МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В. Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа № 1

по дисциплине: Операционные системы
тема: «Системные вызовы. Базовая работа с процессами в ОС Linux»

Выполнил: ст. группы ПВ-223
Игнатьев Артур Олегович

Проверил:
доц. Островский Алексей Мичеславович
асс. Четвертухин Виктор Романович

Белгород 2024г.

Цель работы: изучить основы работы с системными вызовами и процессами в операционной системе Linux (Ubuntu).

Условие индивидуального задания: Породить один процесс. Аккуратно клонировать его до тех пор, пока имеются в ОС свободные ресурсы. Найти критическое значение для мощности порожденных клонов, когда дальнейшее увеличение числа процессов неприемлемо. Корректно завершить все процессы. Описать поведение Linux (Ubuntu). Провести эксперименты в виртуальной машине для разного объёма ОЗУ.

Ход выполнения работы

Задание 1

Текст программы:

lab1.c

```c
#include <stdio.h> // Подключаем стандартную библиотеку для работы
 // с функциями ввода и вывода (printf, scanf и т.д.)
#include <stdlib.h> // Подключаем библиотеку для работы с различными
 // функциями стандартной библиотеки, например,
// для работы с памятью и функцией exit()
#include <unistd.h> // Подключаем библиотеку для работы с системными
 // вызовами UNIX, такими как fork(), getpid(), sleep()
#include <time.h> // Подключаем библиотеку для работы с временем,
 // используем для генерации случайных чисел и
// отображения системного времени
#include <sys/wait.h> // Подключаем библиотеку для работы с процессами,
 // в частности, для ожидания завершения порожденных
// процессов с помощью waitpid()
#include <signal.h> // Подключаем библиотеку для работы с сигналами
 // в процессе (pause(), управление сигналами)
int main() {

 // PID используется для идентификации процессов в операционной системе
 pid_t process_1_pid_global = -1, process_2_pid_global = -1,
 process_3_pid_global = -1, process_4_pid_global = -1;
 int status; // Используется в функциях waitpid() для отслеживания
 // завершения процессов
 // Функция getpid() возвращает идентификатор текущего процесса
 process_1_pid_global = getpid();
 printf("Процесс 1 (PID: %d)\n", process_1_pid_global);
 // Порождаем новый процесс 2 с помощью системного вызова fork()
 // fork() создает копию текущего процесса (процесс 1),
 // которая становится новым процессом (процессом 2)
 // Возвращаемое значение:
 // - В родительском процессе (процесс 1) fork() возвращает PID
 // нового процесса (процесса 2)
 // - В процессе-потомке (процессе 2) fork() возвращает 0

 process_2_pid_global = fork();
```

```
// Проверяем, находимся ли мы в процессе-потомке (процессе 2)
if (process_2_pid_global == 0) { // Это процесс 2

// Функция getppid() возвращает PID родительского процесса
// (в данном сучае процесс 1)
// Этот идентификатор используется для указания, какой
// процесс породил текущий процесс (процесс 2)
// В контексте этого кода, getppid() вернет PID процесса 1
// для процесса 2

printf("Процесс 2 (PID: %d) создан процессом 1 (PID: %d)\n",
getpid(), getppid());

// Порождаем процесс 3
process_3_pid_global = fork();
if (process_3_pid_global == 0) { // Это процесс 3
printf("Процесс 3 (PID: %d) создан процессом 2 (PID: %d)\n",
getpid(), getppid());

// Порождаем процесс 4
process_4_pid_global = fork();
if (process_4_pid_global == 0) { // Это процесс 4
printf("Процесс 4 (PID: %d) создан процессом 3 (PID: %d). "
"Завершает свою работу.\n", getpid(), getppid());
exit(0);
} else {
// Процесс 3 ожидает завершения процесса 4
// Ожидаем завершения процесса 4 с помощью waitpid()
// process_4_pid_global — это PID процесса 4
// waitpid() приостанавливает выполнение текущего
// процесса (процесс 3),
// до тех пор, пока процесс 4 не завершится
// Аргументы:
// - process_4_pid_global: PID процесса
// - &status: указатель на переменную, куда будет
// записан статус завершения процесса 4
// - 0: процесс будет приостановлен до завершения
// дочернего процесса (ожидание завершения)
waitpid(process_4_pid_global, &status, 0);
srand(time(NULL) ^ (getpid() << 16));
while (1) {
sleep(1); // Ждать 1 сек.
if (rand() % 3 == 0) {
printf("Процесс 3 (PID: %d) завершает работу "
"(срабатывание 1/3)\n", getpid());
exit(0);
// Завершает процесс с кодом 0,
// указывая на успешное завершение
}
}
}
} else {
```

```
while (1) {
sleep(1);
// Проверяем, завершился ли процесс 3 с помощью
// waitpid()
// process_3_pid_global — это PID процесса 3
// waitpid() не блокирует выполнение текущего
// процесса, так как используется флаг WNOHANG
// Аргументы:
// - process_3_pid_global: PID процесса
// - &status: указатель на переменную для получения
// статуса завершения процесса
// - WNOHANG: флаг, который указывает, что waitpid()
// не должен блокировать выполнение,
// если процесс 3 ещё не завершился; функция
// вернет 0, если процесс всё ещё активен
pid_t result = waitpid(process_3_pid_global, &status, WNOHANG);
if (result == 0) {
printf("Процесс 3 (PID: %d) еще работает... "
"Ну, погоди!\n", process_3_pid_global);
} else if (result == process_3_pid_global) {
printf("Процесс 3 завершил свою работу.\n");
printf("Процесс 2 ждет 5 сек.\n");
sleep(5);
printf("Процесс 2 завершает свою работу.\n");
exit(0);
}
}
}
} else {
// Процесс 1 (родительский процесс) ожидает завершения
// процесса 2
waitpid(process_2_pid_global, &status, 0);
// Проверяем, завершился ли процесс нормально с помощью
// макроса WIFEXITED
// WIFEXITED(status) возвращает ненулевое значение (true),
// если процесс завершился
// нормально (через вызов exit())
// Если процесс завершился с ошибкой или был прерван сигналом,
// WIFEXITED вернет 0 (false)
if (WIFEXITED(status)) {
printf("Процесс 2 завершил свою работу.\n");
}
printf("Процесс 1 завершает свою работу и ждёт Ctrl+C\n");
time_t current_time = time(NULL);
printf("Текущее время: %s", ctime(&current_time));
while (1) {
pause();
// pause() приостанавливает выполнение процесса до
// получения любого сигнала, который не игнорируется.
// Процесс возобновит свою работу, как только
// будет получен сигнал. В данном случае это
// позволяет процессу 1 "ждать" завершения
```

```
 // программы, пока его не прервут вручную (например,
 // с помощью Ctrl+C).
 }

 printf("Процесс 1 завершил свою работу.\n");
 }

 return 0;
}
```

