北京交通大学实验报告

课程名称 ： **操作系统**
实验题目 ： **动态可重定位分区内存管理模拟设计与实现**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **何永忠**
报告日期 ： **2023年11月26日**

目录

---

1. 开发运行环境和工具

操作系统	Linux内核版本	处理器	GCC版本
Deepin 20.9	5.18.17-amd64-desktop-hwe	Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz	gcc (Uos 8.3.0.3-3+rebuild) 8.3.0

- **其他工具**:
  - **编辑**：VSCode
  - **编译和运行**：Terminal

2. 实验过程、分析和结论

2.1. 初始化

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

```

```
#define MAX_NUM_TASKS 128

struct Block {
    int start;
    int size;
    int pid;
    struct Block* next, * last;
};

struct Task {
    int start;
    int size;
    int pid;
    struct Block* block;
};

struct Request {
    int size;
    struct Request* next;
};


int preprocess_task_size(int);
int find_free_block_ff(int, struct Block**);
int find_free_block_bf(int, struct Block**);
void queue_push(int);
int queue_pop();
int get_command(FILE*);
int fork_process(int);
void kill_process(int);
int run(char*);
void init();
void clean_stdin();
void execute_wait_list();
void print();

int sys_size = 128 << 20, usr_size = 384 << 20;
struct Block* sys_space = NULL, * usr_space = NULL;
struct Task* tasks = NULL;
struct Request* waitlist_begin = NULL, * waitlist_end = NULL;
int waitlist_size = 0;
int last_pid = -1;
int match_method = -1;


int main() {
    init();
    while (get_command(stdin) >= 0);
    return 0;
}

void init() {
    sys_space = (struct Block*)malloc(sizeof(struct Block));
    sys_space->pid = -1;
    sys_space->start = 0;
    sys_space->size = sys_size;
```

```
    sys_space->next = NULL;
    sys_space->last = NULL;

    usr_space = (struct Block*)malloc(sizeof(struct Block));
    usr_space->pid = -1;
    usr_space->start = sys_size;
    usr_space->size = usr_size;
    usr_space->next = NULL;
    usr_space->last = NULL;

    tasks = (struct Task*)malloc(MAX_NUM_TASKS * sizeof(struct Task));
    for (int i = 0; i < MAX_NUM_TASKS; i++) tasks[i].pid = -1;

ask_match_method:
    printf("Choose a matching method:\n 1. First Fit\n 2. Best Fit\n");
    printf("choose> ");
    scanf("%d", &match_method);
    clean_stdin();
    if (match_method != 1 && match_method != 2) {
        printf("Bad input. Choose again.\n");
        goto ask_match_method;
    }

    waitlist_begin = (struct Request*)malloc(sizeof(struct Request));
    waitlist_end = waitlist_begin;
    waitlist_begin->next = NULL;
    waitlist_begin->size = -1;
}
```

2.2. 申请内存分配操作

```
int fork_process(int size) {
    int pid;
    int task_index = -1;
repeat:
    last_pid++;
    if (last_pid < 0) last_pid = 0;
    pid = last_pid;
    for (int i = 0; i < MAX_NUM_TASKS; i++) {
        if (tasks[i].pid >= 0 && tasks[i].pid == pid) goto repeat;
    }
    for (int i = 0; i < MAX_NUM_TASKS; i++) {
        if (tasks[i].pid >= 0) continue;
        task_index = i;
        break;
    }
    if (task_index == -1) return -1;

    size = preprocess_task_size(size);
    tasks[task_index].size = size;
    int start_loc = -1;
    if (match_method == 1) start_loc = find_free_block_ff(size,
&tasks[task_index].block);
```

```
    else start_loc = find_free_block_bf(size, &tasks[task_index].block);
    if (start_loc >= 0) {
        tasks[task_index].start = start_loc;
        tasks[task_index].pid = pid;
        tasks[task_index].size = size;
        tasks[task_index].block->pid = pid;
        printf("Fork succeed. PID is %d . Block size is %d bits.\n", pid,
size);
        return pid;
    }
    printf("Fork failed. Out of Memory. Trying to push into wait list.\n");
    queue_push(size);
    return -1;
}

int find_free_block_ff(int size, struct Block** block_p) {
    struct Block* now = usr_space;
    int cnt = 1;
    while (1) {
        if (now == NULL) return -1;
        if (now->pid < 0 && now->size >= size) break;
        now = now->next;
        cnt++;
    }
    printf("Use %d times to find free block.\n", cnt);
    *block_p = now;
    if (now->size == size) return now->start;
    struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
    free_space->pid = -1;
    free_space->last = now;
    free_space->next = now->next;
    free_space->size = now->size - size;
    free_space->start = now->start + size;

