

操作系统(D)

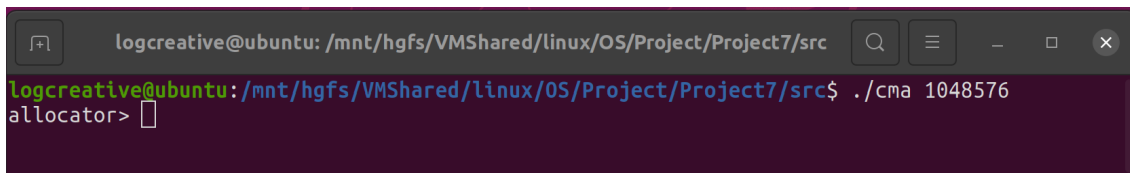
项目 7

Log Creative

2021 年 6 月 15 日

连续内存分配

一 主框架



```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Project7/src
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Project7/src$ ./cma 1048576
allocator> █
```

这里采用一个 alloc 的结构存储进程内存信息：起始、大小和进程名、下一个进程：

```
#define MAXLINE 30
int MAX;

struct alloc
{
    int start;
    int size;
    char process[MAXLINE];
    struct alloc* next;
};
```

采用一个单链表存储内存信息，定义空间起始位置：

```
// List
struct alloc* space_head;
```

获取内存大小：

```
int main(int argc, char* argv[]){
    if(argc==1){
        fprintf(stderr, "Please assign the initial amount of memory.\n");
        return -1;
    }
    MAX = atoi(argv[1]);
```

分配初始内存块：

```

space_head = (struct alloc*) malloc(sizeof(struct alloc));
space_head->start = 0;
space_head->size = MAX;
strcpy(space_head->process, "Unused");
space_head->next = NULL;

```

获取命令:

```

char command[MAXLINE];
do{
    fprintf(stdout, "allocator> ");
    fscanf(stdin, "%s", command);
    if(strcmp(command, "RQ")==0){
        char process[MAXLINE];
        int size;
        char flag[2];

        fscanf(stdin, "%s", process);
        fscanf(stdin, "%d", &size);
        fscanf(stdin, "%s", flag);

        if(request(process, size, flag))
            fprintf(stderr, "No sufficient memory!\n");

    } else if (strcmp(command, "RL")==0){
        char process[MAXLINE];
        fscanf(stdin, "%s", process);

        if(release(process))
            fprintf(stderr, "No such process to be released!\n");

    } else if (strcmp(command, "C")==0)
        compat();
    else if (strcmp(command, "STAT")==0)
        status();
    } while(strcmp(command, "X")!=0);
}

```

二 状态

```

void status() {
    struct alloc* tmp = space_head;
    while(tmp){
        char process_name[MAXLINE];
        if(strcmp(tmp->process, "Unused")==0) strcpy(process_name, "Unused");
        else {
            strcpy(process_name, "Process ");
            strcat(process_name, tmp->process);
        };
        fprintf(stdout, "Addresses [%d:%d] %s\n", tmp->start, tmp->start + tmp->size - 1, process_name);
        tmp = tmp->next;
    }
}

```

三 分配内存

请求两个进程，一个需要产生碎片，另一个是正好分配。

```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/OS/Project/Project7/src$ ./cma 1048576
allocator> RQ P0 40000 W
allocator> STAT
Addresses [0:39999] Process P0
Addresses [40000:1048575] Unused
allocator> RQ P1 1008576 W
allocator> STAT
Addresses [0:39999] Process P0
Addresses [40000:1048575] Process P1
allocator>
```

通过 flag 定义分配内存的方式。

```
int request(char* process, int size, char* flag){
    struct alloc* tmp = space_head;
    struct alloc* select = NULL;
    int besthole;
    switch(flag[0]){
        //...
    }
}
```