Вывод программы:

```
Процесс 1 (PID: 3284)
Процесс 2 (PID: 3285) создан процессом 1 (PID: 3284)
Процесс 3 (PID: 3286) создан процессом 2 (PID: 3285)
Процесс 4 (PID: 3287) создан процессом 3 (PID: 3286). Завершает свою работу.
Процесс 3 (PID: 3286) еще работает... Ну, погоди!
Процесс 3 (PID: 3286) еще работает... Ну, погоди!
Процесс 3 (PID: 3286) еще работает... Ну, погоди!
Процесс 3 (PID: 3286) завершает работу (срабатывание 1/3)
Процесс 3 завершил свою работу.
Процесс 2 ждет 5 сек.
Процесс 2 завершает свою работу.
Процесс 2 завершил свою работу.
Процесс 1 завершает свою работу и ждёт Ctrl+C
Текущее время: Wed Sep 25 02:48:27 2024
```

Задание 2

Текст программы
lab1_1.c

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

#define MAX_PROCESSES 10000  // Лимит на количество процессов для защиты системы

// Функция для выполнения некоторого вычисления
int perform_computation(int process_number) {
    int sum = 0;
    for (int i = 0; i < 1000000; i++) {
        sum += i + process_number;
    }
    return sum;
}

int main() {
    pid_t pid;
    int process_count = 0;
    int status;

    while (process_count < MAX_PROCESSES) {
        pid = fork();

        if (pid < 0) {  // Ошибка при форке
            printf("Невозможно создать больше процессов. "
                   "Всего создано процессов: %d\n", process_count);
            break;
        } else if (pid == 0) {  // Процесс потомок
            int result = perform_computation(process_count + 1);  // Выполняем вычисление
            printf("Процесс %d (PID: %d) создан процессом-родителем (PID: %d). Результат
вычисления: %d\n",
                   process_count + 1, getpid(), getppid(), result);

            // Потомки ждут в бесконечном цикле, пока их не завершат
            while (1) {
                sleep(10);
            }
        } else {  // Процесс родитель
            process_count++;
        }
    }

