Contiguous Memory Allocation

Overview

In Section 9.2, we presented different algorithms for contiguous memory allocation. This project will involve managing a contiguous region of memory of size MAX where addresses may range from 0...MAX−1. Your program must respond to four different requests:

1. Request for a contiguous block of memory
2. Release of a contiguous block of memory
3. Compact unused holes of memory into one single block
4. Report the regions of free and allocated memory

Program Structure

Memory Representation

Considering that a lot of `insert` and `delete` operations will be executed in the execution of the program， I implement memory blocks in the form of doubly-linked list as below

```
/* block of memory */

struct Block{
    char* name;
    int address;
    int size;
    struct Block* next;
    struct Block* previous;
};

block* head;


block* initBlock(char *name, int address, int size)
{
    block* tmp = malloc(sizeof(block));
    tmp->name = name;
    tmp->address = address;
    tmp->size = size;
    tmp->next = NULL;
    tmp->previous = NULL;
    return tmp;
}
```

```
/* report the regions of memory that are allocated and the regions that are unused */

void display()
{
    block* p = head->next;
    while(p)
    {
        if(strcmp(UNUSED,p->name) != 0)
        {
            printf("Addresses [%d,%d]\tProcess %s\n",p->address,p->address+p->size-1,p->name);
        }
        else
        {
            printf("Addresses [%d,%d]\tUnused\n",p->address,p->address+p->size-1);
        }
        p = p->next;
    }

}



void deleteBlock(block* p)
{
    p->previous->next = p->next;
    if(p->next) p->next->previous = p->previous;
    free(p);
}
```

Memory releasing

If a partition being released is adjacent to an existing hole, be sure to combine the two holes into a single hole.

```
bool release(char* name)
{
    block* p = head->next;
    bool flag = 0;
    while(p)
    {
        if(strcmp(p->name, name) == 0)
        {
            p->name = UNUSED;
            if(p->next != NULL && strcmp(p->next->name,UNUSED) == 0)
            {
                p->size += p->next->size;
                deleteBlock(p->next);
            }
            if(p->previous != head && strcmp(p->previous->name,UNUSED) == 0)
            {
                p->size += p->previous->size;
                p->address = p->previous->address;
                deleteBlock(p->previous);
            }
            flag = 1;
            break;
        }
        p = p->next;
    }
    return flag;
}
```

Memory allocation

- First Fit

```
bool requestF(char* name, int size)
{
    block* p = head->next;
    bool flag = 0;
    while(p)
    {
        if(strcmp(p->name,UNUSED) == 0 && p->size >= size)
        {
            allocation(p,name,size);
            flag = 1;
            break;
        }
        p = p->next;
    }
    return flag;
}
```

- Best Fit

```
bool requestB(char* name, int size)
{
    block* p = head->next;
    block* tmp;
    bool flag = 0;
    while(p)
    {
        if(strcmp(p->name,UNUSED) == 0 && p->size >= size)
        {
            if(!tmp)    tmp = p;
            else
            {
                if(tmp->size > p->size)
                    tmp = p;
            }

            flag = 1;
        }
        p = p->next;
    }

    allocation(tmp,name,size);
    return flag;
}
```

- Worst Fit

```
bool requestW(char* name, int size)
{
    block* p = head->next;
    block* tmp;
    bool flag = 0;
    while(p)
    {
        if(strcmp(p->name,UNUSED) == 0 && p->size >= size)
        {
            if(!tmp)    tmp = p;
            else
            {
                if(tmp->size < p->size)
                    tmp = p;
            }

            flag = 1;
        }
        p = p->next;
    }

    allocation(tmp,name,size);
    return flag;
}
```

Compact

This command will compact unused holes of memory into one region. Finally, the STAT command for reporting the status of memory is entered.

```
void compact()
{
    block* p = head->next;
    int address = 0;
    int sum = 0;
    int cnt = 0;

    while(p->next)
    {
        if(strcmp(p->name,UNUSED) == 0)
        {
            sum += p->size;
            p = p->next;
            deleteBlock(p->previous);
        }
        else
        {
            p->address = address;
            address += p->size;
            p = p->next;
        }
    }
    if(strcmp(p->name,UNUSED) == 0)
    {
        p->address = address;
        p->size += sum;
    }
    else
    {
        p->address = address;
        block* tmp = initBlock(UNUSED,address, sum);
        p->next = tmp;
        tmp->previous = p;
    }

}
```

Display

Print out the memory status composed of allocated memory range and memory fragment name.

```
void display()
{
    block* p = head->next;
    while(p)
    {
        if(strcmp(UNUSED,p->name) != 0)
        {
            printf("Addresses [%d,%d]\tProcess %s\n",p->address,p->address+p->size-1,p->name);
        }
        else
        {
            printf("Addresses [%d,%d]\tUnused\n",p->address,p->address+p->size-1);
        }
        p = p->next;
    }

}
```

User Interface

```
int main(int argc, char* argv[])
{
    if(argc == 1)
    {
        printf("Error Input!");
        return -1;
    }


    int size = atoi(argv[1]);
    char* file;
    bool flag = 0; // flag being 1 means commands are from files.
    if(argc >= 3)
    {
        flag = 1;
        file = argv[2];
    }
    /* initialization of the head node and first memory block */
    head = malloc(sizeof(block));
    block* tmp = initBlock(UNUSED,0,size);
    head->next = tmp;
    tmp->previous = head;
    char cmd[80];
    char* cpy;
    char* token;
    char* name;
    char* mode;


    FILE* fp;
    if(flag)
        fp = fopen(file,"r");
    while(1)
    {

        if(flag)
            fgets(cmd,80,fp);
        else
        {
            printf("allocator>");
            fgets(cmd, 80, stdin);
        }

        cpy = strdup(cmd);
        if(cmd[0] == 'C')
        {
            compact();
        }
        else if(cmd[0] == 'R' && cmd[1] == 'L')
        {
            token = strsep(&cpy,DELIM);
            token = strsep(&cpy,DELIM);
            //printf("%s",token);
            if(!release(token))
            {
                printf("Process not found.\n");
            }
        }
        else if(cmd[0] == 'R' && cmd[1] == 'Q')
        {
            token = strsep(&cpy,DELIM);
            token = strsep(&cpy,DELIM);
            name = token;
            token = strsep(&cpy,DELIM);
            int space = atoi(token);
            token = strsep(&cpy,DELIM);
            mode = token;
            //printf("%s,%d,%s",name,space,mode);
            if(mode[0] == 'W')
            {
                if(!requestW(name,space))
                    printf("No sufficient space");
            }
            else if(mode[0] == 'B')
            {
                if(!requestB(name,space))
                    printf("No sufficient space");
            }
            else if(mode[0] == 'F')
            {
                if(!requestF(name,space))
                    printf("No sufficient space");
            }
        }
        else if(strncmp(cmd,"STAT",4) == 0)
        {
            display();
        }
        else if(strncmp(cmd,"exit",4) == 0)
        {
            break;
        }

    }

    return 0;
}
```

How to run the program

To compile the files, enter `gcc -o test allocation.c`

To run it, enter `./test 1000`