如果没有选择出孔，则会返回错误指标，终止分配。

```
allocator> STAT
Addresses [0:299999] Process P1
Addresses [300000:1048575] Unused
allocator> RQ P2 10000000 W
No sufficient memory!
allocator>
```

```
if(!select) return 1;
```

如果选择出了孔，如果这个孔的大小比所需要的内存大，则会产生内部碎片。

select new_alloc



```
if(select->size > size){
    struct alloc* new_alloc = (struct alloc*) malloc(sizeof(struct alloc));
    new_alloc->start = select->start + size;
    new_alloc->size = select->size - size;
    strcpy(new_alloc->process, "Unused");
    new_alloc->next = select->next;

    select->size = size;
    strcpy(select->process, process);
    select->next = new_alloc;
}
```

如果刚好，就直接赋予该空间该进程即可。

```

else if (select->size == size){
    strcpy(select->process, process);
}
return 0;
}

```

不同的分配方式:

首次适应 选择第一个满足分配空间的孔。

```

allocator> STAT
Addresses [0:9999] Process P1
Addresses [10000:39999] Unused
Addresses [40000:89999] Process P3
Addresses [90000:159999] Unused
Addresses [160000:219999] Process P5
Addresses [220000:1048575] Unused
allocator> RQ P8 5000 F
allocator> STAT
Addresses [0:9999] Process P1
Addresses [10000:14999] Process P8
Addresses [15000:39999] Unused
Addresses [40000:89999] Process P3
Addresses [90000:159999] Unused
Addresses [160000:219999] Process P5
Addresses [220000:1048575] Unused
allocator>

```

```

case 'F':
    // First-fit
    while(tmp){
        if(strcmp(tmp->process,"Unused")==0 && tmp->size >= size){
            select = tmp;
            break;
        }
        tmp = tmp -> next;
    }
    break;

```

最优适应 遍历所有的孔，选择能够使得碎片大小最小的孔。

```

allocator> STAT
Addresses [0:9999] Process P1
Addresses [10000:29999] Unused
Addresses [30000:79999] Process P3
Addresses [80000:139999] Unused
Addresses [140000:189999] Process P5
Addresses [190000:1048575] Unused
allocator> RQ P6 100 B
allocator> STAT
Addresses [0:9999] Process P1
Addresses [10000:10099] Process P6
Addresses [10100:29999] Unused
Addresses [30000:79999] Process P3
Addresses [80000:139999] Unused
Addresses [140000:189999] Process P5
Addresses [190000:1048575] Unused
allocator>

```

```

case 'B':
// Best-fit
besthole = MAX;
while(tmp){
    if(strcmp(tmp->process,"Unused")==0 && tmp->size >= size && tmp->size - size <
        besthole){
        select = tmp;
        besthole = tmp->size - size;
    }
    tmp = tmp->next;
}
break;

```

最差适应 遍历所有的孔，选择能够使碎片大小最大的孔。

The terminal window shows the state of a memory allocator. The first 'STAT' command shows memory blocks for processes P1, P6, P3, P5, and several 'Unused' blocks. A request for 200 units of memory (RQ P7 200 W) is made. The second 'STAT' command shows the state after allocation: process P7 now occupies a block of 190 units, and the 'Unused' block at [190200:1048575] remains.

```

logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:9999] Process P1
Addresses [10000:10099] Process P6
Addresses [10100:29999] Unused
Addresses [30000:79999] Process P3
Addresses [80000:139999] Unused
Addresses [140000:189999] Process P5
Addresses [190000:1048575] Unused
allocator> RQ P7 200 W
allocator> STAT
Addresses [0:9999] Process P1
Addresses [10000:10099] Process P6
Addresses [10100:29999] Unused
Addresses [30000:79999] Process P3
Addresses [80000:139999] Unused
Addresses [140000:189999] Process P5
Addresses [190000:190199] Process P7
Addresses [190200:1048575] Unused
allocator>

