Московский авиационный институт
(Национальный исследовательский университет)

Институт: №8 «Информационные технологии и прикладная математика»
Кафедра: 806 «Вычислительная математика и программирование»
Дисциплина: «Операционные системы»

Лабораторная работа № 4

Тема: Файловые системы

Студент: Бирюков В. В.
Группа: М80-207Б-19
Преподаватель: Миронов Е. С.
Дата:
Оценка:

Москва, 2020

Постановка задачи

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.

Вариант 20.

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами.

Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в pipe1 или в pipe2 в зависимости от правила фильтрации. Процесс child1 и child2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: строки длины больше 10 символов отправляются в pipe2, иначе в pipe1. Дочерние процессы инвертируют строки.

Алгоритм решения задачи

Обмениваться данными с процессами будем через отображенные в память файлы. Для этого родительский процесс создает два файла, и устанавливает их длину равной размеру страницы. Также родительский процесс считывает имена двух файлов для записи результата. Дочерние процессы переопределяют свой вывод соответствующим файлом и вызывают exec, передав в программу имя отображаемого файла.

В это время родительский процесс отображает оба файла в память. Файлам был задан размер, равный размеру страницы, так как для получившегося массива выделяется память кратная этому размеру. Затем процесс начинает считывать строки. Передача строк в дочерние процессы происходит по описанному во 2 лабораторной алгоритму, используя буфер для 10 символов. Однако теперь запись происходит в соответствующий массив. Для синхронизации процессов используются сигналы. После того как вся строка записана, родительский процесс посылает в соответствующий дочерний сигнал SIGUSR1 и ждет в ответ такой же сигнал, чтобы продолжить чтение следующей строки. Если в массиве закончилось место, родительский процесс также отправляет и ждет сигнал, а затем продолжает запись строки с начала массива.

Дочерний процесс открывает файл для отображения и отображает его в память. Затем ждет сигнал, считывает строку из массива и посылает сигнал родительскому процессу. Номер родительского процесса также передается в дочерний как аргумент. Об окончании строки сигнализирует символ перевода строки, об окончании строк вообще сигнализирует нулевой символ.

