Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Операционные среды и системное программирование

ОТЧЕТ
к лабораторной работе №4
на тему

УПРАВЛЕНИЕ ПРОЦЕССАМИ И ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ

Студент О. Л. Дайнович
Преподаватель Н. Ю. Гриценко

Минск 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Изучение основных особенностей подсистемы управления процессами и средств взаимодействия процессов в Unix. Практическое проектирование, реализация и отладка программных комплексов из нескольких взаимодействующих процессов.

Написать программу, реализующую процесс, который при получении сигнала, стандартно вызывающего завершение, создает свою копию, которая продолжает выполняться с прерванного места, и лишь после этого завершается, избегая таким образом безусловного «уничтожения» неперехватываемым сигналом.

Примечание: в Unix-системах, в отличие от Windows, реализация такой возможности достигается гораздо легче и естественнее.

В качестве демонстрации «живости» процесса и его выполнения можно использовать произвольные действия, повторяющиеся периодически и дающие заметный результат. Для консольных приложений это может быть, например, счетчик, значение которого обновляется с заданной частотой и записывается в файл.

2 ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Сигналы являются программными прерываниями, которые посылаются процессу, когда случается некоторое событие. Сигналы могут возникать синхронно с ошибкой в приложении, например SIGFPE (ошибка вычислений с плавающей запятой) и SIGSEGV (ошибка адресации), но большинство сигналов является асинхронными. Сигналы могут посылаться процессу, если система обнаруживает программное событие, например, когда пользователь дает команду прервать или остановить выполнение, или получен сигнал на завершение от другого процесса. Сигналы могут прийти непосредственно от ядра ОС, когда возникает сбой аппаратных средств ЭВМ. Система определяет набор сигналов, которые могут быть отправлены процессу. В Linux применяется около 30 различных сигналов. При этом каждый сигнал имеет целочисленное значение и приводит к строго определенным действиям. [1]

Механизм передачи сигналов состоит из следующих частей:

– установление и обозначение сигналов в форме целочисленных значений;

– маркер в строке таблицы процессов для прибывших сигналов;

– таблица с адресами функций, которые определяют реакцию на прибывающие сигналы.

Отдельные сигналы подразделяются на три класса:

– системные сигналы (ошибка аппаратуры, системная ошибка и т.д.);

– сигналы от устройств;

– сигналы, определенные пользователем.

Как только сигнал приходит, он отмечается записью в таблице процессов. Если этот сигнал предназначен для процесса, то по таблице указателей функций в структуре описания процесса выясняется, как нужно реагировать на этот сигнал. При этом номер сигнала служит индексом таблицы.

Известно три варианта реакции на сигналы:

– вызов собственной функции обработки;

– игнорирование сигнала (не работает для SIGKILL);

– использование предварительно установленной функции обработки по умолчанию. [2]

Чтобы реагировать на разные сигналы, необходимо знать концепции их обработки. Процесс должен организовать так называемый обработчик сигнала в случае его прихода. Для этого используется функция signal(). [3]

3 ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ

В результате лабораторной работы был написан скрипт, реализующий инверсию порядка символов в каждой строке потока.

Основной код написан на языке C. Bash скрипт собирает программу, состоящую из файла main.cpp, с помощью файла Makefile и выполняет код.

Программа запускает процесс, который выводит в консоль счетчик (рисунок 1).

```
oleg@oleg-pc:~/study/SystemProg/lab4$ bash lab4.sh
g++ main.cpp -o main.o
./main.o
Running... Counter: 0
Running... Counter: 1
Running... Counter: 2
Running... Counter: 3
Running... Counter: 4
```

Рисунок 1 – Вывод счетчика в консоль

При получении стандартного сигнала о прерывании из консоли, процесс создает дочерний процесс, который продолжает вывод счетчика в консоль, а родительский процесс завершается. Такое действие можно делать неограниченное количество раз (рисунок 2).

```
Running... Counter: 4
Running... Counter: 5
Running... Counter: 6
^C
=========================
Child process created. Counter: 7
Parent process stopped. Counter: 7
=========================
Running... Counter: 7
Running... Counter: 8
Running... Counter: 9
^C
=========================
Child process created. Counter: 10
Parent process stopped. Counter: 10
=========================
Running... Counter: 10
Running... Counter: 11
Running... Counter: 12
```

Рисунок 2 – Вызов прерывания и создание дочернего процесса

ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы были изучены основные особенности подсистемы управления процессами и средства взаимодействия процессов в Unix. Практическое проектирование, реализация и отладка программных комплексов из нескольких взаимодействующих процессов. Также была разработана программа, реализующая процесс, который при получении сигнала, стандартно вызывающего завершение, создает свою копию, которая продолжает выполняться с прерванного места, и лишь после этого завершается. избегая таким образом безусловного «уничтожения» неперехватываемым сигналом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Понятие о сигналах [Электронный ресурс]. – Режим доступа: https://www.opennet.ru/docs/RUS/linux_parallel/node10.html – Дата доступа: 11.03.2024.

[2] Сигналы [Электронный ресурс]. – Режим доступа: https:// www.kryukov.biz/soderzhanie/protsessy/signaly/ – Дата доступа: 11.03.2024.

[3] Практика работы с сигналами [Электронный ресурс]. – Режим доступа: https://habr.com/ru/articles/141206/ – Дата доступа: 11.03.2024.

ПРИЛОЖЕНИЕ А
(обязательное)
Листинг кода

Листинг 1 – Файл lab4.sh

```
#!/usr/bin/bash
make
./build/main
```

Листинг 2 – Файл makefile

```
all: compile run clean

compile:
	g++ main.cpp -o main.o

run: main.o
	./main.o

clean:
	rm -rf *.o
```

Листинг 3 – Файл main.cpp

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

int counter = 0;
volatile sig_atomic_t stop_flag = 0;

void signal_handler(int signum) {
    stop_flag = 1;

    if (stop_flag) {

      pid_t child_pid = fork();

      if (child_pid > 0) {
          printf("\n=========================\n");
          printf("Child process created. Counter: %d\n", counter);

          wait(NULL);
      }

      else if (child_pid == 0) {
          printf("Parent process stopped. Counter: %d\n", counter);
          printf("=========================\n");
      }

      else if (child_pid < 0) {
          fprintf(stderr, "Error: Fork failed.\n");
          exit(1);
```

```
      }

      stop_flag = 0;
    }
}

int main() {
    signal(SIGINT, signal_handler);

    while (counter < 30) {
      printf("Running... Counter: %d\n", counter);
      counter++;

      sleep(1);
    }

    return 0;
}
```