Титульник

**Оглавление**

[Глава 1. Постановка задачи 3](#_Toc135384518)

[Глава 2. Исследование компилятора gcc, язык ассемблера. Связь процесса и операционной системы. 4](#_Toc135384519)

[2.1. Написание программы на языке C 4](#_Toc135384520)

[2.2. Транслирование написанной программы на язык Assembler 4](#_Toc135384521)

[Глава 3. Makefile, git. Изучение параллельного процесса средствами Linux. Синхронизация доступа к общему ресурсу. 8](#_Toc135384522)

[3.1. Написание программы на С. 8](#_Toc135384523)

[3.1. Makefile, git. 9](#_Toc135384524)

[3.2. Создание параллельного процесса 9](#_Toc135384525)

[Вывод 11](#_Toc135384526)

Глава 1. Постановка задачи

**Цель**: изучить связь процесса компиляции и операционной системы, научиться работать с параллельными процессами средствами Linux.

**Задачи**:

1. Нужно написать программу на С, странслировать его в Assembler с разными опциями оптимизации.
2. Добавить комментарии в сгенерированный ассемберный код
3. Добавить к разработанной программе параллельный процесс средствами Linux и синхронизацию доступа к общему ресурсу.

Глава 2. Исследование компилятора gcc, язык ассемблера. Связь процесса и операционной системы.

2.1. Написание программы на языке C

Для изучения связи процесса компиляции и операционной системы была написана небольшая программа, состоящая из двух функций: fib и main.

В функции fib параметром является номер числа Фибоначчи, в основном коде функции высчитывается число Фибоначчи, оно же и возвращается.

В функции main пользователь вводит номер числа Фибоначчи, далее выводят само число, вызывая функцию fib.

#include <stdio.h>

int fib(int n) {

if (n == 1 || n == 2)

return 1;

return fib(n-1)+fib(n-2);

}

int main() {

int n;

printf("n = ");

scanf("%d", &n);

for (int i = 1; i <= n; i++)

printf("%d ", fib(i));

printf("\n");

return 0;

}

2.2. Транслирование написанной программы на язык Assembler

После написания программы на языке С, нам нужно транслировать ее на язык Assembler, чтобы посмотреть, как ведет себя операционная система при компиляции программы.

Для этого необходимо в командной строке ввести:

gcc -S -o <>.s <>.c -O[0123s]

Здесь –S необходимо для того, чтобы система понимала, что мы транслируем программу на язык Assembler, <>.s необходимо для создания файла с кодом Assembler. Оптимизацию я выбрала -Оs

После выполненной операции получен код:

.file "hello.c" // Название файла на языке С

.text

.globl fib // Объявление имени fib глобальным

.type fib, @function // Определение типа fib

fib: // Функция для числа Фибоначчи

.LFB0:

.cfi\_startproc // Начало функции

pushq %rbp и определение типов

.cfi\_def\_cfa\_offset 16 (функции и параметра n)

.cfi\_offset 6, -16

xorl %ebp, %ebp

pushq %rbx

.cfi\_def\_cfa\_offset 24

.cfi\_offset 3, -24

leal -1(%rdi), %ebx

pushq %rcx

.cfi\_def\_cfa\_offset 32

.L3:

cmpl $1, %ebx // Сравнение 1 и регистра ebx (в ebx находится переменная n)

jbe .L5 // Если 1 <= ebx, тогда идет переход на метку L5

movl %ebx, %edi

subl $2, %ebx // Здесь и выше - условие if (n==1 || n==2)

call fib // Рекурсивный вызов функции fib

addl %eax, %ebp

jmp .L3 // Переходим в начало данной метки (рекурсия)

.L5:

leal 1(%rbp), %eax // В метке L5

popq %rdx находится код

.cfi\_def\_cfa\_offset 24 для исполнения

popq %rbx return 1

.cfi\_def\_cfa\_offset 16

popq %rbp

.cfi\_def\_cfa\_offset 8

ret

.cfi\_endproc

.LFE0:

.size fib, .-fib

.section .rodata.str1.1,"aMS",@progbits,1

.LC0:

.string "n = " // Параметр для метода printf

.LC1:

.string "%d" // Параметр для метода printf

.LC2:

.string "%d " // Параметр для метода scanf

.section .text.startup,"ax",@progbits

.globl main // Обьявление main глобальным именем

.type main, @function // Определение типа main

main:

.LFB1:

