Московский авиационный институт
(национальный исследовательский университет)
Институт № 8 «Информационные технологии и прикладная математика»

Лабораторная работа №4
по курсу «Операционные системы»

Студент: Минеева Светлана Алексеевна

Группа: М8О-210Б-21

Преподаватель: Миронов Е.С

Вариант №14
Оценка: ____
Дата: 21.11.2022
Подпись: _____

Москва, 2022

Содержание:

1. Цель работы

Приобретение практических навыков в:

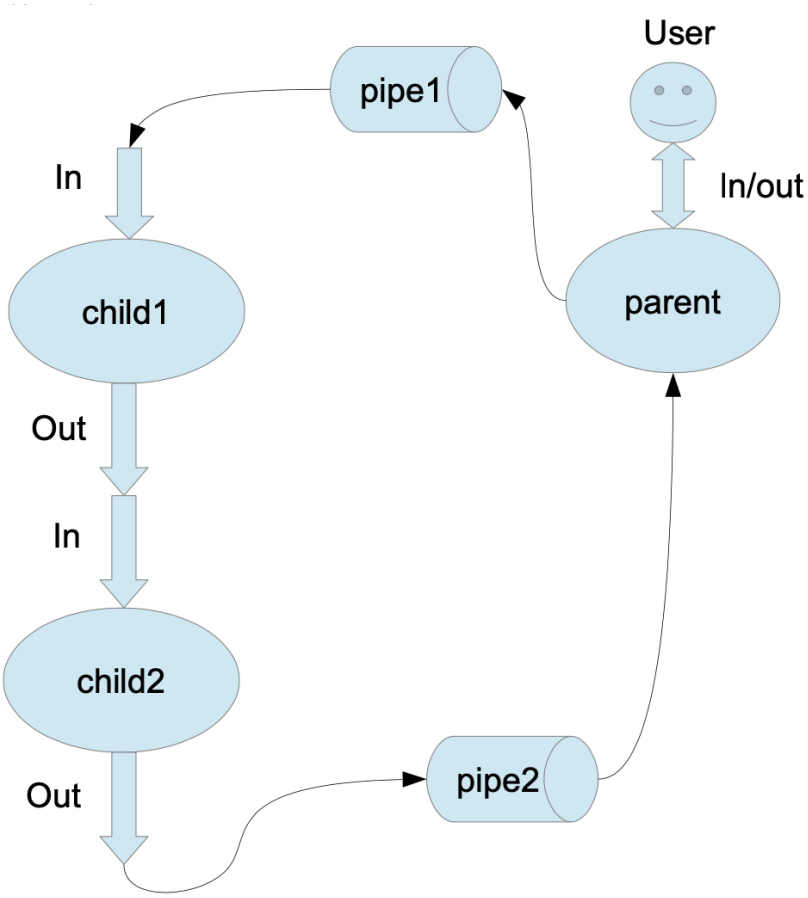
- Освоение принципов работы с файловыми системами
- Обеспечение обмена данных между процессами посредством технологии «File mapping»

2. Задание

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в одной из двух операционных систем. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или через отображаемые файлы (memory-mapped files).

Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

3. Вариант задания



Родительский процесс создает два дочерних процесса. Перенаправление стандартных потоков ввода-вывода показано на картинке выше. Child1 и Child2 можно «соединить» между собой дополнительным каналом. Родительский и дочерний процесс должны быть представлены разными программами.

Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1. Процесс child1 и child2 производят работу над строками. Child2 пересылает результат своей работы родительскому процессу. Родительский процесс полученный результат выводит в стандартный поток вывода.

Вариант №14: Child1 переводит строки в нижний регистр. Child2 убирает все задвоенные пробелы.

4. Общие сведения о программе

Программа состоит из файлов parent.c, child1.c, child2.c и structure.h. В файле parent.c хранится родительский процесс, создание дочерних процессов. В child1.c строки переводятся в нижний регистр. В child2.c убираются все двойные пробелы. Ключевым файлом является structure.h, именно в нем происходят основные задачи программы – функции для работы с мьютексами.

Основные системные вызовы и функции в программе:

1. **fork –** для создания дочернего процесса;
2. **execv** - для замены образа памяти процесса;
3. **open** – для создания файла и его открытия;
4. **close** – для закрытия файлового дескриптора;
5. **mmap** – для отображения файла в память;
6. **ftruncate** – для обрезки файла до заданного размера;
7. **shm_unlink –** для удаления именованного семафора;
8. **cond_wait** – для блокировки семафора;
9. **pthread_mutex_unlock** – для разблокировки семафора;
10. **pthread_mutex_init** – для инициализации семафора;
11. **pthread_mutex_destroy** – для уничтожения семафора.