    // Завершаем все порожденные процессы
```

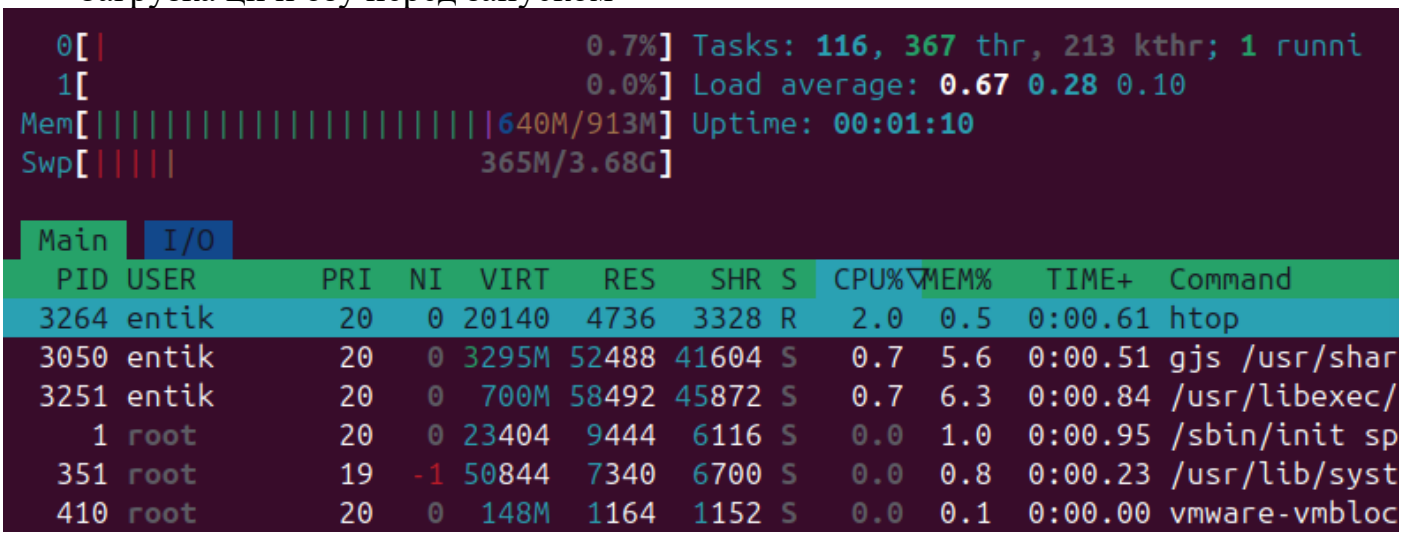
```c
    printf("Завершаем все порожденные процессы...\n");
    for (int i = 0; i < process_count; i++) {
        wait(&status);  // Ожидаем завершения потомков
    }

