

内存管理

一.实验目的

1. 了解并掌握内存管理，进行内存容量检查，通过关闭高速缓存来避免检测错误。
2. 学习两种内存管理方法：位图管理和列表管理，同时应用列表法进行实践。
3. 了解并掌握内存优化问题。

二.实验原理

2.1 内存管理的重要性

1.资源有效利用

内存是计算机的关键资源之一。有效的内存管理确保系统能够充分利用有限的内存空间，使得多个程序或进程可以同时运行而不会因为内存不足而出现故障。例如，在一个多任务操作系统中，同时打开多个应用程序（如文字处理软件、浏览器、音乐播放器等），内存管理系统需要合理分配内存给每个应用程序，以保证它们都能正常工作。

2. 程序稳定性和性能

正确的内存管理有助于提高程序的稳定性和性能。如果内存分配不当，可能会导致程序出现内存泄漏（程序不断占用内存却不释放，导致可用内存逐渐减少）、内存碎片（内存空间被分割成许多小的、不连续的片段，难以分配给需要较大连续内存块的程序）等问题，从而使程序运行缓慢甚至崩溃。

2.2 内存管理的基本操作

1. 内存分配

- **静态分配**：在程序编译时就确定内存分配的大小和位置。例如，在 C 语言中定义全局变量和静态变量时，编译器会在程序的数据段为这些变量分配固定大小的内存空间。这种方式简单直接，但缺乏灵活性，因为分配的内存大小在程序运行期间不能改变。
- **动态分配**：程序在运行过程中根据需要请求内存。在 C 和 C++ 语言中，通过函数如 malloc、calloc（C 语言）和 new（C++ 语言）来实现动态内存分配。这些函数会从系统的堆（heap）空间中分配指定大小的内存块，并返回指向该内存块的指针。例如：

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

int main() {
    int *ptr;
    ptr = (int *)malloc(5 * sizeof(int)); // 动态分配能存储5个整数的内存空间
    if (ptr == NULL) {
        printf("内存分配失败\n");
        return 1;
    }
    // 使用分配的内存空间
    for (int i = 0; i < 5; i++) {
        ptr[i] = i;
    }
    // 释放内存
    free(ptr);
    return 0;
}
```

- 在上述 C 语言示例中，malloc 函数用于动态分配内存。首先检查返回的指针是否为 NULL，以确保内存分配成功。然后可以像使用数组一样使用分配的内存空间，最后通过 free 函数释放内存。

2. 内存回收

- 当程序不再需要已分配的内存时，需要及时回收，以便其他程序或进程可以使用。在 C 语言中，使用 free 函数释放由 malloc、calloc 等函数分配的内存；在 C++ 语言中，除了使用 delete 来释放单个对象的内存，使用 delete[] 来释放动态分配的数组内存。不及时回收内存会导致内存泄漏问题。例如，下面是一个内存泄漏的错误示例：

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

void leak_memory() {
    int *ptr = (int *)malloc(sizeof(int));
    *ptr = 10;
    // 没有释放内存就返回，导致内存泄漏
}

int main() {
    leak_memory();
    // 此处程序继续运行，但之前分配的内存已经无法访问，也没有被释放
    return 0;
}
```

3. 内存保护

内存保护机制防止程序访问不属于它的内存区域。现代操作系统通过硬件支持（如内存管理单元 - MMU）和软件配合来实现内存保护。例如，每个进程都有自己独立的虚拟地址空间，当一个进程试图访问另一个进程的内存区域或者访问受保护的内核空间时，MMU 会检测到非法访问并触发操作系统的异常处理机制，通常会导致程序出错并终止，以保护系统的安全和稳定。

2.3 内存管理策略

1. 分区式内存管理

- **固定分区**：将内存划分为固定大小的分区，每个分区可以分配给一个进程。这种方式简单，但分区大小固定可能导致内部碎片（分区内未被充分利用的空间）问题，而且对于大小不同的进程适应性较差。
- **可变分区**：根据进程的实际需要动态地划分内存分区。这种方式可以更好地利用内存空间，但可能会产生外部碎片（内存中存在许多小的空闲分区，但由于不连续而无法分配给需要较大连续空间的进程）。为了解决外部碎片问题，可以采用内存紧缩（将内存中的所有进程移动到内存的一端，使空闲分区合并成一个大的连续分区）等技术，但内存紧缩会带来一定的性能开销。

2. 页式内存管理

把内存空间和进程的虚拟地址空间划分成固定大小的页面（page）。当进程访问内存时，操作系统通过页表（page table）将虚拟页面映射到物理页面。这种方式可以有效减少外部碎片，并且便于内存的换入换出（当内存不足时，可以将暂时不使用的页面交换到磁盘等外部存储设备上，需要时再换入内存）。例如，在一个 32 位的操作系统中，假设页面大小为 4KB，那么虚拟地址空间和物理内存空间都会被划分成许多 4KB 大小的页面，通过页表来记录虚拟页面和物理页面之间的映射关系。

