# C语言模拟实现内存管理

李世旺

2016年5月24日

**摘要**

内存作为计算机的一项重要资源，应该要合理的进行管理。本文就连续分配存储管理方式中的各种动态分区分配方式进行比较。并用C语言进行模拟实验，测算不同分配方式的差异。

# 目录

## 1 实验目的

用C语言模拟实现内存的动态分区分配管理

## 2 实验原理与方案

本实验中的动态分区分配又称为可变分区分配，它是根据进程的实际需要，动态的为之分配内存空间。在实现动态分区分配时将涉及到分区分配中所用的数据结构、分区分配算法和分区的分配与回收操作这样三方面的问题。

### 2.1 数据结构

空闲分区表：用一个带头结点的单链表来存储，结构体的成员包括分区起始地址、分区大小、分区空闲时刻、下一个空闲分区地址。

繁忙分区表：类似空闲分区表，用一个带头结点的单链表来存储，结构体的成员包括分区起始地址、分区大小、分区空闲时刻、下一个繁忙分区地址。

### 2.2 分区分配算法

首次适应法：每次从空闲分区表的头部开始，找到可以分配的分区就分配。

循环首次适应法：分配分区的时候，从上次分配的地方开始往后面查找，到后面没有找到时，再从头开始查找。

### 2.3 分区分配操作

分配内存：如果请求的分区与目标分区相差不大，则直接分配出去，否则划分成两部分。同时还要修改空闲分区表和繁忙分区表。

回收内存：在分配分区的时候，繁忙分区成员的分区空闲时刻已经设置好了，时间触发回收操作。根据回收区的首地址来查找它在空闲分区的插入位置。如果回收区与前一个空闲分区相连，或者与后一个空闲分区相连，此时应该尽可能扩大分区。

## 3 执行结果与分析

对于事先给定的10000个任务数据，限制内存为 2147483648，首次适应法用时：641. 循环首次适应法用时：642. 从结果分析来看，分配算法之间没有明显差异。