    printf("Все процессы завершены. Всего было создано процессов: %d\n", process_count);
    return 0;
}
```

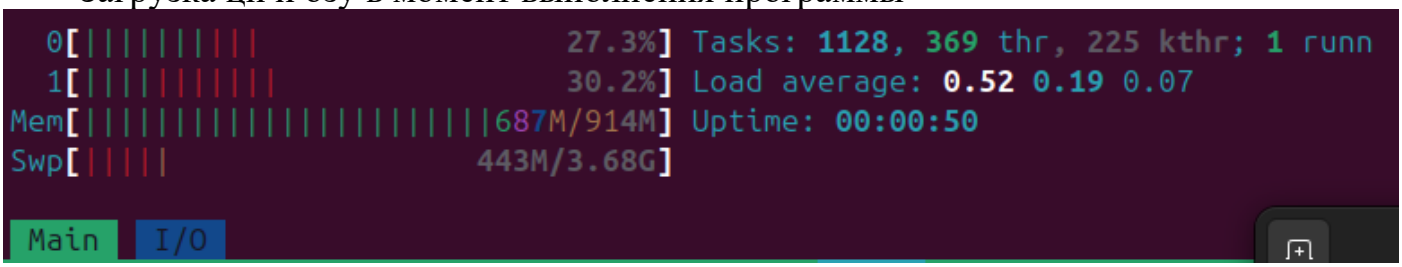
Вывод программы:

На 1gb ОЗУ

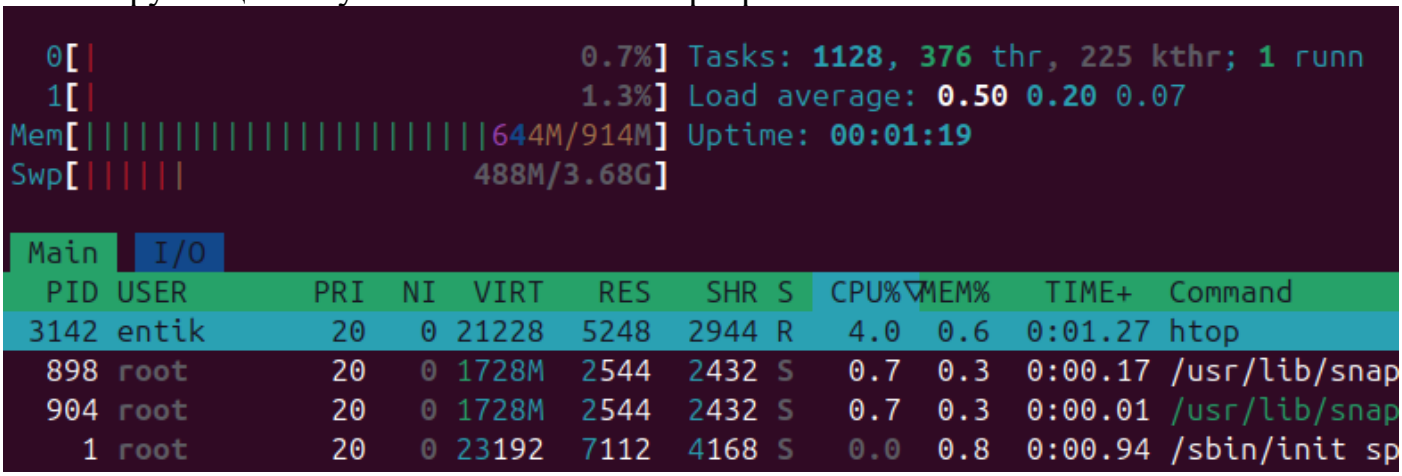
Загрузка цп и озу перед запуском



Загрузка цп и озу в момент выполнения программы



Загрузка цп и озу после выполнения программы

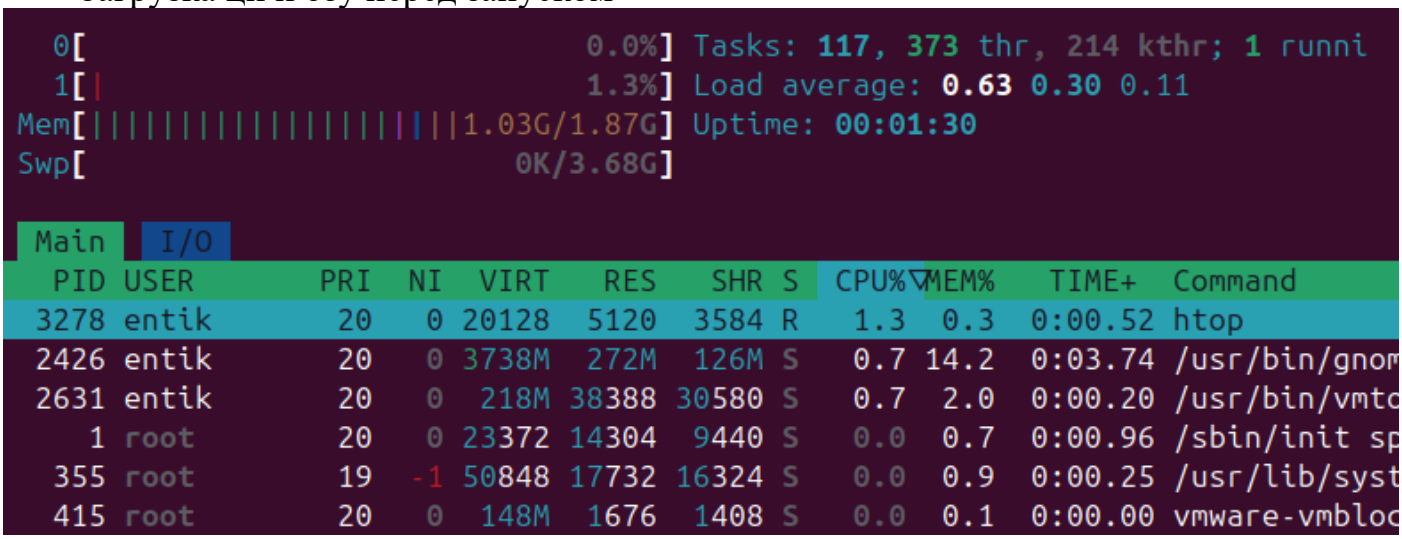


Кол-во созданных проццессов

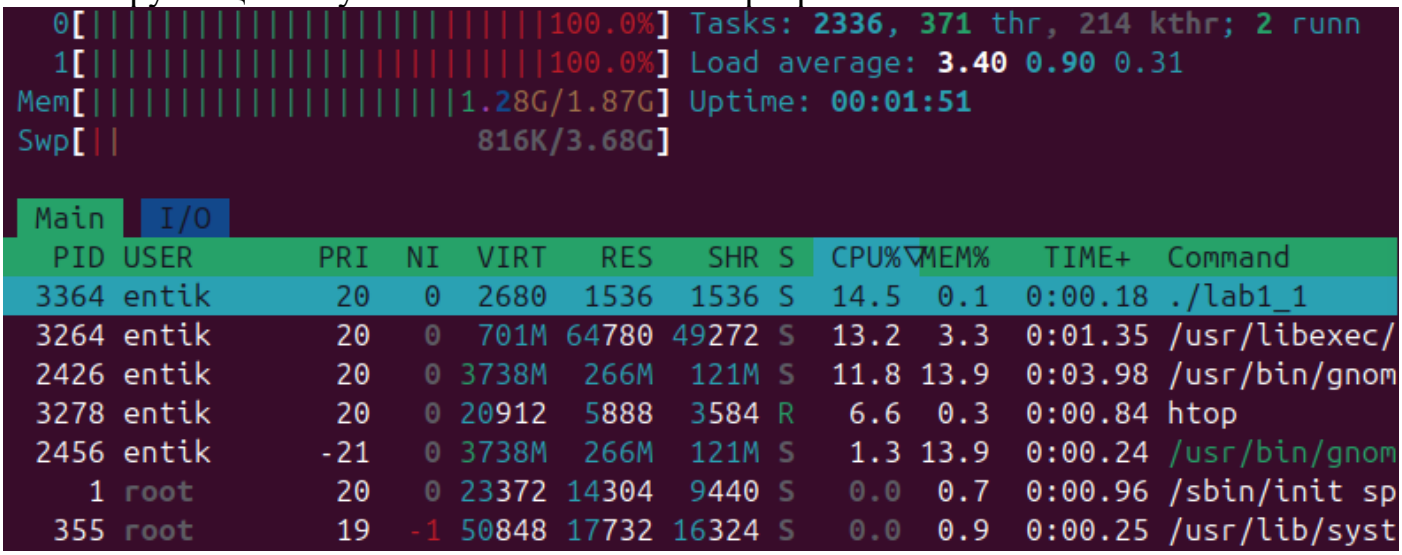
```
Процесс 947 (PID: 4218) создан процессом-родителем (PID: 3271). Результат вычисления: -1564673632
Процесс 968 (PID: 4239) создан процессом-родителем (PID: 3271). Результат вычисления: -1543673632
Процесс 992 (PID: 4263) создан процессом-родителем (PID: 3271). Результат вычисления: -1519673632
Процесс 972 (PID: 4243) создан процессом-родителем (PID: 3271). Результат вычисления: -1539673632