3. 段式内存管理

根据程序的逻辑结构（如代码段、数据段、堆栈段等）将进程的地址空间划分为不同的段（segment）。每个段有自己的名称和长度，可以独立地进行内存分配和保护。段式内存管理比较符合程序的逻辑结构，便于程序的共享和保护，但也可能会产生外部碎片。

4. 段页式内存管理

结合了段式和页式内存管理的优点。先将进程的地址空间按照逻辑段划分，然后每个段再划分成固定大小的页面。这种方式在一定程度上兼顾了程序的逻辑结构和内存的高效利用，是现代操作系统中常用的内存管理策略之一。

2.4 内存管理优化策略

1. 优化内存分配策略

- 内存池技术：预先分配一块较大的内存区域作为内存池，当程序有内存需求时，从内存池中分配合适大小的内存块，使用完毕后归还给内存池，而不是频繁地向操作系统申请和释放小块内存。这样可以减少系统调用开销（因为系统调用分配和释放内存相对耗时），例如在网络服务器程序中，针对每个连接的数据包处理，可以使用内存池来分配接收和发送缓冲区，提高处理效率。
- 合理选择分配算法：
 - **首次适应算法**（First Fit）：在分配内存时，从空闲内存链表或空闲内存块集合中，按照顺序查找第一个能满足需求大小的空闲内存块进行分配。优点是速度较快，能快速找到可用内存；缺点是容易造成内存碎片，使后续大内存块分配困难。改进的方式可以是定期对空闲内存块进行整理排序，按照大小或者地址等规则，便于更合理地查找和分配。
 - **最佳适应算法**（Best Fit）：遍历所有空闲内存块，选择大小最接近请求内存大小的空闲内存块进行分配。这种算法能更好地利用内存空间，减少碎片产生，但缺点是查找效率相对较低，因为需要比较每个空闲内存块的大小。可以通过建立索引数据结构（如二叉搜索树等）来记录空闲内存块大小信息，提高查找最佳匹配块的速度。
 - **最坏适应算法**（Worst Fit）：总是选择最大的空闲内存块进行分配，期望剩余的空闲内存块还能满足后续其他内存分配请求。不过它容易导致产生较多难以利用的小内存碎片，可配合内存碎片整理机制来改善这种情况。

2. 减少内存碎片

- **内存碎片整理**：
 - 紧凑（Compaction）技术：将分散的空闲内存块移动并合并成较大的连续空闲内存块。比如在分区式内存管理中，通过移动已分配的内存分区到内存的一端，使空闲分区在另一端合并成一个大的连续分区。但这种方式在运行时进行会有一定开销，因为需要暂停相关程序、更新内存地址映射等，通常可以选择在系统负载较低的时段进行定期整理。
 - 分区合并算法：针对段式或段页式内存管理中不同逻辑段产生的碎片，通过分析各个段的使用情况，将相邻且可合并的空闲段进行合并，减少段间的碎片，提升内存整体的连续性和可用性。
- **避免过度分配和释放**：尽量一次性申请合适大小的内存，避免多次申请小块内存，因为频繁的申请和释放操作容易导致内存碎片产生。例如，在动态数组实现中，如果能预估到最大元素个数，就一次性申请足够大的内存空间，而不是不断地动态扩容（每次扩容都涉及重新分配内存、复制数据等操作，容易产生碎片）。

3. 提高内存使用效率

- **数据结构优化**：选择更紧凑的数据结构来存储数据，减少内存占用。例如，在存储大量整数时，如果整数范围较小（比如在 0 - 255 之间），可以使用 `uint8_t`（8 位无符号整数类型）来代替 `int`（通常 32 位或更多位），能显著节省内存空间。再如，对于稀疏矩阵（大部分元素为 0 的矩阵），可以采用压缩存储方式（如只存储非零元素及其位置信息），而不是完整地存储整个矩阵。
- **共享内存机制**：多个进程或线程之间如果需要访问相同的数据，可以通过共享内存的方式来避免重复分配内存。比如在数据库管理系统中，多个查询进程可能需要访问同一份数据字典信息，通过共享内存来存放，既节省内存，又便于数据的一致性维护。但需要注意对共享内存的同步访问控制，防止出现数据竞争等问题。

- **缓存机制优化：**合理设置缓存的大小和替换策略，提高缓存命中率，减少对内存或磁盘等较慢存储介质的访问。例如，在操作系统的文件系统缓存中，根据文件的访问频率和最近使用时间等因素来决定缓存哪些文件数据块，采用如 LRU（最近最少使用）替换策略，当缓存已满时替换掉最近最少使用的数据块，从而让常用的数据尽量保留在缓存中，加快文件读写速度。