```

```

case 'W':
// Worst-fit
besthole = -1;
while(tmp){
    if(strcmp(tmp->process,"Unused")==0 && tmp->size >= size && tmp->size - size >
        besthole){
        select = tmp;
        besthole = tmp->size - size;
    }
    tmp = tmp -> next;
}
break;

```

四 释放内存

先单独考虑释放的进程是开头的进程。

```

int release(char *process){
    struct alloc* tmp = space_head;
    if(strcmp(tmp->process, process)==0){

        // x -> 0
    }
}

```

```
// X X -> 0 X
// X 0 -> 0
```

对于开头进程的释放有三种情况：

1. 如果开始的进程后面没有进程了，就直接置零。

```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:49999] Process P5
Addresses [50000:1048575] Unused
allocator> RL P5
allocator> STAT
Addresses [0:1048575] Unused
allocator>
```

```
int after = (tmp->next ? strcmp(tmp->next->process, "Unused") == 0 : 0);
```

2. 如果开头的进程后面是碎片，就需要将释放后的空间与该碎片合并。

```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:99] Process P6
Addresses [100:50099] Unused
Addresses [50100:100099] Process P5
Addresses [100100:100299] Process P7
Addresses [100300:1048575] Unused
allocator> RL P6
allocator> STAT
Addresses [0:50099] Unused
Addresses [50100:100099] Process P5
Addresses [100100:100299] Process P7
Addresses [100300:1048575] Unused
```

```
if (after){
    space_head = tmp->next;
    space_head->start -= tmp->size;
    space_head->size += tmp->size;
}
```

3. 如果开头的进程后面是进程，就直接置零。

```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:999] Process P1
Addresses [1000:1999] Process P2
Addresses [2000:1048575] Unused
allocator> RL P1
allocator> STAT
Addresses [0:999] Unused
Addresses [1000:1999] Process P2
Addresses [2000:1048575] Unused
allocator>
```

```
else {
    strcpy(space_head->process, "Unused");
}
return 0;
}
```

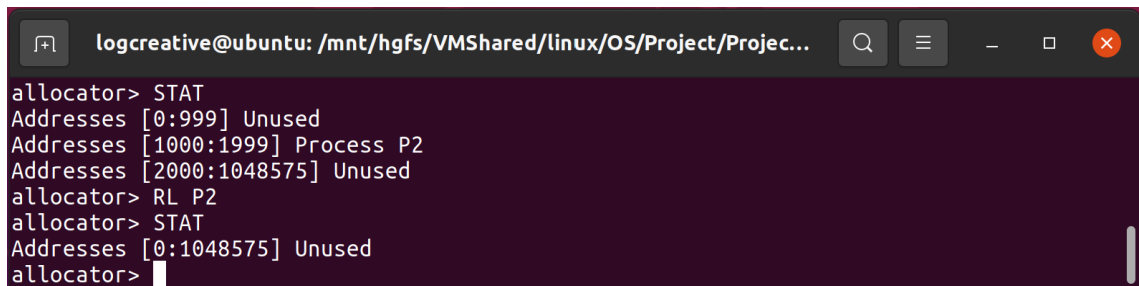
类似的，对于中间进程的释放，需要分成四种情况：

```
while(tmp->next){
    if(strcmp(tmp->next->process,process)==0){
        struct alloc* del = tmp->next;

        // 0 X 0 -> 0
        // X X 0 -> X 0
        // 0 X X -> 0 X
        // X X X -> X 0 X

        int before = strcmp(tmp->process,"Unused") == 0;
        int after = (tmp->next->next ? strcmp(tmp->next->next->process,"Unused") == 0 : 0);
```

1. 前后都为孔，就需要将三孔合一。



```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:999] Unused
Addresses [1000:1999] Process P2
Addresses [2000:1048575] Unused
allocator> RL P2
allocator> STAT
Addresses [0:1048575] Unused
allocator>
```

```
if(before && after){
    tmp->size += del->size + tmp->next->next->size;
    tmp->next = tmp->next->next;
}
```

