/src$
```

```
18  void chikatilo()
19  {
20  kill(0,SIGKILL);
21  }
```

```
33  static struct option options[] = {{"seed", required_argument, 0, 0},
34                                    {"array_size", required_argument, 0, 0},
35                                    {"pnum", required_argument, 0, 0},
36                                    {"by_files", no_argument, 0, 'f'},
37                                    {"timeout",required_argument, 0, 0},
38                                    {0, 0, 0, 0}};
```

```
82              case 4:
83                timeout = atoi(optarg);
84                if (timeout<0)
85                {
86                  printf("Timeout must be a positive number");
87                    return 1;
88                }
89                break;
```

```
183   if(timeout > 0)
184   {
185     signal(SIGALRM, chikatilo);
186     alarm(timeout);
187   }
```

Задание 2

Создать программу, с помощью которой можно продемонстрировать зомби процессы. Необходимо объяснить, как появляются зомби процессы, чем они опасны, и как можно от них избавиться.

Как найти и убить зомби-процессы в системах Linux | (itsecforu.ru)

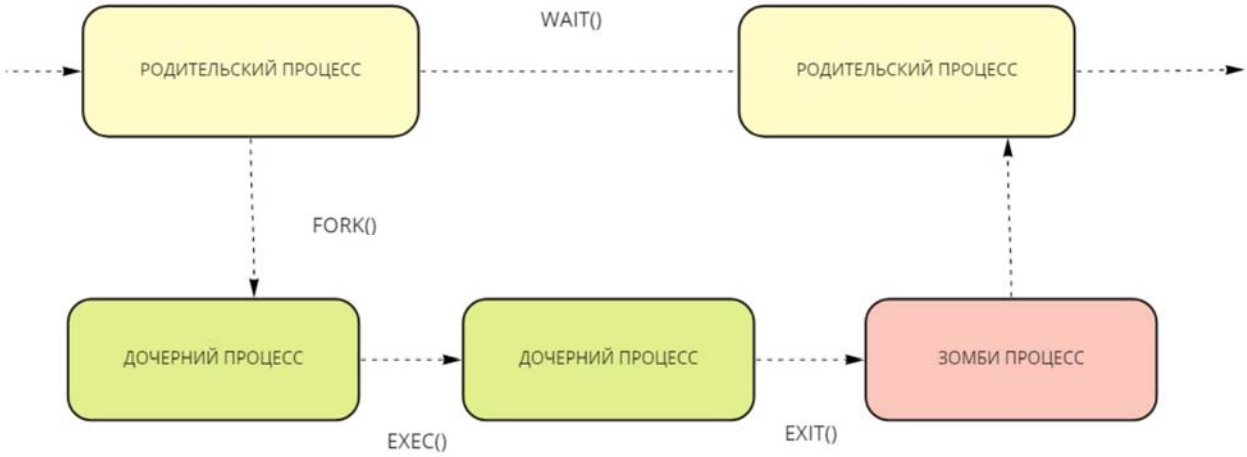
Процесс называется “зомби” или “мертвым” процессом, когда его выполнение завершено, но он все еще может попасть в таблицу процессов.

Родительский процесс считывает статус завершения своего дочернего процесса.

Это делается с помощью системного вызова wait().

Как только это сделано, зомби-процесс ликвидируется.

Чтобы лучше понять процесс образования и устранения зомби-процесса, посмотрите приведенную ниже схему.



Как работает состояние зомби-процесса

В состоянии зомби-процесса родитель вызывает одну функцию wait() во время создания дочернего процесса.

Затем он ждет изменения состояния дочернего процесса.

В случае изменения состояния, при котором дочерний процесс остановился, считывается его код состояния завершения.

После этого PCB (process control block) дочернего процесса уничтожается, а запись очищается.

Это происходит очень быстро, и зомби-процесс существует недолго.

Как найти и убить зомби-процессы

Чтобы убить зомби-процесс, сначала найдите его.

Используйте приведенные ниже команды для выявления зомби-процессов.

$ ps aux | egrep "Z|defunct"

Z, используемый в колонке STAT, и/или [defunct], используемый в последней колонке вывода, идентифицирует зомби-процесс.

На самом деле, вы не можете убить зомби-процессы, так как они уже мертвы.

Все, что вы можете сделать, это уведомить его родительский процесс, чтобы он мог снова попытаться прочитать статус дочернего процесса, который теперь стал зомби-процессом, и, в конце концов, мертвый процесс будет удален из таблицы процессов.

Используйте следующую команду, чтобы узнать ID родительского процесса.

$ ps -o ppid= <PID дочернего процесса>.

Как только вы узнаете ID родительского процесса зомби, отправьте SIGCHLD родительскому процессу.

$ kill -s SIGCHLD <Родительский PID>.

Если это не помогло удалить зомби-процесс из таблицы процессов, вам нужно перезапустить или убить его родительский процесс.

Чтобы убить родительский процесс зомби, используйте следующий код.

$ kill -9 <Родительский PID>

lab4/src/task2.c ×

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

int main ()
{
  pid_t child_pid;

  child_pid = fork ();
  if (child_pid > 0) {
    sleep (60);
  }
  else {
    exit (0);
  }
  return 0;
}
```

```
~/.../lab4/src$ ./task2 &
[1] 2728
~/.../lab4/src$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
runner         1  0.8  0.0 1684572 52388 ?        Ssl  06:07   0:26 /nix/store/nh7lm30k77cg
runner        17  0.0  0.0   7884  4612 pts/0    Ss+  06:15   0:00 /nix/store/8kgsjv57icc1
runner      2519  0.0  0.0   7896  4732 pts/3    Ss   06:56   0:00 /nix/store/8kgsjv57icc1
runner      2728  0.0  0.0   4392   864 pts/3    S    07:00   0:00 ./task2
runner      2729  0.0  0.0      0     0 pts/3    Z    07:00   0:00 [task2] <defunct>
runner      2735  0.0  0.0  37384  3232 pts/3    R+   07:00   0:00 ps aux
```