事实上，首次适应法会使得低地址部分的空间细分成小碎片，而为了找到大分区，需要费力地往后面查找。

循环首次适应法或许解决了分区切割不均匀的问题，但是这样可能会缺乏大的分区。

像最坏适应算法，它的分配最为均匀了，对于中小规模的任务，产生碎片的可能性最小。

和最坏适应算法相反的最佳适应算法，则会留下许许多多的碎片。

## 4 详细代码

```
%\begin{verbatim}
#include<stdio.h>
#include<stdlib.h>
#define GAP 5
typedef struct cnode
{
    unsigned int size;
    unsigned int begin;
    unsigned int finish;
    struct cnode * next;
} charnode;
unsigned int currenttime = 0;
charnode *ProcessList;
charnode *CycleFirstFit;
charnode *allocate(charnode *List, int runtime, int memsize)
{
```

```
charnode *q, *p, *r, *last;
p = List->next;
if (NULL == p)
{
    return NULL;
}
last = List;
while (NULL != p)
{
    if (p->size < memsize)
    {
        //      unsigned int a = 5,b = 6;
        //      printf("\n%d",a>b);
        //      printf("\n%d",a-b>0);
        last = p;
        p = p->next;
        continue;
    }
    else if (p->size - memsize <= GAP)
    {
        ////////////////////////////////////
        ///////////////////////////////////
        last->next = p->next;
        ////////////////////////////////
        p->finish = currenttime + runtime;
        r = ProcessList->next;
        ProcessList->next = p;
        p->next = r;
        return p;
    }
    else
```

```
        {
            ///////////////////////
            q = (charnode *) malloc(sizeof(charnode));
            q->begin = p->begin;
            q->size = memsize;
            q->finish = currenttime + runtime;
            ///////////////////////
            r = ProcessList->next;
            ProcessList->next = q;
            q->next = r;
            ///////////////////////
            p->begin = p->begin + memsize;
            p->size = p->size - memsize;
            return q;
        }
    }           //;
    return NULL;
}
charnode *allocate2(charnode *List, int runtime, int memsize)
{
    int cycleflag = 1;
    charnode *q, *p, *r, *last;
    p = CycleFirstFit->next;
    if (NULL == p)
    {
        cycleflag = 1;
    }
    last = List;
    while (NULL != p)
    {
        if (p->size < memsize)
```

```
{
    //      unsigned int a = 5,b = 6;
    //      printf("\n%d",a>b);
    //      printf("\n%d",a-b>0);
    last = p;
    p = p->next;
    continue;
}
else if (p->size - memsize <= GAP)
{
    //////////////////////////////////
    /////////////////////////////////
    last->next = p->next;
    //////////////////////////////
    p->finish = currenttime + runtime;
    r = ProcessList->next;
    ProcessList->next = p;
    p->next = r;
    CycleFirstFit = last;
    cycleflag = 0;
    return p;
}
else
{
    /////////////////////
    q = (charnode *) malloc(sizeof(charnode));
    q->begin = p->begin;
    q->size = memsize;
    q->finish = currenttime + runtime;
    /////////////////////
    r = ProcessList->next;
```

```
            ProcessList->next = q;
            q->next = r;
            ////////////////////////
            p->begin = p->begin + memsize;
            p->size = p->size - memsize;
            CycleFirstFit = p;
            cycleflag = 0;
            return q;
        }
    }           //;
    if(!cycleflag)return List;
    ////////////////////////////////////////////////////////////
    p = List->next;
    if (NULL == p)
    {
        return NULL;
    }
    last = List;
    while (NULL != p)
    {
        if (p->size < memsize)
        {
            //      unsigned int a = 5,b = 6;
            //      printf("\n%d",a>b);
            //      printf("\n%d",a-b>0);
            last = p;
            p = p->next;
            continue;
        }
        else if (p->size - memsize <= GAP)
        {
            ////////////////////////////////////
            //////////////////////////////////
            last->next = p->next;
            ///////////////////////////////
            p->finish = currenttime + runtime;
            r = ProcessList->next;
            ProcessList->next = p;
            p->next = r;
            CycleFirstFit = last;
            return p;
        }
        else
        {
            ///////////////////////
            q = (charnode *) malloc(sizeof(charnode));
            q->begin = p->begin;
            q->size = memsize;
            q->finish = currenttime + runtime;
            ///////////////////////
            r = ProcessList->next;
            ProcessList->next = q;
            q->next = r;
            ///////////////////////
            p->begin = p->begin + memsize;
            p->size = p->size - memsize;
            CycleFirstFit = p;
            return q;
        }
    }               //;
    return NULL;
}
```

```
void deallocate(charnode *List, charnode *node)
{
    ///////////////////////////////////////////
    charnode *p = List->next, *q, *r, *last;
    if (NULL == p)
    {
        ////////////////////////
        List->next = node;
        node->next = p;
        return;
    }
    last = List;
    while (NULL != p)
    {
        q = p->next;
        if (p->begin + p->size == node->begin)
        {
            if (NULL == q)
            {
                //////////    node insert after p    /////////////
                p->size += node->size;
                return;
            }
            if (node->begin + node->size == q->begin)
            {
                //////////    node insert after p    /////////////
                r = q->next;
