操作系统——内存管理项目

1854062 许之博

开发环境

- 开发平台：Windows10
- 开发工具：WebStorm2021.1.2
- 使用语言：HTML5，CSS3，JavaScript
- 主要开发依赖：core-js3.6.5，element-ui2.15.2，vue2.6.11，vue-router3.5.1

1. 动态分区分配

1.0 操作说明

1. 点击打开dist文件夹中的index.html文件，点击分区管理
2. 可以先通过拖动滑块来调整内存空间大小，默认为640，并选择分配算法(默认为首次适应算法)
3. 连续点击下一步，每一步都可以看到内存空间中的状态变化
4. 执行完所以的作业，可以点击重置重新开始，在中间过程时，也可以随时重置

1.1 背景分析

内存管理是操作系统设计中的重要组成部分，操作系统对内存的划分和动态分配，就是内存管理的概念。为一个用户程序分配一个连续的内存空间的方法称为内存的连续分配管理方式，连续分配方式主要包括单一连续分配，固定分区分配和动态分区分配。

动态内存分配又称为可变分区分配，该方法不预先划分内存，而是在进程装入内存时，根据进程的大小动态地建立分区，并使分区的大小正好适合进程的需要。动态分区算法主要有首次适应算法，最佳适应算法，最坏适应算法，邻近适应算法等方法。本次实验选用首次适应算法和最佳适应算法两种算法。

- 首次适应算法：空闲分区以地址递增的次序链接。分配内存时顺序查找，找到能满足要求的第一个空闲分区。
- 最佳适应算法：空闲分区按容量递增的方式形成分区链，找到第一个能满足要求的空闲分区。

1.2 实现过程

1.2.1 首次适应算法

在描述内存中的作业状态时，采用对象数组的方法，每个对象包含内存中的一个作业的基本信息，包括作业编号，在内存空间中的起点以及终点。

当一个新的作业申请进入内存时，进行如下操作：

1. 对内存状态数组中的所有作业按起点位置升序排列
2. 若内存中本来就为空，并且内存空间大小小于作业大小，直接插入内存状态数组中，结束；若否，则执行接下来的操作。
3. 从头遍历内存状态数组，判断相邻两个作业区间之间的空白区域是否大于等于新作业的大小，若符合，则直接插入。同时还要考虑内存空间的首末端情况，判断0到第一个作业区间间是否有空白区域，以及最后一个作业到内存空间末端是否有空白。

当作业申请释放时，直接从内存空间中找到该作业，并将其删除。

```
firstFit(task)
   {
     let memoryState2=this.sortByKey(this.memoryState,"begin");
     if(task.choose==1)//申请
     { //如果本来内存就为空
       if(this.memoryState.length==0){
        if(this.memorySpace<task.memorySize) return;
        this.memoryState.push({
          assignmentID:task.assignmentID,
          begin:0,
          end:task.memorySize-1,
          size:task.memorySize
        });
        return;
       }
       if(memoryState2[0].begin>0)//如果首个元素初始位置大于0
       {
         if(task.memorySize<memoryState2[0].begin) {
           this.memoryState.push({
             assignmentID:task.assignmentID,
             begin:0,
             end:task.memorySize-1,
             size:task.memorySize});
           return;
         }
         let i=1;
         for(i=1;i<memoryState2.length-1;i++)
         {
           if((memoryState2[i+1].begin-memoryState2[i].end-1)>task.memorySize)
           {
             this.memoryState.push({
               assignmentID:task.assignmentID,
               begin:memoryState2[i].end+1,
               end:memoryState2[i].end+task.memorySize,
               size:task.memorySize});
              // alert('分配成功');
             return;
           }
         }
         if((this.memorySpace-memoryState2[i].end-1)>task.memorySize)
         {
           this.memoryState.push({
             assignmentID:task.assignmentID,
             begin:memoryState2[i].end+1,
             end:memoryState2[i].end+task.memorySize,
             size:task.memorySize});
            // alert("success");
             return;
         }
       }
       else//从头开始找
       {
          let i=0;
          for(i=0;i<memoryState2.length-1;i++)
          {
            if((memoryState2[i+1].begin-memoryState2[i].end-1)>task.memorySize)
            {
              this.memoryState.push({
```

```
                assignmentID:task.assignmentID,
                begin:memoryState2[i].end+1,
                end:memoryState2[i].end+task.memorySize,
                size:task.memorySize});
             // alert('分配成功');
              return;
            }
          }
          if((this.memorySpace-memoryState2[i].end-1)>task.memorySize)
          {
            this.memoryState.push({
              assignmentID:task.assignmentID,
              begin:memoryState2[i].end+1,
              end:memoryState2[i].end+task.memorySize,
              size:task.memorySize});
            return;
          }
        }
        return;
      }
      else//释放内存
      {
        let index = this.memoryState.findIndex(item =>{
          if(item.assignmentID==task.assignmentID){
            return true
          }
        })
        this.memoryState.splice(index,1)
      }
    }
```