    now->pid = -2;
    now->next = free_space;
    now->size = size;
    return now->start;
}

int find_free_block_bf(int size, struct Block** block_p) {
    struct Block* now = usr_space, * best_block;
    int best_size = 0x7fffffff;
    int cnt = 1;
    while (1) {
        if (now == NULL) break;
        if (now->pid < 0 && now->size >= size && best_size >= now->size) {
            best_block = now;
            best_size = now->size;
        }
        now = now->next;
        cnt++;
    }
    if (best_size == 0x7fffffff) return -1;
    printf("Use %d times to find free block.\n", cnt);
    *block_p = best_block;
```

```
    if (best_block->size == size) return best_block->start;
    struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
    free_space->pid = -1;
    free_space->last = best_block;
    free_space->next = best_block->next;
    free_space->size = best_block->size - size;
    free_space->start = best_block->start + size;

    best_block->pid = -2;
    best_block->next = free_space;
    best_block->size = size;
    return best_block->start;
}

int run(char* path) {
    FILE* f = NULL;
    f = fopen(path, "r+");
    if (f == NULL) {
        printf("Error: Invalid file path.\n");
        return 0;
    }
    int result = 0;
    while (!feof(f) && result != -1) result = get_command(f);
    fclose(f);
    return result;
}

int preprocess_task_size(int size) {
    int lower = size & 0x3ff;
    if (lower == 0) return size;
    int upper = size - lower;
    size = upper + 0x400;
    return size;
}

void queue_push(int size) {
    waitlist_begin->next = (struct Request*)malloc(sizeof(struct Request));
    waitlist_begin = waitlist_begin->next;
    waitlist_begin->next = NULL;
    waitlist_begin->size = size;
    waitlist_size++;
}

int queue_pop() {
    if (!waitlist_size) {
        printf("Wait list empty.\n");
        return -1;
    }
    struct Request* tmp = waitlist_end;
    waitlist_end = waitlist_end->next;
    free(tmp);
    waitlist_size--;
    return waitlist_end->size;
}
```

2.3. 回收

```
void kill_process(int pid) {
    struct Block* block = NULL;
    for (int i = 0; i < MAX_NUM_TASKS; i++) {
        if (tasks[i].pid != pid) continue;

        block = tasks[i].block;
        break;
    }
    if (block == NULL) {
        printf("Error: PID dosen't exist.\n");
        return;
    }
    if (block->last && block->last->pid < 0 && block->next && block->next->pid < 0) {
        struct Block* free_block = block->last;
        free_block->next = block->next->next;
        free_block->size += block->size + block->next->size;
        if (free_block->next) free_block->next->last = free_block;
        free(block->next);
        free(block);
    }
    else if (block->last && block->last->pid < 0) {
        struct Block* free_block = block->last;
        free_block->next = block->next;
        free_block->size += block->size;
        if (free_block->next) free_block->next->last = free_block;
        free(block);
    }
    else if (block->next && block->next->pid < 0) {
        struct Block* free_block = block;
        free_block->size += block->next->size;
        free_block->next = block->next->next;
        free_block->pid = -1;
        if (free_block->next) free_block->next->last = free_block;
        free(block->next);
    }
    else {
        block->pid = -1;
    }

    execute_wait_list();

    return;
}

void execute_wait_list() {
    int size = -1, loop_num = waitlist_size;
    for (int i = 0; i < loop_num; i++) {
        size = queue_pop();
        fork_process(size);
    }
```

```
}
```

2.4. 其他代码

```
int get_command(FILE* stream) {
    printf("user> ");
    char input_str[128] = { 0 };
    fgets(input_str, 127, stream);
    if (stream != stdin) printf("%s", input_str);

    int split_index = 0;
    for (int i = 0; i < 127; i++) {
        if (input_str[i] == '\0') break;
        if (input_str[i] == ' ') {
            input_str[i] = '\0';
            split_index = i;
        }
        if (input_str[i] == '\n') {
            input_str[i] = '\0';
            break;
        }
    }

    char* command = input_str, * arguments = input_str + split_index + 1;

    if (!strcmp(command, "exit")) {
        return -1;
    }
    else if (strcmp(command, "fork") == 0) {
        int input_size = atoi(arguments);
        if (input_size <= 0) {
            printf("Bad input size. Input number must larger than
ZERO.\n");
            return 0;
        }
        else if (input_size > usr_size) {
            printf("Bad input size. Input number must smaller than %d
bits.\n", usr_size);
            return 0;
        }
        int pid = fork_process(input_size);
        return 1;
    }
    else if (!strcmp(command, "kill")) {
        int input_pid = atoi(arguments);
        if (input_pid < 0) {
            printf("Bad input PID. Input number must not smaller than
ZERO.\n");
            return 0;
        }
        kill_process(input_pid);
        return 1;
    }
```