Процесс 987 (PID: 4258) создан процессом-родителем (PID: 3271). Результат вычисления: -1524673632
Процесс 963 (PID: 4234) создан процессом-родителем (PID: 3271). Результат вычисления: -1548673632
Процесс 993 (PID: 4264) создан процессом-родителем (PID: 3271). Результат вычисления: -1518673632
Процесс 975 (PID: 4246) создан процессом-родителем (PID: 3271). Результат вычисления: -1536673632
Процесс 994 (PID: 4265) создан процессом-родителем (PID: 3271). Результат вычисления: -1517673632
Процесс 976 (PID: 4247) создан процессом-родителем (PID: 3271). Результат вычисления: -1535673632
Невозможно создать больше процессов. Всего создано процессов: 1010
Завершаем все порожденные процессы...
```

На 2gb ОЗУ

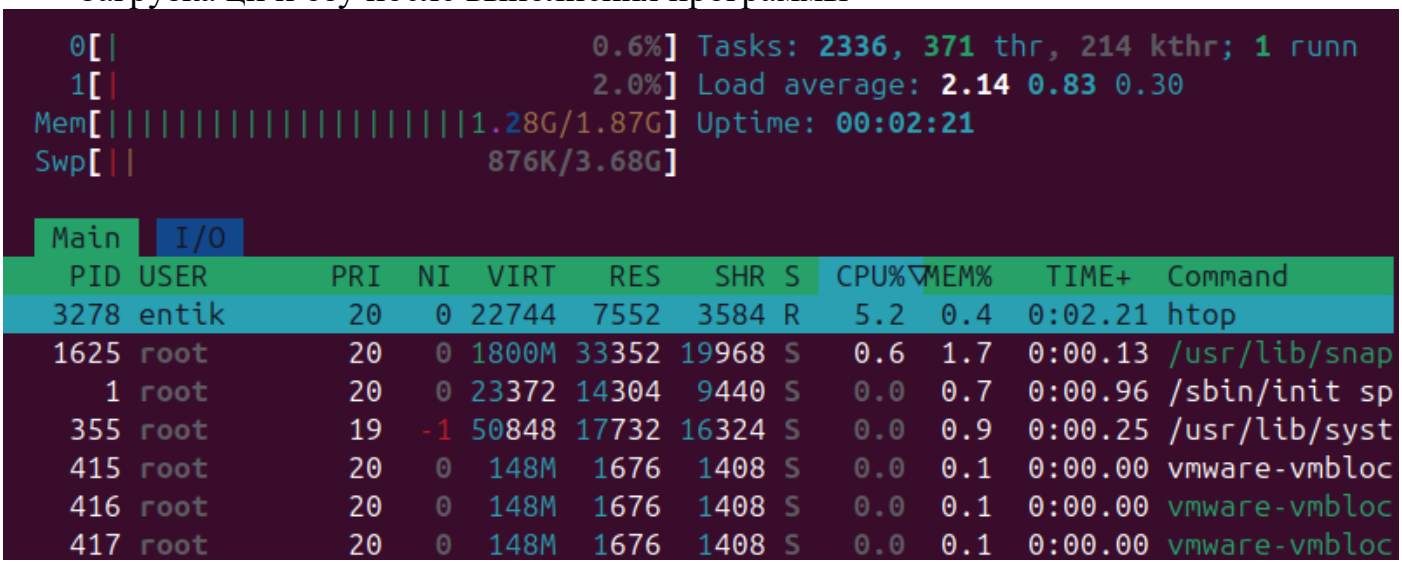
Загрузка цп и озу перед запуском



Загрузка цп и озу в момент выполнения программы