.cfi\_startproc // Начало функции

pushq %rbp

.cfi\_def\_cfa\_offset 16

.cfi\_offset 6, -16

leaq .LC0(%rip), %rdi // Определение параметра "n = " для printf

leaq .LC2(%rip), %rbp // Определение параметров ("%d" для scanf

pushq %rbx

.cfi\_def\_cfa\_offset 24

.cfi\_offset 3, -24

movl $1, %ebx

subq $24, %rsp

.cfi\_def\_cfa\_offset 48

movq %fs:40, %rax

movq %rax, 8(%rsp)

xorl %eax, %eax

call printf@PLT // Вызов функции printf

leaq 4(%rsp), %rsi

leaq .LC1(%rip), %rdi // Определение параметра "%d" для printf

xorl %eax, %eax

call \_\_isoc99\_scanf@PLT // Вызов функции scanf

.L9: // Начало цикла

cmpl %ebx, 4(%rsp) // Если i становится равным n,

jl .L13 то совершается переход к метке L13

movl %ebx, %edi

incl %ebx

call fib // Вызов функции fib

movq %rbp, %rdi

movl %eax, %esi

xorl %eax, %eax

call printf@PLT // Вызов функции printf с подсчитанным ранее fib(i)

jmp .L9

.L13: // Выход из цикла

movl $10, %edi

call putchar@PLT

movq 8(%rsp), %rax

subq %fs:40, %rax

je .L11

call \_\_stack\_chk\_fail@PLT

.L11: // Конец функции main

addq $24, %rsp

.cfi\_def\_cfa\_offset 24

xorl %eax, %eax

popq %rbx

.cfi\_def\_cfa\_offset 16

popq %rbp

.cfi\_def\_cfa\_offset 8

ret

.cfi\_endproc

.LFE1:

.size main, .-main

.ident "GCC: (GNU) 12.2.1 20230201"

.section .note.GNU-stack,"",@progbits

Глава 3. Makefile, git. Изучение параллельного процесса средствами Linux. Синхронизация доступа к общему ресурсу.

3.1. Написание программы на С.

Для изучения параллельного процесса и синхронизации доступа к общему ресурсу написана программа на языке С.

В данной программе две функции: fact и main. В функции main выводится факториал от 10 путем вызова функции fact. В функции fact мы считаем факториал от параметра функции – n.

int n = 10;

int main() {

printf("Факториал от %i: %i\n", n, fact(n));

return 0;

}

int fact(int n) {

if (n == 0) return 1;

if (n == 1) return 1;

return fact(n-1)\*n;

}

Далее мы разделяем компиляцию, перемещая функцию fact в отдельную библиотеку fact.h.

#ifndef \_\_FACT\_H\_\_

#define \_\_FACT\_H\_\_

int fact(int n);

#endif

В данной библиотеке хранится только заголовок вызываемой функции, сама функция находится в файле fact.c.

* 1. Makefile, git.

.PHONY: clean run

OBJS=parallel.o fact.o

run: parallel // Запускаем main()

./$<

parallel: $(OBJS) // Создаем запускаемый файл

gcc -o parallel $(OBJS)

%.o: %.c // Создаем объектные файлы

gcc -c -o $@ $<

# parallel.c: fact.h

# fact.c: fact.h

clean: //удаляем изменяемые файлы

rm -f parallel \*.o \*~ \*.a

* 1. Создание параллельного процесса

Параллельный процесс создается с помощью системного вызова fork(). Мы его помещаем в переменную rc, для того, чтобы в дальнейшем понимать, какой процесс у нас происходит. Если rc == 0, значит процесс дочерний, где происходят вычисления. Если rc > 0, то происходит родительский процесс – вывод на экран. Если rc < 0, выводим ошибку.

Т.к. у нас буфер – общий ресурс, необходимо синхронизировать доступ к нему. Доступ к синхронизации ресурса происходит с помощью системного вызова pipe(p), где p[2] – файловый дескриптор. В дочернем процессе мы записываем вычисленное значение в буфер, а в родительском процессе читаем значение в буфере.

Необходимо также не забывать об открытии и закрытии концов в обоих процессах для синхронизации (с помощью close).

Как итог, получаем код файла parallel.c:

#include <stdio.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/wait.h>

#include "fact.h"

#define MAXBUFSIZE 1024

const int STDIO = 1;

int n = 10;

int main() {

char buf[MAXBUFSIZE];

int p[2]; //Файловый дескриптор

pipe(p);

ssize\_t rc = fork();

if (rc == 0) {

l = snprintf(buf, sizeof(buf), "Факториал от %i: %i\n", n, fact(n));

close(p[0]); //Закрываем для синхронизации

write(p[1], buf, MAXBUFSIZE);

close(p[1]);

\_exit(0);

}

else if (rc > 0) {

int crc;

close(p[1]);

wait(&crc);

l = read(p[0], buf, MAXBUFSIZE);

write(STDIO, buf, strlen(buf));

close(p[0]);

}

else {

printf("Error");

}

return 0;

}

Вывод

В данной лабораторной была изучена связь процесса компиляции с операционной системой. Помимо этого получены навыки работы с параллельным процессом и синхронизацией доступа к общему ресурсу.