2. 后为孔，后两个合一。

```
else if (after){
    tmp->next->next->start -= del->size;
    tmp->next->next->size += del->size;
    tmp->next = tmp->next->next;
}
```

3. 前为孔，前两个合一。

```
else if (before){
    tmp->size += del->size;
    tmp->next = tmp->next->next;
}
```

4. 都不为孔，直接置零。

```
else {
    strcpy(del->process,"Unused");
}
```

最后移动指标。如果遍历到最后都没有找到，就会返回未找到。

```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:1048575] Unused
allocator> RL P10
No such process to be released!
allocator>
```

```
        return 0;
    }
    tmp = tmp->next;
}
return 1;
}
```

五 紧缩

首先寻找尾部，并存储遍历的块次数 count。

```
void compat() {

    struct alloc* tail = space_head;
    int count = 1;
    while(tail->next){
        tail = tail->next;
        ++count;
    }

    // Move to tail
    // 0 0 X X X -> 0 X X X 0 -> X X X 0 0
    // 0 X X X X -> X X X X 0
    // X 0 X 0 X -> X X 0 X 0 -> X X X 0 0
}
```

紧缩共分两步：移动碎片到尾部、碎片合一。

对于移动碎片，分两种情形：

1. 头部是碎片。需要移动空间头 `space_head` 标记的位置，并遍历到尾部将所有的块开始位置前移，再将碎片挂载在尾部。

```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:299999] Unused
Addresses [300000:399999] Process P2
Addresses [400000:599999] Process P3
Addresses [600000:1048575] Unused
allocator> C
allocator> STAT
Addresses [0:99999] Process P2
Addresses [100000:299999] Process P3
Addresses [300000:1048575] Unused
allocator>
```

```
struct alloc* tmp = space_head;
while (tmp->next && strcmp(tmp->process, "Unused")==0){
    space_head = tmp->next;

    struct alloc* ch_tmp = space_head;
}
```



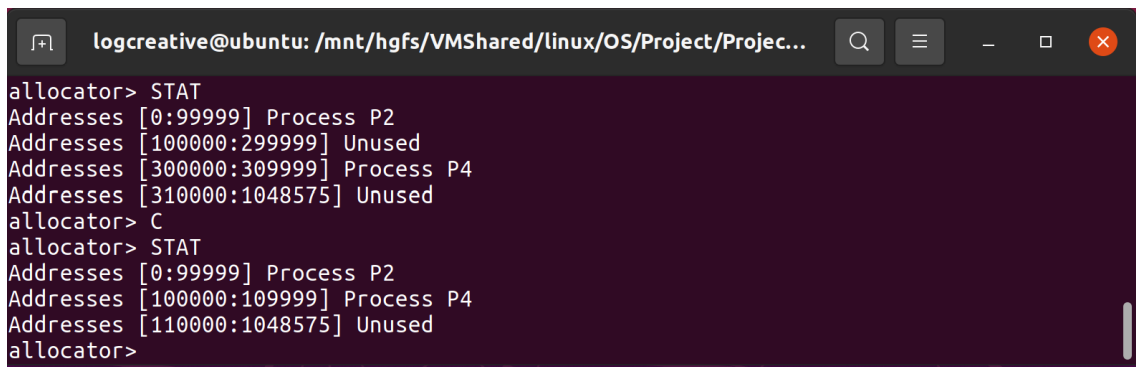
```

        while(ch_tmp){
            ch_tmp->start -= tmp->size;
            ch_tmp = ch_tmp->next;
        }

        tmp->next = tail->next;
        tmp->start = tail->start + tail->size;
        tail->next = tmp;
        tail = tmp;
        tmp = space_head;
        --count;
        if(!count) break;
    }