Загрузка цп и озу после выполнения программы

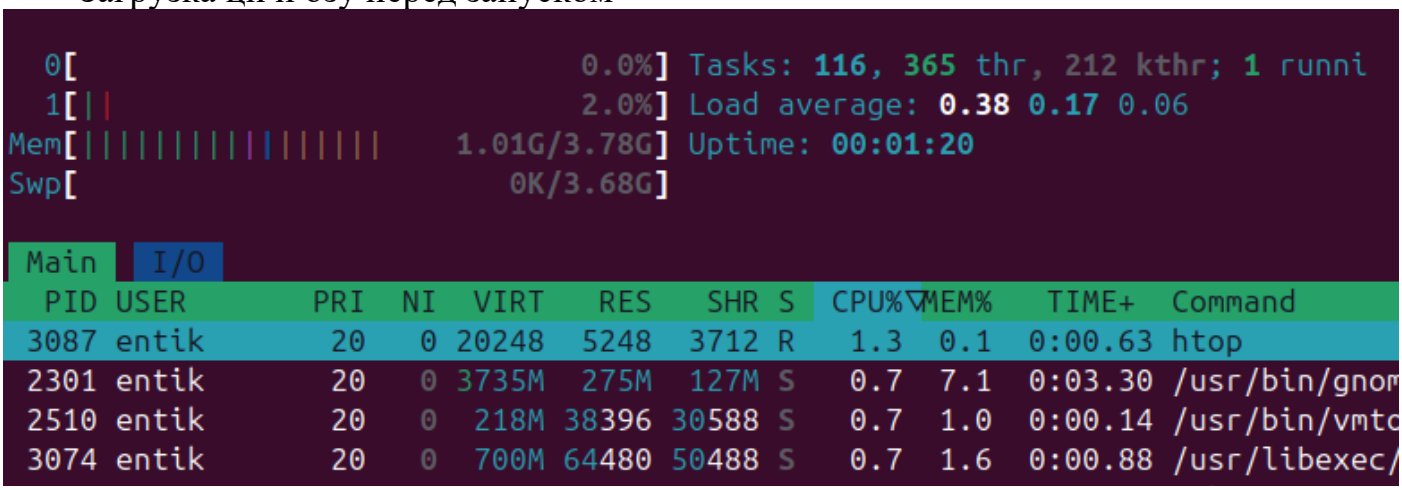


Кол-во созданных процессов

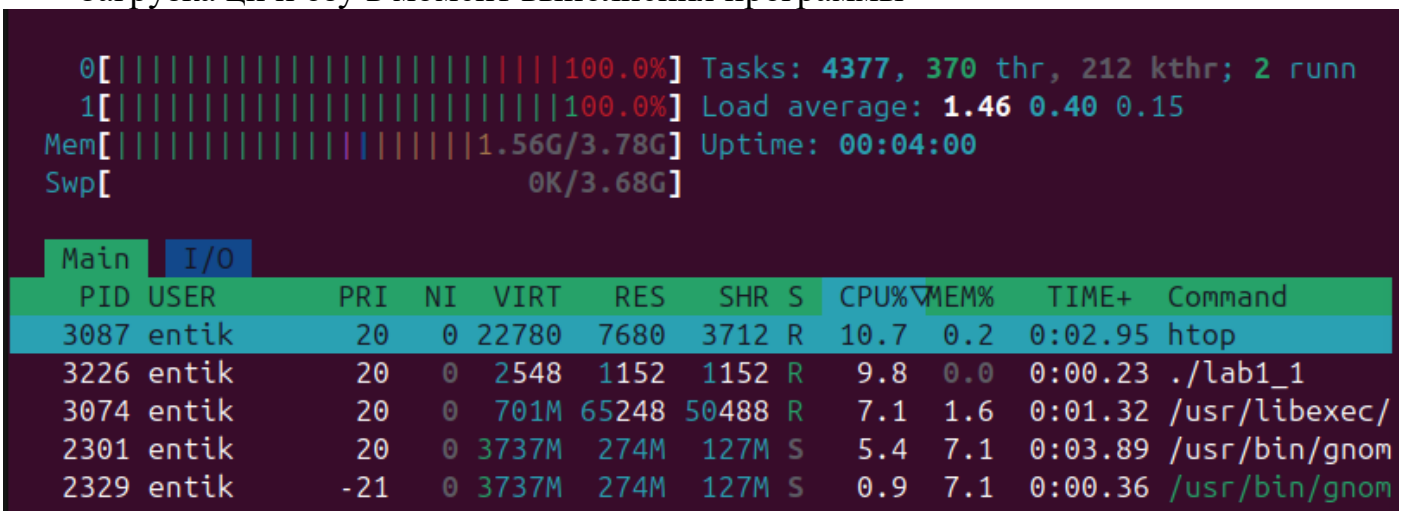
```
Процесс 2206 (PID: 5456) создан процессом-родителем (PID: 3247). Результат вычисления: -305673632
Процесс 2207 (PID: 5457) создан процессом-родителем (PID: 3247). Результат вычисления: -304673632
Процесс 2208 (PID: 5458) создан процессом-родителем (PID: 3247). Результат вычисления: -303673632
Процесс 2181 (PID: 5431) создан процессом-родителем (PID: 3247). Результат вычисления: -330673632
Процесс 2209 (PID: 5459) создан процессом-родителем (PID: 3247). Результат вычисления: -302673632
Процесс 2210 (PID: 5460) создан процессом-родителем (PID: 3247). Результат вычисления: -301673632
Процесс 2170 (PID: 5420) создан процессом-родителем (PID: 3247). Результат вычисления: -341673632
