Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу

«Операционные системы»

Группа: М80 – 206Б-18
Студент: Черненко И.Д.
Преподаватель: Соколов А.А.
Оценка: _____
Дата: _____

Москва, 2019.

Содержание

Постановка задачи

Дочерний процесс представляет собой сервер по работе с деревом общего вида и принимает команды со стороны родительского процесса.

Общие сведения о программе

Программа компилируется из двух файлов main.c и c_queue.c. Также используется заголовочные файлы: stdio.h, stdbool.h, stdlib.h, sys/types.h, unistd.h, c_queue.h. В программе используются следующие системные вызовы:

1. **read** – для чтения данных из файла
2. **write** – для записи данных в файл
3. **pipe** – для создания однонаправленного канала, через который могут общаться два процесса. Системный вызов возвращает два дескриптора файлов. Один для чтения из канала, другой для записи в канал.
4. **fork** – для создания дочернего процесса.

Общий метод и алгоритм решения.

Для реализации поставленной задачи необходимо:

1. Используя системный вызов pipe создать канал, по которому будут обмениваться данными два процесса.
2. Используя системный вызов fork создать дочерний процесс.
3. В родительском процессе считывать данные со стандартного потока и правильно распарсить подаваемую команду.
4. Как в родительском процессе данные считались, необходимо записать их в канал с помощью системного вызова write.
5. Как только родительский процесс записал данные в канал дочерний процесс считывает их, производит вычисления с деревом общего вида.
6. Дочерний процесс выводит результат используя write.

Основные файлы программы

main.c:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include "c_queue.h"
typedef struct ans ans;
struct ans {
   int cmd;
   int val;
   char path[32];
};
typedef struct node node,*pnode;
struct node {
   pnode s;
   pnode b;
   int val;
   bool is_root;
};
pnode node_create(int val) {
   pnode new_node = (pnode)malloc(sizeof(node));
   if (new_node) {
      new_node -> val = val;
      new_node -> s = NULL;
      new_node -> b = NULL;
      new_node -> is_root = false;
   }
   return new_node;
}
pnode* search(pnode* t, queue *path) {
```

```
    if (!(*t) && !q_is_empty(path)) {
        return NULL;
    }
    if (!q_is_empty(path)) {
        char c = q_front(path);
        pop(path);
        if (c == 's') {
            return search(&(*t) -> s, path);
        } else if (c == 'b') {
            return search(&(*t) -> b, path);
        }
        return NULL;
    }
    return t;
}
bool add(pnode* t, int val, queue *path) {
    if (!(*t) && q_is_empty(path)) {
        (*t) = node_create(val);
        return true;
    }
    pnode* pr = search(t, path);
    if (!pr) {
        return false;
    }
    pnode new_node = node_create(val);
    if (!new_node) {
        return false;
    }
    new_node -> b = (*pr);
    (*pr) = new_node;
    return true;
}
void rmv(pnode* t) {
    while((*t) -> s != NULL){
        rmv(&((*t) -> s));
    }
```

```
    pnode tmp = *t;
    *t = (*t) -> b;
    free(tmp);
}
bool valid_numb(char* numb) {
    if (numb == NULL) {
        return false;
    }
    bool flag = true;
    int i = 0;
    if (numb[i] != '-' && !(numb[i] >= '0' && numb[i] <= '9')) {
        flag = false;
    }
    i++;
    while (i < 11) {
        if (numb[i] == '\0') {
            break;
        }
        if (!(numb[i] >= '0' && numb[i] <= '9')) {
            flag = false;
            break;
        }
        i++;
    }
    return flag;
}
bool valid_path(char* path) {
    if (path == NULL) {
        return false;
    }
    if (path[0] == '@' && path[1] == '\0') {
        return true;
    }
    for (int i = 0; i < 32; i++) {
        if (path[i] == '\0') {
            break;
```

