实 验 报 告

课程名称： 操作系统实验

学 院： 计算机科学与工程学院

专 业： 软件工程 班 级： 软件18-1班

姓 名：KAFLE SAMRAT学 号： 201801060933

山 东 科 技 大 学

实 验 报 告

页

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 组 别 |  | 姓 名 | KAFLE SAMRAT | 同组实验者 |  |
| 实验项目  名称 | Linux内存管理 | | 实验日期 |  | |
| 教师评语 |  | | | | |
| 实验成绩： | | | 指导教师（签名）：  年 月 日 | | |

**实验目的：**

1， 通过本次试验体会操作系统中内存的分配模式；

2， 掌握内存分配的方法（FF,BF,WF）；

3， 学会进程的建立，当一个进程被终止时内存是如何处理被释放块，并当内存不满足进程申请时是如何使用内存紧凑；

4， 掌握内存回收过程及实现方法；

5， 学会进行内存的申请释放和管理；

**实验内容：**

#include<stdio.h>

#include<stdlib.h>

#define PROCESS\_NAME\_LEN 32 /\* Process name length \*/

#define MIN\_SLICE 10 /\* The size of the smallest fragment \*/

#define DEFAULT\_MEM\_SIZE 1024 /\* Memory size \*/

#define DEFAULT\_MEM\_START 0 /\* The starting position \*/

/\* Memory allocation algorithm \*/

#define MA\_FF 1

#define MA\_BF 2

#define MA\_WF 3

/\* Describes the data structure for each free block \*/

typedef struct free\_block\_type{

int size;

int start\_addr;

struct free\_block\_type \*next;

}FBT;

/\* A description of the memory blocks allocated by each process \*/

typedef struct allocated\_block{

int pid;

int size;

int start\_addr;

char process\_name[PROCESS\_NAME\_LEN];

struct allocated\_block \*next;

}AB;

/\* A pointer to the first pointer to a list of free blocks in memory \*/

FBT \*free\_block;

/\* The process allocates the first pointer to a linked list of memory blocks \*/

AB \*allocated\_block\_head = NULL;

int mem\_size = DEFAULT\_MEM\_SIZE; /\* Memory size \*/

int ma\_algorithm = MA\_FF; /\* Current allocation algorithm \*/

static int pid = 0; /\*The initial pid\*/

int flag = 0; /\* Sets the memory size flag \*/

int min\_mem\_size = 10; /\* Sets a flag that the remaining partitions are too small \*/

FBT \*init\_free\_block(int mem\_size);

void display\_menu();

int set\_mem\_size();

int display\_mem\_usage();

int dispose(AB \*free\_ab);

int free\_mem();

int kill\_process();

int allocate\_mem(AB \*ab);

int new\_process();

void rearrange\_FF();

void rearrange\_BF();

void rearrange\_WF();

void rearrange(int algorithm);

void set\_algorithm();

void do\_exit(){

/\* The general operating system will reclaim the applied memory after the program exit or return, so the place is empty.\*/

return;

}

int main(int argc, char const \*argv[]){

/\* code \*/

char choice;

pid = 0;

free\_block = init\_free\_block(mem\_size); // Initialize the free zone

while(1){

fflush(stdin);

display\_menu(); // According to the menu

fflush(stdin);

while((choice = getchar()) != '\n'){

//choice = getchar();

fflush(stdin);

switch(choice){

case '1':set\_mem\_size();break;

case '2':set\_algorithm();flag = 1;break;

case '3':new\_process();flag = 1;break;

case '4':kill\_process();flag = 1;break;

case '5':display\_mem\_usage();flag = 1;break;

case '0':do\_exit();exit(0);

default: break;

}

fflush(stdin);

}

}

}

void display\_menu(){

puts("");

printf("1 - Set memory size(fedault=%d)\n",DEFAULT\_MEM\_SIZE);

printf("2 - Select memory allocation algorithm\n");

printf("3 - New process\n");

printf("4 - Terminate a process \n");

printf("5 - Display memory usage\n");

printf("0 - Exit\n");

