实验过程

- **First Fit (FF)**

  算法思想

  在 First Fit 算法中，分配器会扫描空闲块列表，找到第一个足够大的空闲块，并将其分配出去。如果该空闲块大于请求的内存，则会分割该块。

  优缺点

  - **优点**: 查找速度相对较快，因为一旦找到合适的空闲块，立即停止搜索。
  - **缺点**: 容易导致内存碎片，特别是外部碎片。

  提高查找性能

  在本次实验中好像并无。

- **Best Fit (BF)**

  算法思想

  Best Fit 算法在所有的空闲块中找到与请求最匹配（即最接近）的空闲块。这通常需要扫描整个空闲块列表。

  优缺点

  - **优点**: 最小化了每次分配后的剩余空间，从而减少了浪费。

- **缺点**: 查找性能较差，因为通常需要扫描整个列表；可能导致更多小的空闲块，进而增加碎片。

  提高查找性能

  维护一个按照大小排序的空闲块列表来提高查找性能。

- **Worst Fit (WF)**

  算法思想

  Worst Fit 算法选择最大的可用空闲块进行分配，假设剩下的空间可能更有用。

  优缺点

  - **优点**: 理论上，选择最大的空闲块应该能减少碎片。
  - **缺点**: 在实践中，通常并不比其他算法好，查找性能也通常是最差的。

  提高查找性能

  维护一个按大小排序的空闲块列表来提高查找性能。

对于FF的空闲块排序（事实上是不需要排序的，因为无论如何排列，只要找到第一个可用的就行），对于BF、WF我采用了相同的排序算法——归并排序，这是一种稳定的且速度极快的排序算法，对于提升内存分配速率有很大作用。

- 内碎片

由于我们对于最小碎片长度有要求，一个空闲块的大小enough但是并不sufficient。比如在我们的实验中最小的碎片长度为10，一个空闲块大小为28，现在要给他分配一个20大小的proc，只能剩下8，如果被切出，不符合要求，因此28要一次性分配给proc。（此时proc的size是否应该保持为request_size（20）而不是28？我个人认为要记录为28，无论这8个空间是否被proc使用，但是其所“占用”的就是这么大，对于其他proc的影响也是这么大）

- 外碎片

  由于我们的多次申请空间释放空间导致的长度小于最小值的碎片。例如经过一系列行为，导致701—705的空间为free的，但是显然这部分空间永远无法被使用到。

  内存紧缩解决的就是这种碎片。

- 回收内存时。空闲块合并主要是通过insertion sort实现，根据空闲块的地址排序，如果空闲块的内存是连续的，那就将其合并为一个块。

代码实现：

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

#define PROCESS_NAME_LEN 32
#define MIN_SLICE 10
```

```c
#define DEFAULT_MEM_SIZE 1024
#define DEFAULT_MEM_START 0

#define MA_FF 1
#define MA_BF 2
#define MA_WF 3

#include "2.3.h"

struct free_block_type {
  int size;
  int start_addr;
  struct free_block_type *next;
};
struct free_block_type *free_block = NULL;

struct allocated_block {
  int pid;
  int size;
  int start_addr;
  char process_name[PROCESS_NAME_LEN];
  struct allocated_block *next;
};
struct allocated_block
*allocated_block_head = NULL;

int allocate_mem(struct allocated_block
*ab);
void kill_process();
```

```
struct allocated_block *find_process(int
pid);
void free_mem(struct allocated_block *ab);
void dispose(struct allocated_block
*free_ab);
void kill_block(struct allocated_block
*ab);
void display_mem_usage();
void set_mem_size();
void set_algorithm();
void display_menu();
void new_process();
void rearrange(int algorithm);
void sort_free_blocks();
void compact();
void memory_compaction();
void do_exit();
int mem_size = DEFAULT_MEM_SIZE;
int ma_algorithm = MA_FF;
static int pid = 0;
int flag = 0;

struct free_block_type *init_free_block(int
mem_size) {
  struct free_block_type *fb;
  fb = (struct free_block_type
*)malloc(sizeof(struct free_block_type));
  if (fb == NULL) {
    printf("No mem\n");
    return NULL;
```

```
  }
  fb->size = mem_size;
  fb->start_addr = DEFAULT_MEM_START;
  fb->next = NULL;
  return fb;
}

void set_mem_size() {
  int size;
  if (flag != 0) {
    printf("Cannot set memory size
again\n");
    return;
  }
  printf("set memory_size to: ");
  scanf("%d", &size);
  if (size > 0) {
    mem_size = size;
    free_block->size = mem_size;
  }
  flag = 1;
}

void set_algorithm() {
  int algorithm;
  printf("\t1 - First Fit\n");
  printf("\t2 - Best Fit \n");
  printf("\t3 - Worst Fit \n");
  scanf("%d", &algorithm);
```