```

2. 中间某处是碎片。需要多计算一次移动次数，因为移动了一个指针，并移动了一个块。



```

logcreative@ubuntu: /mnt/hgfs/VMShared/linux/OS/Project/Projec...
allocator> STAT
Addresses [0:99999] Process P2
Addresses [100000:299999] Unused
Addresses [300000:309999] Process P4
Addresses [310000:1048575] Unused
allocator> C
allocator> STAT
Addresses [0:99999] Process P2
Addresses [100000:109999] Process P4
Addresses [110000:1048575] Unused
allocator>

```

```

while(count) {
    if(!tmp->next) break;
    if(tmp->next->next && strcmp(tmp->next->process,"Unused")==0){
        struct alloc* mover = tmp->next;
        tmp->next = tmp->next->next;

        struct alloc* ch_tmp = tmp->next;
        while(ch_tmp){
            ch_tmp->start -= mover->size;
            ch_tmp = ch_tmp->next;
        }

        mover->next = tail->next;
        mover->start = tail->start + tail->size;
        tail->next = mover;
        tail = mover;
        --count;
    }
    tmp = tmp->next;
    --count;
}

```

最后将所有的碎片紧缩为一个空余空间。

```

// Compat the space
while(tmp && tmp->next){
    tmp->size += tmp->next->size;

```

```
    tmp->next = tmp->next->next;
}
}
```

A 全部代码

Listing 1: [src/Makefile](#)

```
all:
    gcc -g cma.c -o cma
clean:
    rm cma
```

Listing 2: [src/cma.c](#)

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

#define MAXLINE 30
int MAX;

struct alloc
{
    int start;
    int size;
    char process[MAXLINE];
    struct alloc* next;
};

// List
struct alloc* space_head;

int request(char* process, int size, char* flag){
    struct alloc* tmp = space_head;
    struct alloc* select = NULL;
    int besthole;
    switch(flag[0]){
        case 'F':
            // First-fit
            while(tmp){
                if(strcmp(tmp->process,"Unused")==0 && tmp->size >= size){
                    select = tmp;
                    break;
                }
                tmp = tmp -> next;
            }
            break;
        case 'B':
            // Best-fit
            besthole = MAX;
            while(tmp){
                if(strcmp(tmp->process,"Unused")==0 && tmp->size >= size && tmp->size - size < besthole){
```

```

        select = tmp;
        besthole = tmp->size - size;
    }
    tmp = tmp->next;
}
break;
case 'W':
    // Worst-fit
    besthole = -1;
    while(tmp){
        if(strcmp(tmp->process,"Unused")==0 && tmp->size >= size && tmp->size - size > besthole){
            select = tmp;
            besthole = tmp->size - size;
        }
        tmp = tmp -> next;
    }
    break;
}
if(!select) return 1;
if(select->size > size){
    struct alloc* new_alloc = (struct alloc*) malloc(sizeof(struct alloc));
    new_alloc->start = select->start + size;
    new_alloc->size = select->size - size;
    strcpy(new_alloc->process,"Unused");
    new_alloc->next = select->next;

    select->size = size;
    strcpy(select->process, process);
    select->next = new_alloc;
} else if (select->size == size){
    strcpy(select->process, process);
}
return 0;
}

int release(char *process){
    struct alloc* tmp = space_head;
    if(strcmp(tmp->process, process)==0){

        // X -> 0
        // X X -> 0 X
        // X 0 -> 0

        int after = (tmp->next ? strcmp(tmp->next->process,"Unused") == 0 : 0);

        if (after){
            space_head = tmp->next;
            space_head->start -= tmp->size;
            space_head->size += tmp->size;
        } else {
            strcpy(space_head->process, "Unused");
        }

        return 0;
    }
    while(tmp->next){

```

```

if(strcmp(tmp->next->process,process)==0){
    struct alloc* del = tmp->next;