```
    else if (!strcmp(input_str, "print")) {
        print();
        return 1;
    }
    else if (!strcmp(input_str, "run")) {
        int result = run(arguments);
        return result;
    }
    else if (input_str[0] == '\0');
    else {
        printf("Command not found.\n");
        return 0;
    }
}

void print() {
    char str[2][13] = { {"        True"}, {"       False"} };
    char* s;
    struct Block* p = sys_space;
    printf("+------------------------------------------------------------------
--+\n");
    printf("|                          System Space
|\n");
    printf("+------------+-----------+-----------+-----------+----------
--+\n");
    printf("|    Free    |   Begin    |    Size    |    End     |     PID
|\n");
    printf("+------------+-----------+-----------+-----------+----------
--+\n");
    while (p != NULL) {
        s = p->pid < 0 ? str[0] : str[1];
        printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
+ p->size - 1, p->pid);
        printf("+------------+-----------+-----------+-----------+------
------+\n");
        p = p->next;
    }
    printf("+------------------------------------------------------------------
--+\n\n");

    p = usr_space;
    printf("+------------------------------------------------------------------
--+\n");
    printf("|                           User Space
  |\n");
    printf("+------------+-----------+-----------+-----------+----------
--+\n");
    printf("|    Free    |   Begin    |    Size    |    End     |     PID
|\n");
    printf("+------------+-----------+-----------+-----------+----------
--+\n");
    while (p != NULL) {
        s = p->pid < 0 ? str[0] : str[1];
        printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
+ p->size - 1, p->pid);
```

```
        printf("+-----------+-----------+-----------+-----------+-----------+\n");
        p = p->next;
    }
    printf("+-------------------------------------------------------------------+\n");
}

void clean_stdin() {
    int c;
    do {
        c = getchar();
    } while (c != '\n' && c != EOF);
}

int run(char* path) {
    FILE* f = NULL;
    f = fopen(path, "r+");
    if (f == NULL) {
        printf("Error: Invalid file path.\n");
        return 0;
    }
    int result = 0;
    while (!feof(f) && result != -1) result = get_command(f);
    fclose(f);
    return result;
}
```

附录

```
#include <stdio.h>
#include <malloc.h>
#include <string.h>
#include <stdlib.h>

#define MAX_NUM_TASKS 128

struct Block {
    int start;
    int size;
    int pid;
    struct Block* next, * last;
};

struct Task {
    int start;
    int size;
    int pid;
    struct Block* block;
};

struct Request {
```

```
    int size;
    struct Request* next;
};


int preprocess_task_size(int);
int find_free_block_ff(int, struct Block**);
int find_free_block_bf(int, struct Block**);
void queue_push(int);
int queue_pop();
int get_command(FILE*);
int fork_process(int);
void kill_process(int);
int run(char*);
void init();
void clean_stdin();
void execute_wait_list();
void print();

int sys_size = 128 << 20, usr_size = 384 << 20;
struct Block* sys_space = NULL, * usr_space = NULL;
struct Task* tasks = NULL;
struct Request* waitlist_begin = NULL, * waitlist_end = NULL;
int waitlist_size = 0;
int last_pid = -1;
int match_method = -1;


int main() {
    init();
    while (get_command(stdin) >= 0);
    return 0;
}

int get_command(FILE* stream) {
    printf("user> ");
    char input_str[128] = { 0 };
    fgets(input_str, 127, stream);
    if (stream != stdin) printf("%s", input_str);

    int split_index = 0;
    for (int i = 0; i < 127; i++) {
        if (input_str[i] == '\0') break;
        if (input_str[i] == ' ') {
            input_str[i] = '\0';
            split_index = i;
        }
        if (input_str[i] == '\n') {
            input_str[i] = '\0';
            break;
        }
    }

    char* command = input_str, * arguments = input_str + split_index + 1;