Процесс 2173 (PID: 5423) создан процессом-родителем (PID: 3247). Результат вычисления: -338673632
Процесс 2211 (PID: 5461) создан процессом-родителем (PID: 3247). Результат вычисления: -300673632
Процесс 2186 (PID: 5436) создан процессом-родителем (PID: 3247). Результат вычисления: -325673632
Процесс 2212 (PID: 5462) создан процессом-родителем (PID: 3247). Результат вычисления: -299673632
Процесс 2189 (PID: 5439) создан процессом-родителем (PID: 3247). Результат вычисления: -322673632
Невозможно создать больше процессов. Всего создано процессов: 2218
Завершаем все порожденные процессы...
```

На 4 gb ОЗУ

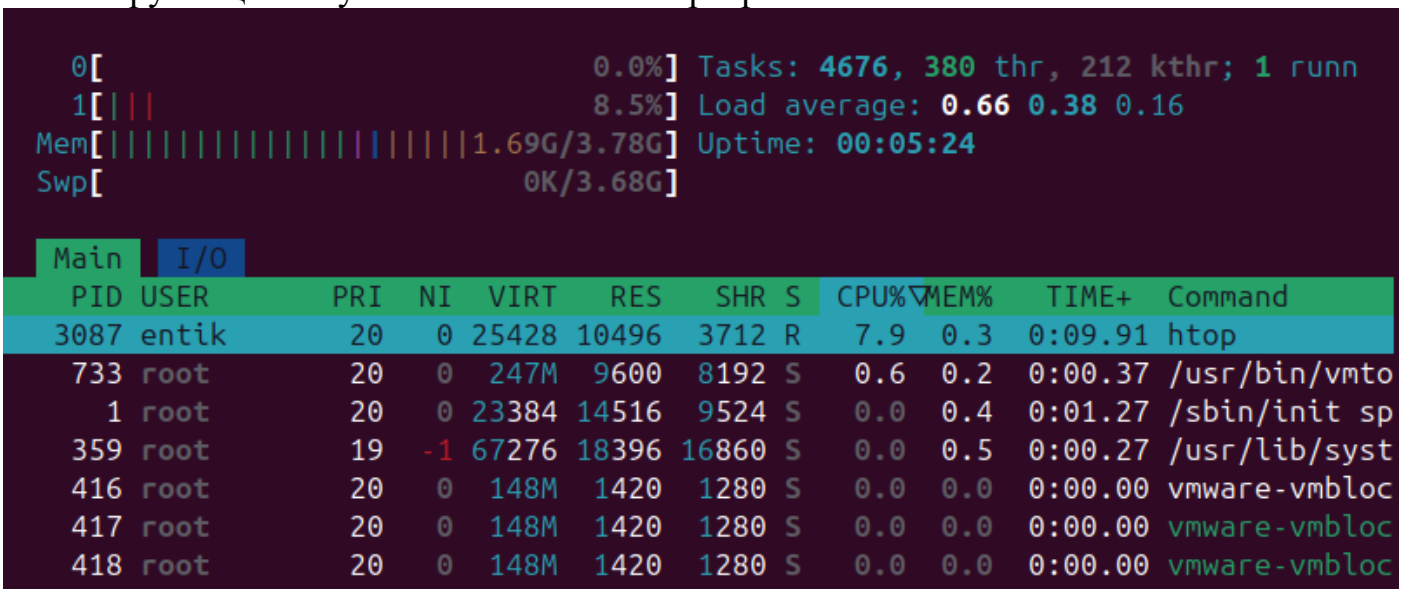
Загрузка цп и озу перед запуском



Загрузка цп и озу в момент выполнения программы



Загрузка цп и озу после выполнения программы



Кол-во созданных процессов