```
        } else if (path[i] != 's' && path[i] != 'b') {
            return false;
        }
    }
    return true;
}
ans* parser(char* cmd) {
    ans* parsed = (ans*)malloc(sizeof(ans));
    char* pch = strtok(cmd," \n");
    while (pch != NULL) {
        if (strcmp(pch, "prt") == 0) {
            parsed->cmd = 0;
            break;
        } else if (strcmp(pch, "rmv") == 0) {
            pch = strtok(NULL, " \n");
            if (valid_path(pch)) {
                parsed->cmd = 1;
                strcpy(parsed->path, pch);
                if (parsed->path[0] == 'b') {
                    parsed->cmd = -1;
                }
                break;
            } else {
                parsed->cmd = -1;
                break;
            }
        } else if (strcmp(pch, "add") == 0) {
            pch = strtok(NULL, " \n");
            if (valid_path(pch)) {
                strcpy(parsed->path, pch);
                pch = strtok(NULL, " \n");
                if (parsed->path[0] == 'b') {
                    parsed->cmd = -1;
                    break;
                }
                if (valid_numb(pch)) {
```

```
                parsed->cmd = 2;
                parsed->val = atoi(pch);
                break;
            } else {
                parsed->cmd = -2;
                break;
            }
        } else {
            parsed->cmd = -1;
            break;
        }
    } else if (strcmp(pch, "ext") == 0) {
        parsed->cmd = 3;
        break;
    } else {
        parsed->cmd = -777;
        break;
    }
  }
  return parsed;
}
void tree_print(pnode t, int depth) {
  if (t) {
    for (int i = 0; i < depth; i++) {
        write(1, "\t", 1);
    }
    char numb[11] = {'\0'};
    sprintf(numb, "%d", t->val);
    int i = 0;
    while (numb[i] != '\0') {
        i++;
    }
    write(1, numb, i);
    write(1,"\n", 1);
    tree_print(t -> s, depth + 1);
    tree_print(t -> b, depth);
```

```
    }
}
int main() {
    setvbuf(stdout, (char *) NULL, _IONBF, 0);
    pnode test = NULL;
    char cmd[100] = {'\0'};
    ans *parsed = (ans *) malloc(sizeof(ans));
    int fd1[2];
    pid_t pr = -1;
    if (pipe(fd1) == -1) {
        perror("pipe\n");
        exit(1);
    }
    pr = fork();
    if (pr < 0) {
        write(1, "Can't create process\n", 22);
    } else if (pr > 0) {
        close(fd1[0]);
        while (read(0, cmd, 100)) {
            parsed = parser(cmd);
            write(fd1[1], &parsed->cmd, 4);
            write(fd1[1], &parsed->val, 4);
            write(fd1[1], parsed->path, 32);
            if (parsed->cmd == 3) {
                return 0;
            }
            for (int i = 0; i < 100; i++) {
                cmd[i] = '\0';
            }
        }
    } else {
        while (1) {
            close(fd1[1]);
            read(fd1[0], &parsed->cmd, 4);
            read(fd1[0], &parsed->val, 4);
            read(fd1[0], parsed->path, 32);
```

```
queue *q = q_create();
int k = 0;
while (parsed->path[k] != '\0') {
    push(q, parsed->path[k]);
    k++;
}
if (q_size(q) == 0) {
    push(q, '\0');
}
if (parsed->cmd == 3) {
    return 0;
} else if (parsed->cmd == 2) {
    if (test == NULL) {
        while (q_size(q) != 0) {
            pop(q);
        }
        test = node_create(parsed->val);
        test->is_root = true;
    } else {
        add(&test, parsed->val, q);
    }
} else if (parsed->cmd == 1) {
    pnode* f = search(&test, q);
    if (test == NULL) {
        write(1, "empty tree\n", 11);
    } else if ((*f) == NULL) {
            write(1, "its root\n", 9);
            rmv(&test);
    } else {
            rmv(f);
    }
} else if (parsed->cmd == 0) {
    if (test == NULL) {
        write(1, "empty tree\n", 11);
    } else {
        tree_print(test, 0);
```