1.2.2 最佳适应算法

除了1.2.1内存对象数组的概念，这里又引入了空白区域的概念，也用一个对象数组表示，每一个对象都是内存中的没有作业存在的空白区间，包括起点，终点，大小这些信息。

当一个新的作业申请进入内存时，进行如下操作：

1. 将空白区间数组按大小进行升序排列
2. 从左向右遍历空白区间，找到第一个满足大小大于等于新作业大小的空白区间a，将新作业插入内存对象数组，起点为该空白区间a的起点。
3. 计算新的空白区间b，新空白区间的起点为新插入的作业的终点+1，终点为空白区间a的终点。从空白区间数组中删除空白区间a，插入新的空白区间b

当作业申请释放，进行如下操作：

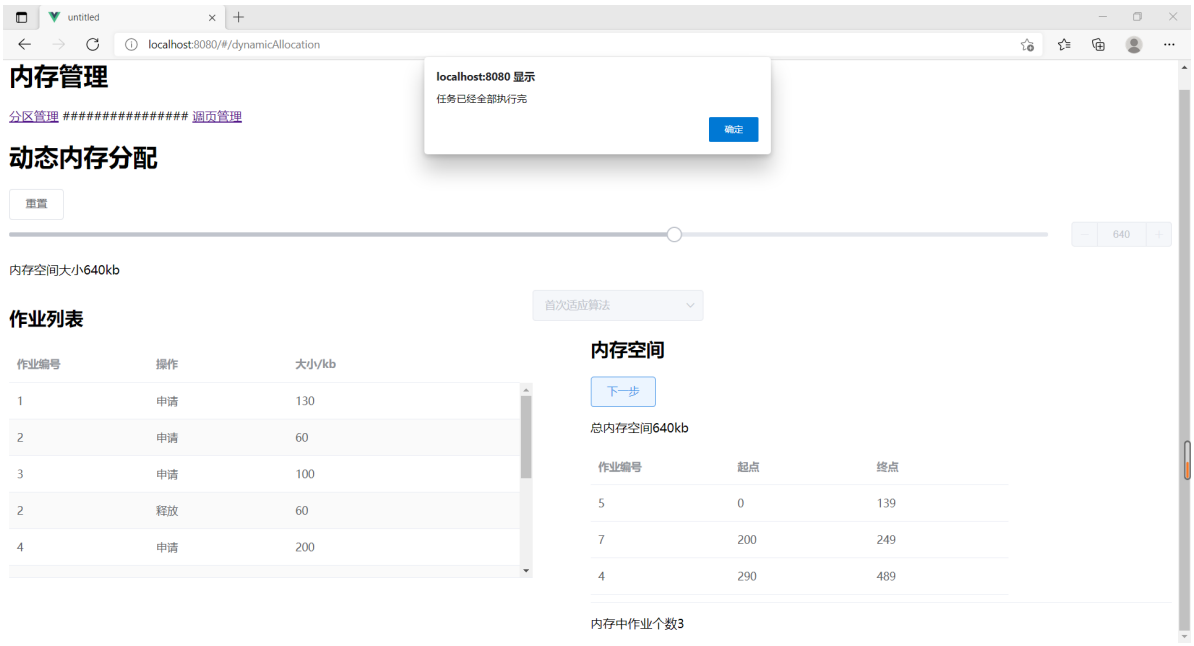
1. 在内存空间数组中找到该作业，根据该作业在内存空间中的位置新建一个空白区间c，该空白区间的起点，终点以及大小都与该作业相同
2. 从内存空间数组中删除待删除作业
3. 将空白区间c插入空白区间数组，然后将空白区间数组按空白区间的起点升序排列
4. 从头遍历空白区间数组，若遍历到位置i(从0开始），判断第i+1个空白区间与第i+2个空白区间是否相连，即前者的终点是否恰好为后者的起点-1。