    // 0 X 0 -> 0
    // X X 0 -> X 0
    // 0 X X -> 0 X
    // X X X -> X 0 X

    int before = strcmp(tmp->process,"Unused") == 0;
    int after = (tmp->next->next ? strcmp(tmp->next->next->process,"Unused") == 0 : 0);

    if(before && after){
        tmp->size += del->size + tmp->next->next->size;
        tmp->next = tmp->next->next->next;
    } else if (after){
        tmp->next->next->start -= del->size;
        tmp->next->next->size += del->size;
        tmp->next = tmp->next->next;
    } else if (before){
        tmp->size += del->size;
        tmp->next = tmp->next->next;
    } else {
        strcpy(del->process,"Unused");
    }

    return 0;
}
tmp = tmp->next;
}
return 1;
}

void compat() {

    struct alloc* tail = space_head;
    int count = 1;
    while(tail->next){
        tail = tail->next;
        ++count;
    }

    // Move to tail
    // 0 0 X X X -> 0 X X X 0 -> X X X 0 0
    // 0 X X X X -> X X X X 0
    // X 0 X 0 X -> X X 0 X 0 -> X X X 0 0

    struct alloc* tmp = space_head;
    while (tmp->next && strcmp(tmp->process,"Unused")==0){
        space_head = tmp->next;

        struct alloc* ch_tmp = space_head;
        while(ch_tmp){
            ch_tmp->start -= tmp->size;
            ch_tmp = ch_tmp->next;
        }
    }
}

```

```

    tmp->next = tail->next;
    tmp->start = tail->start + tail->size;
    tail->next = tmp;
    tail = tmp;
    tmp = space_head;
    --count;
    if(!count) break;
}

while(count) {
    if(!tmp->next) break;
    if(tmp->next->next && strcmp(tmp->next->process,"Unused")==0){
        struct alloc* mover = tmp->next;
        tmp->next = tmp->next->next;

        struct alloc* ch_tmp = tmp->next;
        while(ch_tmp){
            ch_tmp->start -= mover->size;
            ch_tmp = ch_tmp->next;
        }

        mover->next = tail->next;
        mover->start = tail->start + tail->size;
        tail->next = mover;
        tail = mover;
        --count;
    }
    tmp = tmp->next;
    --count;
}

// Compat the space
while(tmp && tmp->next){
    tmp->size += tmp->next->size;
    tmp->next = tmp->next->next;
}
}

void status() {
    struct alloc* tmp = space_head;
    while(tmp){
        char process_name[MAXLINE];
        if(strcmp(tmp->process,"Unused")==0) strcpy(process_name,"Unused");
        else {
            strcpy(process_name, "Process ");
            strcat(process_name, tmp->process);
        };
        fprintf(stdout,"Addresses [%d:%d] %s\n", tmp->start, tmp->start + tmp->size - 1, process_name);
        tmp = tmp->next;
    }
}

int main(int argc, char* argv[]){
    // if(argc==1){
    //     fprintf(stderr, "Please assign the initial amount of memory.\n");
    //     return -1;

```

```

// }
// MAX = atoi(argv[1]);
MAX = 1048576;

space_head = (struct alloc*) malloc(sizeof(struct alloc));
space_head->start = 0;
space_head->size = MAX;
strcpy(space_head->process, "Unused");
space_head->next = NULL;

char command[MAXLINE];
do{
    fprintf(stdout, "allocator> ");
    fscanf(stdin, "%s", command);
    if(strcmp(command, "RQ")==0){
        char process[MAXLINE];
        int size;
        char flag[2];

        fscanf(stdin, "%s", process);
        fscanf(stdin, "%d", &size);
        fscanf(stdin, "%s", flag);

        if(request(process, size, flag))
            fprintf(stderr, "No sufficient memory!\n");

    } else if (strcmp(command, "RL")==0){
        char process[MAXLINE];
        fscanf(stdin, "%s", process);

        if(release(process))
            fprintf(stderr, "No such process to be released!\n");

    } else if (strcmp(command, "C")==0)
        compat();
    else if (strcmp(command, "STAT")==0)
        status();
} while(strcmp(command, "X")!=0);
}

```