Листинг программы

```
// main.c
// основной процесс
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "sys/mman.h"
#include "string.h"
#include "signal.h"
#include "fcntl.h"

#define check(VALUE, MSG, BADVAL) if (VALUE == BADVAL) { perror(MSG);
                                                         exit(1); }

int main() {
  size_t pagesize = sysconf(_SC_PAGESIZE);

  int pid = getpid();

  char fn1[256];
  char fn2[256];
  if (scanf("%s", fn1) <= 0) {
    perror("scanf error");
    return -1;
  }
  if (scanf("%s", fn2) <= 0) {
    perror("scanf error");
    return -1;
  }
  getchar();

  FILE* file1 = fopen(fn1, "wt");
  check(file1, "fopen 1 error", NULL)
  FILE* file2 = fopen(fn2, "wt");
  check(file2, "fopen 2 error", NULL)

  char mfilename1[] = "mmap1";
  char mfilename2[] = "mmap2";
  int mfile1 = open(mfilename1, O_RDWR | O_CREAT);
  check(mfile1, "open 1 error", 0)
  int mfile2 = open(mfilename2, O_RDWR | O_CREAT);
  check(mfile2, "open 2 error", 0)
  size_t i1 = 0, i2 = 0;

  ftruncate(mfile1, pagesize);
  ftruncate(mfile2, pagesize);

  int id1 = fork();
```

```
if (id1 == -1)  {
  perror("fork 1 error");
  return -1;

} else if (id1 == 0) {
  fclose(file2);
  close(mfile2);
  check(dup2(fileno(file1), fileno(stdout)), "dup2 error", -1)
  fclose(file1);

  char spid[10];
  snprintf(spid, 10, "%d", pid);
  char* const args[] = {"child", mfilename1, spid, (char *)NULL};
  check(execv("child", args), "execv child 1 error", -1)

} else {
  int id2 = fork();

  if (id2 == -1)  {
    perror("fork 2 error");
    return -1;

  } else if (id2 == 0) {
    fclose(file1);
    close(mfile1);
    check(dup2(fileno(file2), fileno(stdout)), "dup2 error", -1)
    fclose(file2);

    char spid[10];
    snprintf(spid, 10, "%d", pid);
    char* const args[] = {"child", mfilename2, spid, (char *)NULL};
    check(execv("child", args), "execv child 2 error", -1)

  } else {
    fclose(file1);
    fclose(file2);

    char *fmap1 = (char *)mmap(NULL, pagesize, PROT_WRITE | PROT_READ,
                                               MAP_SHARED, mfile1, 0);
    check(fmap1, "mmap 1 error", MAP_FAILED)
    char *fmap2 = (char *)mmap(NULL, pagesize, PROT_WRITE | PROT_READ,
                                               MAP_SHARED, mfile2, 0);
    check(fmap2, "mmap 2 error", MAP_FAILED)

    sigset_t set;
    check(sigemptyset(&set), "sigemptyset error", -1)
    check(sigaddset(&set, SIGUSR1), "sigaddset error", -1)
    check(sigprocmask(SIG_BLOCK, &set, NULL), "sigprocmask error", -1)
    int sig;

    char c;
    char str[10];
    str[0] = '\0';
    int n = 0;
    while (scanf("%c", &c) > 0) {
      if (c != '\n') {
        if (n < 10) {
          str[n] = c;
```

```
  } else if (str[0] != '\0') {
    for (int i = 0; i < 10; ++i) {
      // write(fd2[1], &str[i], sizeof(char));
      fmap2[i2] = str[i];
      if (++i2 == pagesize) {
        i2 = 0;
        check(kill(id2, SIGUSR1), "send signal to child 2 error",
                                                              -1)
        check(sigwait(&set, &sig), "sigwait error", -1)
      }
      str[i] = '\0';
    }
    // write(fd2[1], &c, sizeof(char));
    fmap2[i2] = c;
    if (++i2 == pagesize) {
      i2 = 0;
      check(kill(id2, SIGUSR1), "send signal to child 2 error",
                                                              -1)
      check(sigwait(&set, &sig), "sigwait error", -1)
    }
  } else {
    // write(fd2[1], &c, sizeof(char));
    fmap2[i2] = c;
    if (++i2 == pagesize) {
      i2 = 0;
      check(kill(id2, SIGUSR1), "send signal to child 2 error",
                                                              -1)
      check(sigwait(&set, &sig), "sigwait error", -1)
    }
  }
  ++n;
} else {
  if (str[0] != '\0') {
    for (int i = 0; i < n; ++i) {
      // write(fd1[1], &str[i], sizeof(char));
      fmap1[i1] = str[i];
      if (++i1 == pagesize) {
        i1 = 0;
        check(kill(id1, SIGUSR1), "send signal to child 1 error",
                                                              -1)
        check(sigwait(&set, &sig), "sigwait error", -1)
      }
      str[i] = '\0';
    }
    // write(fd1[1], &c, sizeof(char));
    fmap1[i1] = c;
    i1 = 0;
    check(kill(id1, SIGUSR1), "send signal to child 1 error", -1)
    check(sigwait(&set, &sig), "sigwait error", -1)
  } else {
    // write(fd2[1], &c, sizeof(char));
    fmap2[i2] = c;
    i2 = 0;
    check(kill(id2, SIGUSR1), "send signal to child 2 error", -1)
    check(sigwait(&set, &sig), "sigwait error", -1)
  }
  n = 0;
}
```

```
      }
      c = '\0';
      fmap1[0] = c;
      fmap2[0] = c;
      check(kill(id1, SIGUSR1), "send signal to child 1 error", -1)
      check(kill(id2, SIGUSR1), "send signal to child 2 error", -1)
      check(munmap(fmap1, pagesize), "munmap error", -1);
      check(munmap(fmap2, pagesize), "munmap error", -1);
      close(mfile1);
      close(mfile2);
    }
  }
  return 0;
}

// child.c
// дочерний процесс
#include "unistd.h"
#include "stdio.h"
#include "stdlib.h"
#include "sys/mman.h"
#include "string.h"
#include "signal.h"
#include "fcntl.h"

#define check(VALUE, MSG, BADVAL) if (VALUE == BADVAL) { perror(MSG);
                                                         exit(-1); }

char * add(char *str, int cap, int n, char c) {
  if (n == cap) {
    cap *= 2;
    str = (char *)realloc(str, sizeof(char) * cap);
    check(str, "realloc error", NULL)
  }
  str[n] = c;
  return str;
}

int main(int agrc, char const *argv[]) {
  size_t pagesize = sysconf(_SC_PAGESIZE);
  char c = '\0';
  int n = 0;
  int cap = 256;
  char* str = (char *)malloc(sizeof(char) * cap);
  check(str, "malloc error", NULL)

  int mfile = open(argv[1], O_RDWR);
  check(mfile, "open error", -1)
  int pid = atoi(argv[2]);

  char* fmap = (char *)mmap(NULL, pagesize, PROT_READ, MAP_SHARED,
                                                       mfile, 0);
  check(fmap, "mmap error", MAP_FAILED)
  size_t i = 0;

  sigset_t set;
  check(sigemptyset(&set), "sigemptyset error", -1)
  check(sigaddset(&set, SIGUSR1), "sigaddset error", -1)
```

```
  check(sigprocmask(SIG_BLOCK, &set, NULL), "sigprocmask error", -1)
  int sig;

  for(;;) {
    check(sigwait(&set, &sig), "sigwait error", -1);
    for (i = 0; i < pagesize; ++i) {
      c = fmap[i];
      if (c != '\n' && c != '\0') {
        str = add(str, cap, n, c);
        ++n;
      } else if (c == '\0') {
        break;
      } else {
        for (int i = n-1; i >= 0; i--) {
          printf("%c", str[i]);
        }
        printf("%c", c);
        n = 0;
        break;
      }
    }
    if (c == '\0') {
      break;
    } else {
      check(kill(pid, SIGUSR1), "send signal to parent error", -1)
    }
  }
  free(str);
  check(munmap(fmap, pagesize), "munmap error", -1)
  close(mfile);
}
```