Задание 3

Скомпилировать process_memory.c. Объяснить, за что отвечают переменные etext, edata, end.

28) Виртуальная память в ОС - CoderLessons.com

Виртуальная память – распространенная стратегия распределения памяти, используемая во всех современных операционных системах, основанная на идее расширения физической памяти путем размещения расширенной памяти на диске и использования таблиц страниц (или сегментов) для трансляции адресов.

Преимущества виртуальной памяти

К основным преимуществам виртуальной памяти относят:

1. избавление программиста от необходимости управлять общим пространством памяти,
2. повышение безопасность использования программ за счет выделения памяти,
3. возможность иметь в распоряжении больше памяти, чем это может быть физически доступно на компьютере.

Свойства виртуальной памяти

Виртуальная память делает программирование приложений проще:

- скрывая фрагментацию физической памяти;
- устраняя необходимость в программе для обработки наложений в явном виде;
- когда каждый процесс запускается в своем собственном выделенном адресном пространстве, нет необходимости переместить код программы или получить доступ к памяти с относительной адресацией.

Виртуализация памяти может рассматриваться как обобщение понятия виртуальной памяти.

Почти все реализации виртуальной памяти делят виртуальное адресное пространство на страницы, блоки смежных адресов виртуальной памяти.

При работе машины с виртуальной памятью, используются методы страничной и сегментной организации памяти.

```
~/.../lab4/src$ ./process_memory

Address etext: 3B200B9D
Address edata: 3B402018
Address end  : 3B402050
ID main      is at virtual address: 3B2008BA
ID showit    is at virtual address: 3B200A37
ID cptr      is at virtual address: 3B402010
ID buffer1   is at virtual address: 3B402030
ID i     is at virtual address: 37C24604
A demonstration
ID buffer2   is at virtual address: 37C245E0
Alocated memory at 3B869670
This message is output by the function showit()
```

Задание 4

Создать makefile, который собирает программы из задания 1 и 3.

```
1  CC=gcc
2  CFLAGS=-Wall
3
4  process_memory:
5    $(CC) -o process_memory process_memory.c $(CFLAGS)
6
7  clean :
8    rm process_memory
```

```
~/.../lab4/src$ make clean
rm process_memory
~/.../lab4/src$ make process_memory
gcc -o process_memory process_memory.c -Wall
```

Задание 5

Доработать parallel_sum.c так, чтобы:

- Сумма массива высчитывалась параллельно.

- Массив генерировался с помощью функции GenerateArray из ***лабораторной работы №3***.

- Программа должна принимать входные аргументы: количество потоков, seed для генерирования массива, размер массива (./psum --threads_num "num" --seed "num" --array_size "num").

- Вместе с ответом программа должна выводить время подсчета суммы (генерация массива не должна попадать в замер времени).

- Вынести функцию, которая считает сумму в отдельную библиотеку.

03_pthreads_txt.pdf (nsu.ru)

OpenNET: статья - Многопотоковое программирование под Linux (threads linux gcc)

Компилировать (точнее линковать) надо с опцией -lpthread

```
~/.../lab4/src$ gcc -o sum.o sum.c -c
~/.../lab4/src$ ar rcs libsum.a sum.o
```

```
~/.../lab4/src$ ./parallel_sum --seed=3 --array_size=300 --threads_num=5
Total: 786113764
Elapsed time: 0.236000ms
```

Задание 6

Создать makefile для parallel_sum.c.

```
1  CC=gcc
2  CFLAGS=-Wall
3
4  process_memory:
5    $(CC) -o process_memory process_memory.c $(CFLAGS)
6
7  parallel_sum : utils.o utils.h
8    $(CC) -pthread -o parallel_sum utils.o parallel_sum.c -L. -lsum
   $(CFLAGS)
9
10 utils.o : utils.h
11   $(CC) -o utils.o -c utils.c $(CFLAGS)
12
13 clean :
14   rm process_memory parallel_sum utils.o sum.o
```