                p->next = r;
                p->size += (node->size + q->size);
                return;
        }
        else
        {
            //////////    node insert after p    /////////////
            p->size += node->size;
            return;
        }
    }
    else if (p->begin < node->begin)
    {
        last = p;
        p = p->next;
        continue;
    }
    else
    {
        //////////    node insert before p    /////////////
        if (node->begin + node->size == p->begin)
        {
            last->next = node;
            node->next = q;
            node->size += p->size;
            return;
        }
        else
        {
            last->next = node;
            node->next = p;
            return;
        }
    }
    }               // node insert at the end of the list
    last->next = node;
    node->next = p;
    return;
}
int main(int argc, char** argv)
{
    unsigned int i;
    unsigned int num_of_wait;
    unsigned int max_memory;
    unsigned int num_of_line;
    FILE *fp;
    if (2 != argc)
        printf("\nTips: a.exe data.txt");
    fp = fopen(argv[1], "rt");
    fscanf(fp, "%u", &num_of_line);
    fscanf(fp, "%u", &max_memory);
    unsigned int cursor = 0;
    unsigned int triggercursor = 0;
    unsigned int trigger[num_of_line];
    unsigned int visited[num_of_line];
    unsigned int arraive[num_of_line];
    unsigned int runtime[num_of_line];
    unsigned int memsize[num_of_line];
    printf("\n line of data : %u\n limit of memory : %u", num_of_line, max_n
//      unsigned int a = 5,b = 6;
//      printf("\n%d",a>b);
//      printf("\n%d",a-b>0);
//      return 0;
    trigger[triggercursor] = 0;
    for (i = 0; i < num_of_line; i++)
```

```
{
    fscanf(fp, "%u%u%u", &arraive[i], &runtime[i], &memsize[i]);
    visited[i] = 0;
    if (trigger[triggercursor] != arraive[i])
    {
        trigger[++triggercursor] = arraive[i];
    }
}
trigger[++triggercursor] = 100000;
charnode *FreeList, *node;
ProcessList = (charnode *) malloc(sizeof(charnode));
ProcessList->next = NULL;
FreeList = (charnode *) malloc(sizeof(charnode));
FreeList->next = NULL;
////////////////////////
node = (charnode *) malloc(sizeof(charnode));
node->begin = 0;
node->size = max_memory;
node->finish = 0;
deallocate(FreeList, node);
CycleFirstFit = FreeList;
//////////////////////////
charnode* (*all)(charnode *List, int runtime, int memsize);
printf("\ninput␣1␣for␣First␣Fit␣or␣2␣for␣Next␣Fit␣:␣-->");
int exitflag = 1;
while(exitflag)
{
    char ch = getchar();
    switch(ch)
    {
        case '1':
            {
                all = allocate;
                exitflag = 0;
                break;
            }
            case '2':
            {
                all = allocate2;
                exitflag = 0;
                break;
            }
            default :
            {
                printf("\ninput 1 for First Fit or 2 for Next Fit : -->");
                break;
            }
        }

    }
    printf("\n-----wait for a minute ...");
    while (1)
    {
        //////// Time Machine Real Trigger : when process arraived
        //////// Time Machine Trigger : when process finished
        num_of_wait = 0;
        for (i = 0; i < num_of_line && arraive[i] <= currenttime; i++)
        {
            if (!visited[i])
            {
                num_of_wait++;
            }
```

```
}
if (!num_of_wait)
{
    if (num_of_line - 1 <= i)
    {
        charnode *p = ProcessList->next;
        int maxtime = 0;
        if (NULL == p)
        {
            printf("\n-----well done! -----");
            printf("\n the answer : %u", currenttime);
            return 0;
        }
        while (NULL != p)
        {
            if (maxtime < p->finish)
            {
                maxtime = p->finish;
            }
            p = p->next;
        }
        printf("\n-----well done! -----");
        printf("\n the answer : -> %u", maxtime);
        return 0;
    }
    else
    {
        currenttime = arraive[i + 1];
    }
}
///////////////////////////
```

```
for (i = 0; i < num_of_line && arraive[i] <= currenttime; i++)
{
    if (!visited[i])
    {
        if (NULL != all(FreeList, runtime[i], memsize[i]))
            visited[i] = 1;
    }
}
//////////////////////////
charnode *p = ProcessList->next;
charnode *last = ProcessList;
charnode *r, *s;
int mintime = 1000000;
while (NULL != p)
{
    if (mintime > p->finish)
    {
        mintime = p->finish;
        node = last;
    }
    {
        last = p;
        p = p->next;
    }
}
/////////////////////////////////////////////
if (NULL != ProcessList->next)
{
    // |------------->
    // |    --->
    // |----------->
```

```
            r = node->next;
            s = r->next;
//              printf("\n%u", r->finish);
            if (trigger[cursor] < r->finish && currenttime <= trigger[curso
            {
                currenttime = trigger[cursor++];
                continue;
            }
            // confirm to recycle
            node->next = s;
            deallocate(FreeList, r);
            currenttime = r->finish;
            cursor = 0;
            while(trigger[cursor] < currenttime && cursor < triggercursor)
            {
                cursor++;
            }
        }
        ////////////////////////////////////
    }
}


%\end{verbatim}
```