Тесты и протокол исполнения

```
./main
1.txt
2.txt
12345678900
12345
abcdefghijklmno

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
sssssssssssssssssssssssssssssss
sss

// 1.txt
54321

sss

// 2.txt
00987654321
onmlkjihgfedcba
```

sssssssssssssssssssssssssssssssscccccccccccccccccccccccccccccccc
cccccccccccccccccccccccccccccccccccbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Выводы

В ходе лабораторной работы я познакомился с отображенными в память файлами, как со способом взаимодействия процессов и с управлением системными сигналами. Отображение файлов в память позволяет удобнее работать с информацией в файле или модифицировать один файл из нескольких процессов. Также отображение в память используется для загрузки динамических библиотек. Для целей межпроцессорного взаимодействия можно использовать как обычные файлы, так и объекты разделяемой памяти. Я решил использовать обычные файлы фиксированного размера. Синхронизация процессов при помощи сигналов показалась мне удобнее, чем, например, при помощи мьютекса в разделяемой памяти. Однако такой способ менее безопасен, так как сигнал процессу может послать кто угодно.

Список литературы

1. Memory Mapped Files – *Beej's Guide to Unix IPC.*

   URL: http://beej.us/guide/bgipc/html/multi/mmap.html

2. Таненбаум Э., Бос Х. *Современные операционные системы. – 4-е изд.* – СПб.: Издательский дом «Питер», 2018.