}

// Initializes a linked list of free partitions

FBT \*init\_free\_block(int mem\_size){

FBT \*fb;

fb = (FBT\*)malloc(sizeof(FBT));

if(fb==NULL){

printf("No mem\n");

return NULL;

}

fb->size = mem\_size;

fb->start\_addr = DEFAULT\_MEM\_START;

fb->next = NULL;

return fb;

}

// Reset the memory size

int set\_mem\_size(){

int size;

if(flag!=0){

printf("Cannot set memory size again\n");

return 0;

}

printf("Total memory size =");

scanf("%d",&size);

if(size>0){

mem\_size = size;

free\_block->size = mem\_size;

}

flag = 1;

min\_mem\_size = mem\_size / 100;

return 1;

}

int display\_mem\_usage(){

/\* Displays current memory usage, including free partitions and allocated ones \*/

FBT \*fbt = free\_block;

AB \*ab = allocated\_block\_head;

// if(fbt == NULL) return -1;

printf("\e[0;31;1m------------------------------------------------------------------\e[0m\n");

// Display free area

printf("\e[0;32;1mFree Memory:\e[0m\n");

printf("\e[0;33;1m%20s %20s\e[0m\n"," start\_addr"," size");

while(fbt!=NULL){

printf("%20d %20d\n",fbt->start\_addr,fbt->size);

fbt = fbt->next;

}

// Displays the allocated areas

printf("\n");

printf("\e[0;35;1mUsed Memory:\e[0m\n");

printf("\e[0;33;1m%10s %20s %20s %10s\e[0m\n","PID","ProcessName","start\_addr","size");

while(ab != NULL){

printf("%10d %20s %20d %10d\n",ab->pid,ab->process\_name,ab->start\_addr,ab->size);

ab = ab->next;

}

printf("\e[0;31;1m------------------------------------------------------------------\e[0m\n");

return 0;

}

// Release the linked list nodes

int dispose(AB \*free\_ab){

/\* Release ab data structure nodes \*/

AB \*pre,\*ab;

if(free\_ab == allocated\_block\_head){

// If you want to release the first node

allocated\_block\_head = allocated\_block\_head->next;

free(free\_ab);

return 1;

}

pre = allocated\_block\_head;

ab = allocated\_block\_head->next;

while(ab!=free\_ab){

pre = ab;

ab = ab->next;

}

pre->next = ab->next;

free(ab);

return 2;

}

// Frees up memory occupied by the process

int free\_mem(AB \*ab){

/\* Return the allocated areas represented by AB and make a possible merge \*/

int algorithm = ma\_algorithm;

FBT \*fbt,\*pre,\*work;

fbt = (FBT\*)malloc(sizeof(FBT));

if(!fbt) return -1;

/\*

For a possible merger, the basic strategy is as follows

1. Insert the newly released node to the end of the queue in the free partition

2. Organize free lists by address

3. Check and merge adjacent free partitions

4. Reorder the free linked list according to the current algorithm \*/

fbt->size = ab->size;

fbt->start\_addr = ab->start\_addr;

// Insert to the end

work = free\_block;

if(work == NULL){

free\_block = fbt;

fbt->next == NULL;

}else{

while(work ->next != NULL){

work = work->next;

}

fbt->next = work->next;

work->next = fbt;

}

// Rearrange the layout according to the address

rearrange\_FF();

/\* Merge possible partitions; If two free partitions are connected, they are merged\*/

pre = free\_block;

while(pre->next){

work = pre->next;

if(pre->start\_addr + pre->size == work->start\_addr ){

pre->size = pre->size + work->size;

pre->next = work->next;

free(work);

continue;

}else{

pre = pre->next;

}

}

// Sort by the current algorithm

rearrange(ma\_algorithm);

return 1;

}

// Find the linked list node corresponding to pid

AB \*find\_process(int pid){

AB \*tmp = allocated\_block\_head;

while(tmp != NULL){

if(tmp->pid == pid){

return tmp;