```
  if (algorithm >= MA_FF && algorithm <=
MA_WF) {
    ma_algorithm = algorithm;
  }
  rearrange(ma_algorithm);
}

struct free_block_type *merge(struct
free_block_type *a,
                              struct
free_block_type *b, int criterion) {
  struct free_block_type dummy;
  struct free_block_type *current = &dummy;
  if (criterion == MA_BF) {
    while (a && b) {
      if (a->size < b->size) {
        current->next = a;
        a = a->next;
      } else {
        current->next = b;
        b = b->next;
      }
      current = current->next;
    }
    current->next = a ? a : b;
  } else if (criterion == MA_WF) {
    while (a && b) {
      if (a->size > b->size) {
        current->next = a;
        a = a->next;
```

```
      } else {
        current->next = b;
        b = b->next;
      }
      current = current->next;
    }
    current->next = a ? a : b;
  }
  return dummy.next;
}
struct free_block_type *mergeSort(struct
free_block_type *head, int criterion) {
  if (!head || !head->next) return head;

  struct free_block_type *a = head, *b =
head->next;
  while (b && b->next) {
    head = head->next;
    b = b->next->next;
  }
  b = head->next;
  head->next = NULL;

  return merge(mergeSort(a, criterion),
mergeSort(b, criterion), criterion);
}

void rearrange_FF() { return; }
void rearrange_BF() { free_block =
mergeSort(free_block, MA_BF); }
```

```c
void rearrange_WF() { free_block =
mergeSort(free_block, MA_WF); }
void rearrange(int algorithm) {
  switch (algorithm) {
    case MA_FF:
      rearrange_FF();
      break;
    case MA_BF:
      rearrange_BF();
      break;
    case MA_WF:
      rearrange_WF();
      break;
    default:
      printf("Invalid algorithm.\n");
  }
}

void sort_free_blocks() {
  // insertion sort
  struct free_block_type *current, *next,
*tmp, *pre;

  if (!free_block || !free_block->next)
return;

  current = free_block->next;
  free_block->next = NULL;

  while (current) {
```

```
    pre = NULL;
    next = current->next;

    tmp = free_block;
    while (tmp->next && tmp->next-
>start_addr < current->start_addr) {
      pre = tmp;
      tmp = tmp->next;
    }
    if (!pre) {
      current->next = free_block;
      free_block = current;
    } else {
      current->next = pre->next;
      pre->next = current;
    }
    current = next;
  }
}

int allocate_mem(struct allocated_block
*ab) {
  struct free_block_type *fb, *pre;
  int request_size = ab->size;
  fb = pre = free_block;
  int total_free_size = 0;
  // 根据当前算法在空闲分区链表中搜索合适空闲分区进
行分配，分配时注意以下情况：
  while (fb) {
```

```
    if (fb->size - request_size >=
MIN_SLICE) {
      //  1. 找到可满足空闲分区且分配后剩余空间足
够大，则分割
      ab->start_addr = fb->start_addr;
      ab->size = request_size;
      fb->start_addr += request_size;
      fb->size -= request_size;
      goto success;
    } else if (fb->size >= request_size) {
      //  2. 找到可满足空闲分区且但分配后剩余空间
比较小，则一起分配
      ab->start_addr = fb->start_addr;
      ab->size = fb->size;
      pre->next = fb->next;
      free(fb);
      goto success;
    } else {
      total_free_size += fb->size;
      pre = fb;
      fb = fb->next;
    }
  }
  if (total_free_size >= request_size) {
    memory_compaction();
    //  3.
    //  找不可满足需要的空闲分区但空闲分区之和能满
足需要，则采用内存紧缩技术，进行空闲分区的合并，然后
再分配
    fb = pre = free_block;
```

```
    while (fb) {
        if (fb->size - request_size >=
MIN_SLICE) {
            ab->start_addr = fb->start_addr;
            ab->size = request_size;
            fb->start_addr += request_size;
            fb->size -= request_size;
            goto success;
        } else if (fb->size >= request_size)
{
            ab->start_addr = fb->start_addr;
            ab->size = fb->size;
            pre->next = fb->next;
            free(fb);
            goto success;
        } else {
            pre = fb;
            fb = fb->next;
        }
    }
  }
  return -1;

success:
  rearrange(ma_algorithm);
  return 1;
  //  4. 在成功分配内存后，应保持空闲分区按照相应
算法有序
  //  5. 分配成功则返回1，否则返回-1
}
```

```
void memory_compaction() {
  // Step 1: 将所有已分配的块移动到内存的起始地址
  int new_start_addr = 0;
  struct allocated_block *cur =
allocated_block_head;
  while (cur) {
    cur->start_addr = new_start_addr;
    new_start_addr += cur->size;
    cur = cur->next;
  }
  // Step 2: 释放所有旧的空闲块
  struct free_block_type *fb = free_block;
  while (fb) {
    struct free_block_type *tmp = fb;
    fb = fb->next;
    free(tmp);
  }