    if (!strcmp(command, "exit")) {
```

```
        return -1;
    }
    else if (strcmp(command, "fork") == 0) {
        int input_size = atoi(arguments);
        if (input_size <= 0) {
            printf("Bad input size. Input number must larger than
ZERO.\n");
            return 0;
        }
        else if (input_size > usr_size) {
            printf("Bad input size. Input number must smaller than %d
bits.\n", usr_size);
            return 0;
        }
        int pid = fork_process(input_size);
        return 1;
    }
    else if (!strcmp(command, "kill")) {
        int input_pid = atoi(arguments);
        if (input_pid < 0) {
            printf("Bad input PID. Input number must not smaller than
ZERO.\n");
            return 0;
        }
        kill_process(input_pid);
        return 1;
    }
    else if (!strcmp(input_str, "print")) {
        print();
        return 1;
    }
    else if (!strcmp(input_str, "run")) {
        int result = run(arguments);
        return result;
    }
    else if (input_str[0] == '\0');
    else {
        printf("Command not found.\n");
        return 0;
    }
}

void kill_process(int pid) {
    struct Block* block = NULL;
    for (int i = 0; i < MAX_NUM_TASKS; i++) {
        if (tasks[i].pid != pid) continue;

        block = tasks[i].block;
        break;
    }
    if (block == NULL) {
        printf("Error: PID dosen't exist.\n");
        return;
    }
    if (block->last && block->last->pid < 0 && block->next && block->next-
>pid < 0) {
```

```
131         struct Block* free_block = block->last;
132         free_block->next = block->next->next;
133         free_block->size += block->size + block->next->size;
134         if (free_block->next) free_block->next->last = free_block;
135         free(block->next);
136         free(block);
137     }
138     else if (block->last && block->last->pid < 0) {
139         struct Block* free_block = block->last;
140         free_block->next = block->next;
141         free_block->size += block->size;
142         if (free_block->next) free_block->next->last = free_block;
143         free(block);
144     }
145     else if (block->next && block->next->pid < 0) {
146         struct Block* free_block = block;
147         free_block->size += block->next->size;
148         free_block->next = block->next->next;
149         free_block->pid = -1;
150         if (free_block->next) free_block->next->last = free_block;
151         free(block->next);
152     }
153     else {
154         block->pid = -1;
155     }
156 
157     execute_wait_list();
158 
159     return;
160 }
161 
162 void init() {
163     sys_space = (struct Block*)malloc(sizeof(struct Block));
164     sys_space->pid = -1;
165     sys_space->start = 0;
166     sys_space->size = sys_size;
167     sys_space->next = NULL;
168     sys_space->last = NULL;
169 
170     usr_space = (struct Block*)malloc(sizeof(struct Block));
171     usr_space->pid = -1;
172     usr_space->start = sys_size;
173     usr_space->size = usr_size;
174     usr_space->next = NULL;
175     usr_space->last = NULL;
176 
177     tasks = (struct Task*)malloc(MAX_NUM_TASKS * sizeof(struct Task));
178     for (int i = 0; i < MAX_NUM_TASKS; i++) tasks[i].pid = -1;
179 
180 ask_match_method:
181     printf("Choose a matching method:\n 1. First Fit\n 2. Best Fit\n");
182     printf("choose> ");
183     scanf("%d", &match_method);
184     clean_stdin();
185     if (match_method != 1 && match_method != 2) {
186         printf("Bad input. Choose again.\n");
```

```
        goto ask_match_method;
    }

    waitlist_begin = (struct Request*)malloc(sizeof(struct Request));
    waitlist_end = waitlist_begin;
    waitlist_begin->next = NULL;
    waitlist_begin->size = -1;
}

int fork_process(int size) {
    int pid;
    int task_index = -1;
repeat:
    last_pid++;
    if (last_pid < 0) last_pid = 0;
    pid = last_pid;
    for (int i = 0; i < MAX_NUM_TASKS; i++) {
        if (tasks[i].pid >= 0 && tasks[i].pid == pid) goto repeat;
    }
    for (int i = 0; i < MAX_NUM_TASKS; i++) {
        if (tasks[i].pid >= 0) continue;
        task_index = i;
        break;
    }
    if (task_index == -1) return -1;

    size = preprocess_task_size(size);
    tasks[task_index].size = size;
    int start_loc = -1;
    if (match_method == 1) start_loc = find_free_block_ff(size,
&tasks[task_index].block);
    else start_loc = find_free_block_bf(size, &tasks[task_index].block);
    if (start_loc >= 0) {
        tasks[task_index].start = start_loc;
        tasks[task_index].pid = pid;
        tasks[task_index].size = size;
        tasks[task_index].block->pid = pid;
        printf("Fork succeed. PID is %d . Block size is %d bits.\n", pid,
size);
        return pid;
    }
    printf("Fork failed. Out of Memory. Trying to push into wait list.\n");
    queue_push(size);
    return -1;
}

int find_free_block_ff(int size, struct Block** block_p) {
    struct Block* now = usr_space;
    int cnt = 1;
    while (1) {
        if (now == NULL) return -1;
        if (now->pid < 0 && now->size >= size) break;
        now = now->next;
        cnt++;
    }
    printf("Use %d times to find free block.\n", cnt);
```