5. Общий метод и алгоритм решения

Данная лабораторная работа заключается в предотвращении нахождения конкурирующих потоков процессов в критической секции кода

одновременно. Один из методов решения этой проблемы при помощи библиотеки pthread – это мьютексы. Мьютекс – это один из вариантов семафорных механизмов для организации взаимного исключения. У мьютекса есть два состояния: он свободен или заблокирован. Когда поток начинает работать с мьютексом, то он заблокируется. Остальные потоки после этого не имеют доступа к критической секции кода, они ждут разблокировки мьютекса. Разблокировка мьютекса происходит только тем потоком, который с ним в данный момент работает. Освобождение заблокированного мьютекса обычно происходит при выходе потока из критической секции кода. В конце программы мьютекс необходимо уничтожить.

Исполнение защищённого участка кода при использовании мьютекса происходит последовательно потоками, а не параллельно. Порядок доступа не определен. Мьютекс создается один на всех, чтобы ограничить доступ к критической секции кода больше одного потока одновременно.

6. Текст программы

structure.h

```
#include <stdio.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>
#include <sys/stat.h>
#include <pthread.h>
#include <stdlib.h>

const int structure_block_size = 4096;

typedef struct structure {
  char memory_name[20];
  int file_descriptor;
  char *buffer;
} structure;

typedef struct state {
  pthread_mutex_t mutex;
  pthread_mutex_t write_mutex;
  pthread_cond_t condition;
  int memory_length;
} state;

int structure_create(structure *memory_structure, char *memory_name, char host) {
  memory_structure->file_descriptor = shm_open(memory_name, O_CREAT | O_RDWR, S_IWUSR
| S_IRUSR);
```

```
  if(memory_structure->file_descriptor == -1) {
    return -1;
  }

  if(host) {
    if(ftruncate(memory_structure->file_descriptor, structure_block_size +
sizeof(state))) {
      return -1;
    }
  }

  memory_structure->buffer = mmap(
    NULL,
    structure_block_size + sizeof(state),
    PROT_READ | PROT_WRITE,
    MAP_SHARED,
    memory_structure->file_descriptor,
    0
  );

  if(memory_structure->buffer == (void*) -1) {
    return -1;
  }

  if(host) {
    pthread_mutexattr_t attribute_mutex;
    pthread_condattr_t attribute_condition;

    if(pthread_mutexattr_init(&attribute_mutex) ||
      pthread_mutexattr_setpshared(&attribute_mutex, PTHREAD_PROCESS_SHARED) ||
      pthread_mutex_init(&(((state*)(memory_structure->buffer))->mutex),
&attribute_mutex) ||
      pthread_mutex_init(&(((state*)(memory_structure->buffer))->write_mutex),
&attribute_mutex) ||
      pthread_mutex_lock(&(((state*)(memory_structure->buffer))->write_mutex)) ||
      pthread_condattr_init(&attribute_condition) ||
      pthread_condattr_setpshared(&attribute_condition, PTHREAD_PROCESS_SHARED) ||
      pthread_cond_init(&(((state*)(memory_structure->buffer))->condition),
&attribute_condition)) {
      return -1;
    }
  }

  memcpy(memory_structure->memory_name, memory_name, strlen(memory_name) + 1);
  return 0;
}

void structure_destroy(structure *memory_structure) {
  pthread_mutex_destroy(&(((state*)(memory_structure->buffer))->mutex));
  pthread_mutex_destroy(&(((state*)(memory_structure->buffer))->write_mutex));
  pthread_cond_destroy(&(((state*)(memory_structure->buffer))->condition));
  munmap(memory_structure->buffer, structure_block_size);
```

```
  shm_unlink(memory_structure->memory_name);
  close(memory_structure->file_descriptor);
  memory_structure->file_descriptor = -1;
}

void structure_write(structure *memory_structure, char *memory_string, int
memory_length1) {
  pthread_mutex_lock(&(((state*)(memory_structure->buffer))->write_mutex));
  pthread_mutex_lock(&(((state*)(memory_structure->buffer))->mutex));
  ((state*)(memory_structure->buffer))->memory_length = memory_length1;
  memcpy(memory_structure->buffer + sizeof(state), memory_string, memory_length1);
  pthread_cond_broadcast(&(((state*)(memory_structure->buffer))->condition));
  pthread_mutex_unlock(&(((state*)(memory_structure->buffer))->mutex));
}

char* structure_read(structure *memory_structure, int *length) {
  pthread_mutex_lock(&(((state*)(memory_structure->buffer))->mutex));
  pthread_mutex_unlock(&(((state*)(memory_structure->buffer))->write_mutex));
  pthread_cond_wait(&(((state*)(memory_structure->buffer))->condition),
&(((state*)(memory_structure->buffer))->mutex));
  int memory_length1 = (((state*)(memory_structure->buffer))->memory_length);
  *length = memory_length1;
  char* memory = malloc(memory_length1);
  memcpy(memory, memory_structure->buffer + sizeof(state), memory_length1);
  pthread_mutex_unlock(&(((state*)(memory_structure->buffer))->mutex));
  return memory;
}
```