   若相连，则新建空白区间f，f为二者的合并，然后从空白区间数组中删除相连的两个空白区间，插入新的空白区间f，再将空白区间数组按照起点位置升序排列，然后继续从数组的位置i遍历；若不相连，则继续遍历。

```
bestFit(task)
    {
      if(task.choose==1)
      {
        for(let i=0;i<this.blankSpace.length;i++)
        {
          if(task.memorySize<(this.blankSpace[i].size))
          {
            //分配
            this.memoryState.push({
              assignmentID:task.assignmentID,
              begin:this.blankSpace[i].begin,
              end:this.blankSpace[i].begin+task.memorySize-1,
              size:task.memorySize});
            //重载空白区
            //新建空白区
            let newBlank={
              begin:this.blankSpace[i].begin+task.memorySize,
              end:this.blankSpace[i].end,
              size:this.blankSpace[i].end-this.blankSpace[i].begin-
task.memorySize+1
            }
            //再删除原空白区
            this.blankSpace.splice(i,1);
            //插入新空白区
            this.blankSpace.push(newBlank);
            //重新排序
            this.blankSpace=this.sortByKey(this.blankSpace,"size");
            return;
          }
        }
        return;
      }
      else
      { //删除任务
        let index = this.memoryState.findIndex(item =>{
          if(item.assignmentID==task.assignmentID){
            return true
          }
        });
        let taskInMemory=this.memoryState[index];
        this.memoryState.splice(index,1);//删除进程
        this.blankSpace.push(
            {
              begin:taskInMemory.begin,
              end:taskInMemory.end,
              size:taskInMemory.size
            }
        );
        //连续的空白区进行合并 关键点
        //先排序
        this.blankSpace=this.sortByKey(this.blankSpace,"begin");
        for(let i=0;i<this.blankSpace.length-1;i++) {
          if(this.blankSpace[i].end==(this.blankSpace[i+1].begin-1)){
            let newBlank={
              begin:this.blankSpace[i].begin,
              end:this.blankSpace[i+1].end,
```

```
                size:this.blankSpace[i].size+this.blankSpace[i+1].size
            }
            this.blankSpace.splice(i,2);
            this.blankSpace.push(newBlank);
            this.blankSpace=this.sortByKey(this.blankSpace,"begin");
            i--;
          }
        }
        this.blankSpace=this.sortByKey(this.blankSpace,"size");
      }
    }
```

1.3 效果截图



2. 请求分页分配

2.0 操作说明

1. 点击打开dist文件夹中的index.html，点击调页管理
2. 首先必须要点击确定指令条数和内存块数，可在点击之前拖动滑块调整，指令条数默认为320，内存块数默认为4，选择页面置换算法，默认为LRU
3. 点击执行1步，也可以通过滑块调整每次执行的步数，在执行过程中，可以实时看到待执行指令序列，执行过程和内存块状态变化，以及缺页率
4. 运行过程中可以随时重置

注：有时确定指令条数和内存块数按钮与重置按钮可能要多按几次才能生效，但不影响整体运行

2.1 背景分析

程序的局部性原理：

- 时间局部性：程序中的某一条指令一旦执行，不久后该指令可能再次执行，某数据被访问后可能被再次访问。
- 空间局部性：一旦程序访问了某个存储单元，在不久后，其附近的存储单元也将被访问，即程序在一段时间内所访问的地址，可能集中在一定范围内

虚拟内存：基于局部性原理，在程序装入时，将程序的一部分装入内存，一部分留在外存，程序执行过程中，程序所访问信息不在内存时，将所需部分调入内存，将暂时不用的调出内存。这样，系统似乎为用户提供了一个比实际内存大的多的存储器。虚拟内存的实现需要建立在离散分配的内存管理方式的基础上。虚拟内存的实现有以下三种方式

- 请求分页存储管理
- 请求分段存储管理
- 请求段页式存储管理

本实验中选取请求分页存储管理。进程运行时，若其访问的页面不在内存中而需要将其调入，但内存已经无空闲空间时，需要对页面进行置换。常规的页面置换算法有以下四种，最佳置换算法(OPT)，先进先出算法(FIFO)，最近最久未使用算法(LRU)，时钟(CLOCK)置换算法。本实验中选用FIFO和LRU算法。

- 先进先出算法(FIFO)：优先淘汰最早进入内存的页面，即在内存中驻留最久的页面
- 最近最久未使用算法(LRU)：选择最长时间没访问过的页面进行淘汰

2.2 实现过程

本题中主要使用了待执行指令数组，以及内存块数组，每个内存块可以存放一个页。内存块数组也是一个对象数组，每个对象包括如下信息：块号，存放的页号，进入内存的时间(getInTime)，该页上一次被访问的时间(lastVisitTime)。待执行指令数组是一个对象数组，每个对象包括如下信息：指令编号，所在页号，地址。