}

tmp = tmp->next;

}

printf("\e[0;31;1m Cannot find pid:%d \e[0m\n",pid);

return NULL;

}

int kill\_process(){

AB \*ab;

int pid;

printf("Kill Process,pid=");

scanf("%d",&pid);

ab = find\_process(pid);

if(ab!=NULL){

free\_mem(ab); // Release the allocation table represented by ab

dispose(ab); // Release ab data structure nodes

return 0;

}else{

return -1;

}

}

// Find if there are partitions that can be non-process allocated

int find\_free\_mem(int request){

FBT \*tmp = free\_block;

int mem\_sum = 0;

while(tmp){

if(tmp->size >= request){

// Can be directly allocated

return 1;

}

mem\_sum += tmp->size;

tmp = tmp->next;

}

if(mem\_sum >= request){

// Post-merge allocation

return 0;

}else{

// There is not enough space to allocate

return -1;

}

}

// Sort the allocated table from large to small by starting address

void sort\_AB(){

if(allocated\_block\_head == NULL || allocated\_block\_head->next == NULL)

return;

AB \*t1,\*t2,\*head;

head = allocated\_block\_head;

for(t1 = head->next;t1;t1 = t1->next){

for(t2 = head;t2 != t1;t2=t2->next){

if(t2->start\_addr > t2->next->start\_addr){

int tmp = t2->start\_addr;

t2->start\_addr = t2->next->start\_addr;

t2->next->start\_addr = tmp;

tmp = t2->size;

t2->size = t2->next->size;

t2->next->size = tmp;

}

}

}

}

// Reassign memory addresses to all processes

void reset\_AB(int start){

/\* In a real operating system this is not easy, so memory crunch is not frequently used \*/

AB \*tmp = allocated\_block\_head;

while(tmp != NULL){

tmp->start\_addr = start;

start += tmp->size;

tmp = tmp->next;

}

}

void memory\_compact(){

// Squeeze memory

FBT \*fbttmp = free\_block;

AB \*abtmp = allocated\_block\_head;

// Detect remaining memory

int sum = 0;

while(fbttmp!=NULL){

sum += fbttmp->size;

fbttmp = fbttmp->next;

}

// Merge blocks into one

fbttmp = free\_block;

fbttmp->size = sum;

fbttmp->start\_addr = 0;

fbttmp->next=NULL;

// Release redundant partitions

FBT \*pr = free\_block->next;

while(pr != NULL){

fbttmp = pr->next;

free(pr);

pr = fbttmp;

}

// Reorder the allocated space

sort\_AB();

reset\_AB(sum);

}

// Perform memory allocation

void do\_allocate\_mem(AB \*ab){

int request = ab->size;

FBT \*tmp = free\_block;

while(tmp != NULL){

if(tmp->size >= request){

// allocation

ab->start\_addr = tmp->start\_addr;

int shengyu = tmp->size - request;

if(shengyu <= min\_mem\_size){

// The surplus is too small to allocate all

ab->size = tmp->size;

if(tmp == free\_block){

free\_block = free\_block->next;

free(tmp);

}else{

FBT \*t = free\_block;

while(t->next != tmp){

t = t->next;

}

t->next = tmp->next;

free(tmp);

}

}else{

// Cut out the allocated memory

tmp->size = shengyu;

tmp->start\_addr = tmp->start\_addr + request;

}

return ;

}

tmp = tmp->next;

}

}

int allocate\_mem(AB \*ab){

/\* Allocated memory module \*/

FBT \*fbt,\*pre;

int request\_size=ab->size;

fbt = pre = free\_block;

/\*

According to the current algorithm, the appropriate free partition is searched in the linked list of free partition for allocation.