```
            }
        } else if (parsed->cmd == -2){
            write(1, "invalid value\n", 14);
        } else if (parsed->cmd == -1) {
            write(1, "invalid path\n", 13);
        } else if (parsed->cmd == -777) {
            write(1, "invalid command\n", 16);
        }
        q_destroy(q);
    }
  }
  return 0;
}
```

c_queue.c:

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include "c_queue.h"
#define MIN_CAP 32

queue *q_create(){
  queue *q=(queue*)malloc(sizeof(queue));
  q->body=(char *)malloc(32*sizeof(char));
  /*for (int i = 0; i < 31; ++i) {
    q->path[i]=path[i];
  }*/
  q->size=0;
  q->cap=32;
  q->front=0;
  return q;
}
bool q_is_empty(queue *q){
  if(q->size==0){
    return true;
  } else {
```

```
        return false;
    }
}
bool q_grow(queue *q){
    int new_cap=2*q->cap;
    char *new_body=(char *)realloc(q->body,new_cap* sizeof(char));
    if(new_body==NULL){
        return false;
    }
    for(int i = q->size-1 ;i >= q->front; i=i-1){
        new_body[i+(new_cap-q->cap)]=q->body[i];
    }
    q->body=new_body;
    if(q->front==0) {
        q->front = q->front;
    }else{
        q->front = q->front+(q->cap-new_cap);
    }
    q->cap=new_cap;
    return true;
}
void q_shrink(queue *q){
    if(q->size > q->cap/4){
        return;
    }
    int new_cap=q->cap/2;
    if(new_cap < MIN_CAP){
        new_cap=MIN_CAP;
    }
    if(q->front+q->size >= q->cap){
        for (int i = q->front; i < q->cap; ++i) {
            q->body[i-(q->cap-new_cap)]=q->body[i];
        }
        q->front=q->front-(q->cap-new_cap);
    } else{
        for (int i = q->front; i < q->size+q->front; ++i) {
```

```
            q->body[i-(q->cap-new_cap)]=q->body[i];
        }
    }
    q->body=(char *)realloc(q->body, sizeof(char)*new_cap);
    q->cap=new_cap;
    return;
}

char pop(queue *q){
    char val=q->body[q->front];
    if (q->front==q->cap-1){
        q->front=0;
    } else {
        q->front++;
    }
    q->size--;
    return val;
}
void q_destroy(queue *q){
    free(q->body);
    for (int i = 0; i < 31; ++i) {
        q->body[i]='\0';
    }
    q->size=0;
    q->cap=0;
    q->front=0;
}
bool push(queue *q, char val){
    if(q->size==q->cap){
        if(!q_grow(q)){
            return false;
        }
    }
    q->body[(q->size+q->front)%q->cap]=val;
    q->size++;
    return true;
```

```
}
char q_front(queue *q) {
   return q->body[q->front];
}
int q_size(queue *q) {
   return q->size;
}
```

c_queue.h:

```
#ifndef D_QUEUE_NEW
#define D_QUEUE_NEW
#include <stdbool.h>
typedef struct {
   char *body;
   //char path[32];
   int size;
   int cap;
   int front;
}queue;
bool q_grow(queue *s);
void q_srink(queue *s);
queue *q_create();
void q_destroy(queue *q);
bool q_is_empty(queue *q);
bool push(queue *q, char val);
char pop(queue *q);
char q_front(queue *q);
int q_size(queue *q);
#endif
```

Вывод

Обрел навыки работы с системным вызовом fork(). Ознакомился с принципами работы pipe(), и применил этот системный вызов для передачи данных дочернему процессу. Изучил принципы работы буфера потоков, а также ознакомился с устройством взаимодействия процессов в ОС Linux.