从头遍历待执行指令序列：

- 若该指令所在块已经在内存中，则不进行页面置换，但内存中该页的lastVisitTime改为当前的时间（clock)，未发生缺页。
- 若该指令所在块不在内存中，分为以下两种情况：若内存中仍有块没被页占有，则将该指令所在块调入内存，发生缺页，但不发生置换。调入页的getInTime，lastVisitTime都为clock。若内存中无空间，则既发生缺页也要进行置换，置换采用下面的置换算法。

2.2.1 先进先出页面置换算法(FIFO)

从头遍历内存块数组，找出getInTime最小(也就是最早进入内存的页面)的内存块，将将要置换的页面换入该内存块，并修改getInTime和lastVisitTime。

```
alFIFO()//根据getInTime
    {
      let minIndex=0;
      for(let i=0;i<this.blockNum;i++)
      {
        if(this.blockList[minIndex].getInTime>this.blockList[i].getInTime)
        {
          minIndex=i;
        }
      }
      this.blockList[minIndex].pageID=this.orderList[0].pageID;
      this.blockList[minIndex].lastVisitTime=this.clock;
      this.blockList[minIndex].getInTime=this.clock;
      //alert("成功换页");
    }
```

2.2.2 最近最久未使用置换算法(LRU)

从头遍历内存块数组，找出lastVisitTime最小(也就是最久没有被访问过的)的内存块，将将要置换的页面换入该内存块，并修改getInTime和lastVisitTime。

```
alLRU()//根据lastVisitTime
    {
      let minIndex=0;
      for(let i=0;i<this.blockNum;i++)
      {

 if(this.blockList[minIndex].lastVisitTime>this.blockList[i].lastVisitTime)
        {
          minIndex=i;
        }
      }
      this.blockList[minIndex].pageID=this.orderList[0].pageID;
      this.blockList[minIndex].lastVisitTime=this.clock;
      this.blockList[minIndex].getInTime=this.clock;
      //alert("成功换页");
    }
```

2.2.3 待执行指令序列的生成

缺页率很大程度上与指令序列有关，但若想完全体现页面置换算法的性能，需要指令序列存在相对平衡的随机性和顺序性。本实验中参考了PPT中所给的思路，但设计的指令序列有较大的局限性，缺页率更多的与指令序列有关。本实验中首先根据没页含10条指令的特点生成了一个完全顺序的指令序列instructionList，然后再从该序列中根据一定算法选取指令插入待执行指令序列中。

1. 实施方法如下（以320条指令为例）：
2. 在0 - 319条指令之间，随机选取一个起始执行指令，如序号为m
3. 顺序执行下一条指令，即序号为m+1的指令
4. 通过随机数，跳转到前地址部分0 - m-1中的某个指令处，其序号为m1
5. 顺序执行下一条指令，即序号为m1+1的指令
6. 通过随机数，跳转到后地址部分m1+2~319中的某条指令处，其序号为m2
7. 顺序执行下一条指令，即m2+1处的指令。
8. 重复跳转到前地址部分、顺序执行、跳转到后地址部分、顺序执行的过程，直到执行完320条指令

除此之外，本人还尝试过完全随机的方法，但随机性太强，不满足程序的局部性原理，缺页率过高，也不符合实际情况与需求。

```
    initOrderList()
    {
      this.orderList=[];
      let randNum=Math.floor(Math.random()*(this.instructionNum));
      this.orderList.push(
          {
            address: this.instructionList[randNum].address,
            instructionID: this.instructionList[randNum].instructionID,
            pageID: this.instructionList[randNum].pageID,
          },
          {
            address: this.instructionList[++randNum].address,
            instructionID: this.instructionList[++randNum].instructionID,
```

```
            pageID: this.instructionList[++randNum].pageID,
          },
      )
      for(let i=2;i<this.instructionNum;i++)
      {
        switch((i+1)%4)
        {
          case 2:randNum++;break;
          case 1:randNum=Math.floor(Math.random()*(randNum-1));break;
          case 0:randNum++;break;
          case 3:randNum++;randNum=Math.floor(Math.random()*
(this.instructionNum-randNum)+randNum);break;
        }
        this.orderList.push(
            {
              address: this.instructionList[randNum].address,
              instructionID: this.instructionList[randNum].instructionID,
              pageID: this.instructionList[randNum].pageID,
            }
        )
      }
      //乱序怎么搞
    }
```

2.3 效果截图