```
    *block_p = now;
    if (now->size == size) return now->start;
    struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
    free_space->pid = -1;
    free_space->last = now;
    free_space->next = now->next;
    free_space->size = now->size - size;
    free_space->start = now->start + size;

    now->pid = -2;
    now->next = free_space;
    now->size = size;
    return now->start;
}

int find_free_block_bf(int size, struct Block** block_p) {
    struct Block* now = usr_space, * best_block;
    int best_size = 0x7fffffff;
    int cnt = 1;
    while (1) {
        if (now == NULL) break;
        if (now->pid < 0 && now->size >= size && best_size >= now->size) {
            best_block = now;
            best_size = now->size;
        }
        now = now->next;
        cnt++;
    }
    if (best_size == 0x7fffffff) return -1;
    printf("Use %d times to find free block.\n", cnt);
    *block_p = best_block;
    if (best_block->size == size) return best_block->start;
    struct Block* free_space = (struct Block*)malloc(sizeof(struct Block));
    free_space->pid = -1;
    free_space->last = best_block;
    free_space->next = best_block->next;
    free_space->size = best_block->size - size;
    free_space->start = best_block->start + size;

    best_block->pid = -2;
    best_block->next = free_space;
    best_block->size = size;
    return best_block->start;
}

int run(char* path) {
    FILE* f = NULL;
    f = fopen(path, "r+");
    if (f == NULL) {
        printf("Error: Invalid file path.\n");
        return 0;
    }
    int result = 0;
    while (!feof(f) && result != -1) result = get_command(f);
    fclose(f);
    return result;
```

```
297 }
298
299 int preprocess_task_size(int size) {
300     int lower = size & 0x3ff;
301     if (lower == 0) return size;
302     int upper = size - lower;
303     size = upper + 0x400;
304     return size;
305 }
306
307 void queue_push(int size) {
308     waitlist_begin->next = (struct Request*)malloc(sizeof(struct Request));
309     waitlist_begin = waitlist_begin->next;
310     waitlist_begin->next = NULL;
311     waitlist_begin->size = size;
312     waitlist_size++;
313 }
314
315 int queue_pop() {
316     if (!waitlist_size) {
317         printf("Wait list empty.\n");
318         return -1;
319     }
320     struct Request* tmp = waitlist_end;
321     waitlist_end = waitlist_end->next;
322     free(tmp);
323     waitlist_size--;
324     return waitlist_end->size;
325 }
326
327 void clean_stdin() {
328     int c;
329     do {
330         c = getchar();
331     } while (c != '\n' && c != EOF);
332 }
333
334 void execute_wait_list() {
335     int size = -1, loop_num = waitlist_size;
336     for (int i = 0; i < loop_num; i++) {
337         size = queue_pop();
338         fork_process(size);
339     }
340 }
341
342 void print() {
343     char str[2][13] = { {"        True"}, {"       False"} };
344     char* s;
345     struct Block* p = sys_space;
346     printf("+-------------------------------------------------------------
--+\n");
347     printf("|                          System Space
|\n");
348     printf("+-----------+-----------+-----------+-----------+----------
--+\n");
```

```
349     printf("|    Free    |   Begin    |    Size    |    End     |     PID
|\n");
350     printf("+------------+------------+------------+------------+----------
--+\n");
351     while (p != NULL) {
352         s = p->pid < 0 ? str[0] : str[1];
353         printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
+ p->size - 1, p->pid);
354         printf("+------------+------------+------------+------------+------
------+\n");
355         p = p->next;
356     }
357     printf("+-------------------------------------------------------------
--+\n\n");
358
359     p = usr_space;
360     printf("+-------------------------------------------------------------
--+\n");
361     printf("|                            User Space
  |\n");
362     printf("+------------+------------+------------+------------+----------
--+\n");
363     printf("|    Free    |   Begin    |    Size    |    End     |     PID
|\n");
364     printf("+------------+------------+------------+------------+----------
--+\n");
365     while (p != NULL) {
366         s = p->pid < 0 ? str[0] : str[1];
367         printf("|%s|%12d|%12d|%12d|%12d|\n", s, p->start, p->size, p->start
+ p->size - 1, p->pid);
368         printf("+------------+------------+------------+------------+------
------+\n");
369         p = p->next;
370     }
371     printf("+-------------------------------------------------------------
--+\n");
372 }
```