```
Процесс 4472 (PID: 7767) создан процессом-родителем (PID: 3295). Результат вычисления: 1960326368
Процесс 4547 (PID: 7842) создан процессом-родителем (PID: 3295). Результат вычисления: 2035326368
Процесс 4546 (PID: 7841) создан процессом-родителем (PID: 3295). Результат вычисления: 2034326368
Процесс 4527 (PID: 7822) создан процессом-родителем (PID: 3295). Результат вычисления: 2015326368
Процесс 4552 (PID: 7847) создан процессом-родителем (PID: 3295). Результат вычисления: 2040326368
Процесс 4508 (PID: 7803) создан процессом-родителем (PID: 3295). Результат вычисления: 1996326368
Процесс 4548 (PID: 7843) создан процессом-родителем (PID: 3295). Результат вычисления: 2036326368
Процесс 4551 (PID: 7846) создан процессом-родителем (PID: 3295). Результат вычисления: 2039326368
Процесс 4550 (PID: 7845) создан процессом-родителем (PID: 3295). Результат вычисления: 2038326368
Невозможно создать больше процессов. Всего создано процессов: 4556
Завершаем все порожденные процессы...
```

Задание 3

Отчет по работе

1. Краткое описание всех использованных системных вызовов

В процессе выполнения задания использованы следующие системные вызовы:

- fork(): этот системный вызов используется для создания нового процесса. Он создает копию текущего процесса. В родительском процессе возвращает PID нового процесса, а в потомке возвращает 0. Если процесс не может быть создан, возвращает-1

- Применение: использован для создания новых процессов в цикле до исчерпания ресурсов системы.

- getpid(): возвращает идентификатор (PID) текущего процесса.

- Применение: используется для вывода PID каждого создаваемого процесса.

- getppid(): возвращает идентификатор родительского процесса для текущего процесса.

- Применение: для вывода PID родителя при создании потомка.

- wait(): приостанавливает выполнение текущего процесса до тех пор, пока один из его потомков не завершит выполнение. Возвращает PID завершенного потомка.

- Применение: используется в родительском процессе для ожидания завершения всех порожденных процессов.

- sleep(): приостанавливает выполнение процесса на определенное количество секунд.

- Применение: используется в бесконечном цикле дочерних процессов для эмуляции работы до завершения программы.

2. Программы, написанные в ходе выполнения лабораторной работы

Программа для порождения процессов до исчерпания ресурсов:

3. Анализ:

В результате экспериментов было установлено, что количество процессов, которое система может поддерживать, напрямую зависит от объема оперативной памяти. При 1024 МБ удалось создать 1010 процессов, при 2 ГБ — 2218 процессов, а при 4 ГБ — 4556 процессов. Системные вызовы, такие как fork(), wait(), и другие, корректно обрабатывают создание и завершение процессов, что позволило аккуратно завершить все созданные процессы после выполнения программы.

Хочется отметить, что при объёме оперативной памяти в 256мб и 512мб, система не смогла загрузится. Также во время выполнения цп загружался на 100%, при этом фал подкачки не использовался, только при тестировании с 4гб озу.

Вывод: в лабораторной работе были изучены основы работы с системными вызовами и процессами в операционной системе Linux (Ubuntu). Изучили основные концепции системных вызовов. Получили основное понятие о процессе. Изучили базовые системные вызовы POSIX для работы с процессами, такими как: fork(), wait(). Ознакомились с механизмом системных вызовов в языке программирования C.