Pay attention to the following situations when allocating:

1. If the free partition can be found and the remaining space after allocation is large enough, then divide

2. If the free partition can be found and the remaining space after allocation is relatively small, then allocate it together

3. Find the free partitions that cannot meet the needs, but the sum of the free partitions can meet the needs.

Then the memory compression technique is adopted to merge the free partitions and then redistribute them

4. After the successful allocation of memory, the free partition should be kept in order according to the corresponding algorithm

5. Return 1 if the allocation is successful, otherwise return -1 \*/

// Try to find allocable idle, the results of which are explained in the function

int f = find\_free\_mem(request\_size);

if(f == -1){

// Allocate enough

printf("Free mem is not enough,Allocate fail!\n");

return -1;

}else{

if(f == 0){

// Memory crunch is required to allocate

memory\_compact();

}

// Perform assigned

do\_allocate\_mem(ab);

}

// Rearrange the free partitions

rearrange(ma\_algorithm);

return 1;

}

// Create a new process

int new\_process(){

AB \*ab;

int size;

int ret;

ab = (AB\*)malloc(sizeof(AB));

if(!ab) exit(-5);

ab->next=NULL;

pid++;

sprintf(ab->process\_name,"PROCESS-%02d",pid);

ab->pid = pid;

printf("Memory for %s:",ab->process\_name);

scanf("%d",&size);

if(size>0) ab->size=size;

ret = allocate\_mem(ab); /\* Allocate memory from the free partition, ret==1 indicates successful allocation \*/

if((ret == 1) && (allocated\_block\_head == NULL)){

allocated\_block\_head = ab;

return 1;

}else if(ret == 1){

/\* Successfully allocate, inserts the description of the allocated block into the allocated linked list \*/

ab->next = allocated\_block\_head;

allocated\_block\_head = ab;

return 2;

}else if(ret == -1){

//分配不成功

printf("\e[0;31;1m Allocation fail \e[0m\n");

free(ab);

return -1;

}

return 3;

}

void rearrange\_FF(){

/\* For the first time, the free zone size is sorted in ascending order according to the starting address \*/

// We use bubble sort here

if(free\_block == NULL || free\_block->next == NULL)

return;

FBT \*t1,\*t2,\*head;

head = free\_block;

for(t1 = head->next;t1;t1 = t1->next){

for(t2 = head;t2 != t1;t2=t2->next){

if(t2->start\_addr > t2->next->start\_addr){

int tmp = t2->start\_addr;

t2->start\_addr = t2->next->start\_addr;

t2->next->start\_addr = tmp;

tmp = t2->size;

t2->size = t2->next->size;

t2->next->size = tmp;

}

}

}

}

void rearrange\_BF(){

/\* The best adaptive algorithm sorts free partitions by size from small to large \*/

if(free\_block == NULL || free\_block->next == NULL)

return;

FBT \*t1,\*t2,\*head;

head = free\_block;

for(t1 = head->next;t1;t1 = t1->next){

for(t2 = head;t2 != t1;t2=t2->next){

if(t2->size > t2->next->size){

int tmp = t2->start\_addr;

t2->start\_addr = t2->next->start\_addr;

t2->next->start\_addr = tmp;

tmp = t2->size;

t2->size = t2->next->size;

t2->next->size = tmp;

}

}

}

}

void rearrange\_WF(){

/\* The worst fit algorithm sorts free partitions from large to small \*/

if(free\_block == NULL || free\_block->next == NULL)

return;

FBT \*t1,\*t2,\*head;

head = free\_block;

for(t1 = head->next;t1;t1 = t1->next){

for(t2 = head;t2 != t1;t2=t2->next){

if(t2->size < t2->next->size){

int tmp = t2->start\_addr;

t2->start\_addr = t2->next->start\_addr;

t2->next->start\_addr = tmp;

tmp = t2->size;

t2->size = t2->next->size;

t2->next->size = tmp;

}

}

}

}

/\* Collates a list of memory free blocks according to the specified algorithm \*/

void rearrange(int algorithm){

switch(algorithm){

case MA\_FF:rearrange\_FF();break;

case MA\_BF:rearrange\_BF();break;

case MA\_WF:rearrange\_WF();break;