parent.c

```
#include "unistd.h"
#include "stdio.h"
#include <fcntl.h>
#include <string.h>
#include "structure.h"

int create_child(char* file_name) {
  switch(fork()) {
    case -1: {
      return -1;
    }

    case 0:{
      char* args[] = {NULL};
      execv(file_name, args);
      return -1;
    }

    default: {
      break;
    }
  }
```

```
  return 0;
}

int main() {
  structure parent_child1;
  structure child1_child2;
  structure child2_parent;
  int number_string, count_string = 0;

  if(
    structure_create(&parent_child1, "parent_child1", 1) ||
    structure_create(&child1_child2, "child1_child2", 1) ||
    structure_create(&child2_parent, "child2_parent", 1)
  ) {
    printf("error: cannot create shared memory\n");
    return 1;
  }

  create_child("./child1");
  create_child("./child2");
  printf("Number of string:\n");
  scanf("%d", &number_string);
  int space;
  while((space = getchar()) != '\n' && number_string != EOF);
  printf("Enter string:\n");
  char buffer[256];

  while(1){
    fgets(buffer, 255, stdin);
    int string_length = strlen(buffer);
    structure_write(&parent_child1, buffer, string_length + 1);
    char *input = structure_read(&child2_parent, &string_length);
    printf("%s", input);
    free(input);
    fflush(stdout);
    count_string += 1;

    if (count_string == number_string) {
      structure_destroy(&parent_child1);
      structure_destroy(&child1_child2);
      structure_destroy(&child2_parent);
      break;
    }
  }

  return 0;
}
```

child1.c

```
#include "unistd.h"
```

```
#include "stdio.h"
#include <string.h>
#include <ctype.h>
#include "structure.h"

void to_lower(char* string) {
  int string_length = strlen(string);

  for(int number = 0; number < string_length; number++) {
    string[number] = tolower(string[number]);
  }
}

int main() {
  structure parent_child1;
  structure child1_child2;

  if(structure_create(&parent_child1, "parent_child1", 0) ||
structure_create(&child1_child2, "child1_child2", 0)) {
    printf("error: cannot connect to shared memory\n");
    return 1;
  }

  while(1) {
    int input_length;
    char* input = structure_read(&parent_child1, &input_length);
    to_lower(input);
    structure_write(&child1_child2, input, strlen(input) + 1);
    free(input);
  }

  return 0;
}
```

child2.c

```
#include "unistd.h"
#include "stdio.h"
#include <string.h>
#include "structure.h"

void replace_spaces(char* string, int* length) {
  for(int external_index = 0; external_index  < *length - 1; external_index ++) {
    if(string[external_index] == ' ') {
      while(string[external_index + 1] == ' ') {
        for(int internal_index = external_index + 1; internal_index < *length - 1;
internal_index++) {
          string[internal_index] = string[internal_index + 1];
        }
        *length -= 1;
      }
    }
```

```
  }
}

int main() {
  structure child1_child2;
  structure child2_parent;

  if(structure_create(&child1_child2, "child1_child2", 0) ||
structure_create(&child2_parent, "child2_parent", 0)) {
    printf("error: cannot connect to shared memory\n");
    return 1;
  }

  while(1) {
    int input_length;
    char* input = structure_read(&child1_child2, &input_length);
    replace_spaces(input, &input_length);
    structure_write(&child2_parent, input, strlen(input) + 1);
    free(input);
  }

  return 0;
}
```

7. Демонстрация работы программы

Last login: Sun Nov 20 15:21:33 on ttys000

The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
MacBook-Pro-MacBook:~ macbookpro$ ./parent
Number of string:
7
Enter string:
TT JJ
tt jj
Hjk hhj HHH lkhL
hjk hhj hhh lkhl
hh   nn
hh nn
56 %6GHb
56 %6ghb
*   JHHdjd   HIUG*
jhhdjd hiug
JJ   JJ   JJ   JJ
jj jj jj jj
4564HJKLJI

4564hjklji
MacBook-Pro-MacBook:~ macbookpro$

8. Вывод

Я составила и отладила программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними. В процессе выполнения данной лабораторной работы я освоила принципы работы с файловыми системами и получила практические навыки в обеспечении обмена данных между процессами посредством технологии “File mapping”. Я научилась синхронизировать работу процессов и потоков с помощью семафоров. Мною были освоены разнообразные функции для работы с ними, такие как fork(), execv(), mmap(), shm_unlink(), cond_wait() и многие другие. Сравнивая работу различных реализаций взаимодействия между процессами, а именно реализацию в данной лабораторной работе и в лабораторной работе №2, я отметила, что взаимодействие посредством «file mapping» эффективнее за счёт отсутствия постоянных вызовов read, write и требует меньше памяти.