  // Step 3: 创建一个新的空闲块，其起始地址为最后
一个已分配块的结束地址
  free_block = (struct free_block_type
*)malloc(sizeof(struct free_block_type));
  free_block->start_addr = new_start_addr;
  free_block->size = mem_size -
new_start_addr;
  free_block->next = NULL;
}

void new_process() {
  struct allocated_block *ab;
```

```c
  int size;
  int ret;
  ab = (struct allocated_block
*)malloc(sizeof(struct allocated_block));
  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);
  if ((ret == 1) && (allocated_block_head
== NULL)) {
    allocated_block_head = ab;
  } else if (ret == 1) {
    ab->next = allocated_block_head;
    allocated_block_head = ab;
  } else if (ret == -1) {
    printf("Allocation fail\n");
    free(ab);
  }
}

void kill_process() {
  struct allocated_block *ab;
  int pid;
```

```
  printf("Kill Process, pid=");
  scanf("%d", &pid);
  ab = find_process(pid);
  if (ab != NULL) {
    free_mem(ab); /*释放ab所表示的分配区*/
    dispose(ab);  /*释放ab数据结构节点*/
  }
}
void free_mem(struct allocated_block *ab) {
  int algorithm = ma_algorithm;
  struct free_block_type *fb;
  fb = (struct free_block_type
*)malloc(sizeof(struct free_block_type));
  if (!fb) {
    printf("malloc fail in free_mem\n");
    return;
  }
  fb->size = ab->size;
  fb->start_addr = ab->start_addr;
  fb->next = free_block;
  free_block = fb;
  sort_free_blocks();
  compact();
  rearrange(ma_algorithm);
  // 进行可能的合并，基本策略如下
  // 1. 将新释放的结点插入到空闲分区队列末尾 I
choose the head not end.
  // 2. 对空闲链表按照地址有序排列
  // 3. 检查并合并相邻的空闲分区
  // 4. 将空闲链表重新按照当前算法排序
```

```
}
void dispose(struct allocated_block
*free_ab) {
  struct allocated_block *pre, *ab;
  if (free_ab == allocated_block_head) { /*
如果要释放第一个节点*/
    allocated_block_head =
allocated_block_head->next;
    free(free_ab);
    return;
  }
  pre = allocated_block_head;
  ab = allocated_block_head->next;
  while (ab != free_ab) {
    pre = ab;
    ab = ab->next;
  }
  pre->next = ab->next;
  free(ab);
}

void compact() {
  struct free_block_type *fb = free_block;
  struct free_block_type *next = NULL;

  while (fb && fb->next) {
    next = fb->next;
    if (fb->start_addr + fb->size == next-
>start_addr) {
      fb->size += next->size;
```

```
      fb->next = next->next;
      free(next);
    } else {
      fb = fb->next;
    }
  }
}

struct allocated_block *find_process(int
pid) {
  struct allocated_block *ab =
allocated_block_head;
  while (ab != NULL) {
    if (ab->pid == pid) {
      return ab;
    }
    ab = ab->next;
  }
  return NULL;
}

void kill_block(struct allocated_block *ab)
{
  struct allocated_block *pre_ab =
allocated_block_head;
  if (ab == allocated_block_head) {
    allocated_block_head = ab->next;
    free(ab);
    return;
  }
```

```
  while (pre_ab) {
    if (pre_ab->next == ab) {
      pre_ab->next = ab->next;
      free(ab);
      return;
    }
    pre_ab = pre_ab->next;
  }
}

void display_mem_usage() {
  struct free_block_type *fbt = free_block;
  struct allocated_block *ab =
allocated_block_head;

  if (fbt == NULL) {
    printf("No free memory blocks\n");
    return;
  }

  printf("---------------------------------
-------------------------\n");
  printf("Free Memory:\n");
  printf("%20s %20s\n", "start_addr",
"size");
  while (fbt != NULL) {
    printf("%20d %20d\n", fbt->start_addr,
fbt->size);
    fbt = fbt->next;
  }
```

```
    printf("\nUsed Memory:\n");
    printf("%10s %20s %10s %10s\n", "PID",
"ProcessName", "start_addr", "size");
    while (ab != NULL) {
        printf("%10d %20s %10d %10d\n", ab-
>pid, ab->process_name, ab->start_addr,
               ab->size);
        ab = ab->next;
    }

    printf("------------------------------------
-------------------------\n");
}

int main() {
    char choice;
    pid = 0;
    free_block = init_free_block(mem_size);
 // 初始化空闲区
    while (1) {
        display_menu();
        fflush(stdin);
        choice = getchar();
        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;
    }
  }
}

void display_menu() {
  printf("\n");
  printf("1 - Set memory size
(default=%d)\n", DEFAULT_MEM_SIZE);
  printf("2 - Select memory allocation
algorithm\n");
  printf("3 - New process \n");
  printf("4 - Terminate a process \n");
```

```c
  printf("5 - Display memory usage \n");
  printf("0 - Exit\n");
}

void do_exit() {
  struct allocated_block *ab =
allocated_block_head;
  while (ab) {
    allocated_block_head = ab->next;
    free(ab);
    ab = allocated_block_head;
  }
  struct free_block_type *fb = free_block;
  while (fb) {
    free_block = fb->next;
    free(fb);
    fb = free_block;
  }
}
```