}

}

void set\_algorithm(){

/\* Sets the current allocation algorithm \*/

int algorithm;

printf("\t1 - First Fit\n");

printf("\t2 - Best Fit\n");

printf("\t3 - Worst Fit\n");

scanf("%d",&algorithm);

if(algorithm>=1 && algorithm<=3)

ma\_algorithm = algorithm;

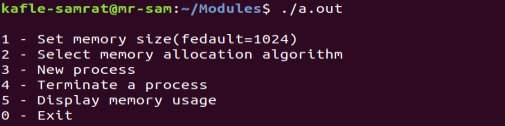
// Rearrange the list of free areas according to the specified algorithm

rearrange(ma\_algorithm);

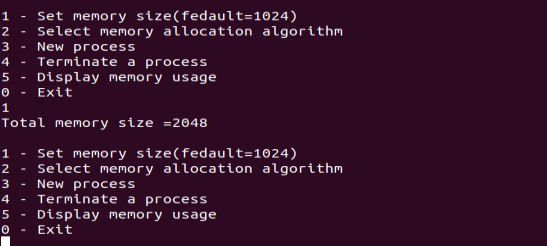
}

**实验结果：**

The experimental interface：

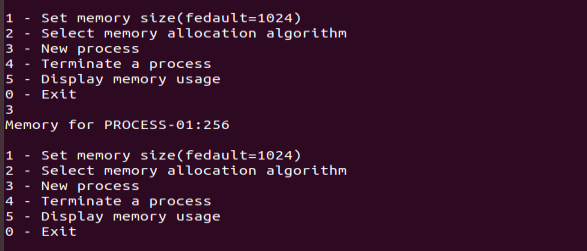


After prompted for input, type 1, and the display is as follows:



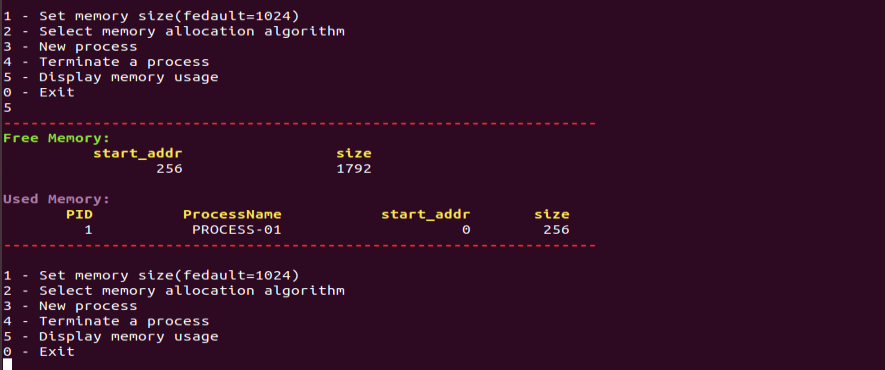
Then input: 3 and set the memory space to 256.

The display is as follows:

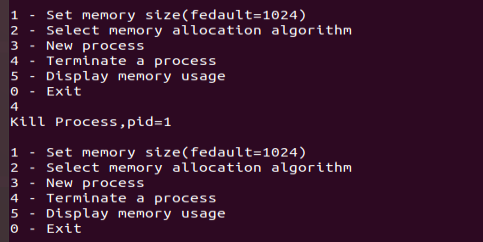


Repeat the previous operation. Input : 5

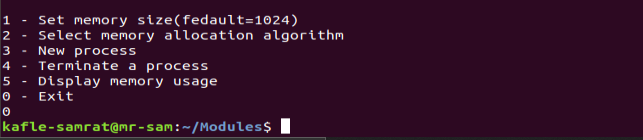
the display is as follows:



After entering : input： 4, kill process 1, as shown below:



Input 0 to Exit the process



**实验心得体会：**

通过计算机实验让我进一步了解操作系统对内存分配的知识，也让我认识到C语言的重要性，对记忆分配的方法和思路可以理解，但在具体实现时我感觉有点困难，通过与同学的沟通和咨询相关信息来发现问题，这些都是C语言基础不扎实，而且长期不练习，将来编程练习太重**。**