三.实验内容与要求

3.1 内存容量检查

1. 整体思路：

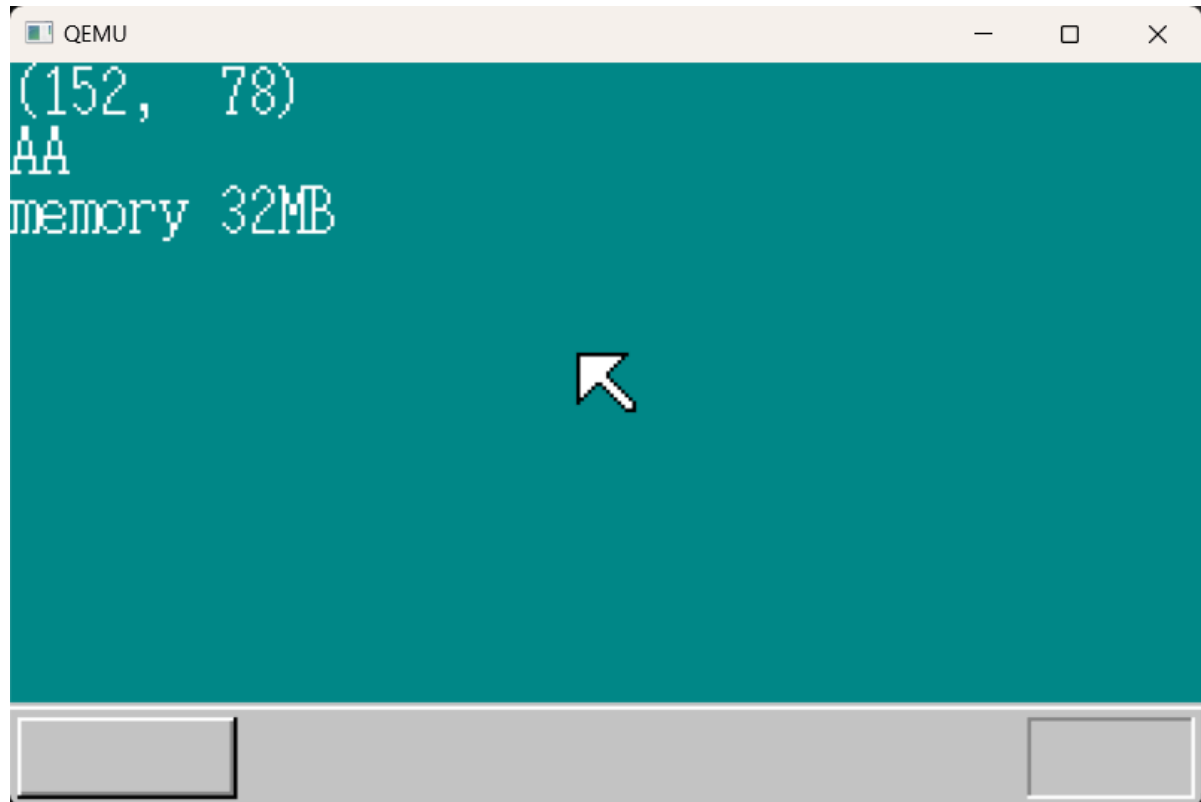
- 确认cpu的型号；判断是否有高速缓冲寄存器，若存在则关闭高速缓冲寄存器；
- 开始循环进行检查操作（为加快速度，每次循环4kb，检测最后的两字节）
- 保存写入的内存地址的旧数据
- 向内存写入数据
- 写入数据后，为防止机型原因，对内存的数据进行反转后对比，确认写入成功
- 恢复旧数据

由于486中含有高速缓冲寄存器的存在（高速缓存：主要为了解决cpu与内存中间存在的速度的差异性的问题，cache会存储着访问内存的地址和内容（一般都是即将或者最近要使用的）。cpu每次要访问内存时，都会先将所访问的地址和内容放入cache中，再有cache放入内存。而读数据时，也是先从cache里先读的），所以在通过往内存里写入值来检测内存的可使用大小的时候，需要先关闭cache。

2. 代码操作

打开整理后的源文件 test1，将bootpack.c和naskfunc.nas中代码补充完整。

点击.bat文件，运行结果如下：



3.2 内存管理

1. 管理方法

(1) 位图管理

定义

位图 (Bitmap) 是一种用二进制位 (0 和 1) 来表示数据状态的数据结构。在内存管理等场景中, 每一位对应一个资源单元 (如内存块、磁盘块等)。例如, 在内存管理的位图中, “0” 位可能表示对应的内存块未被使用, “1” 位表示内存块已被占用。

假设我们有一个由 32 位组成的位图来管理一段内存区域, 该内存区域被划分为 32 个等大小的内存块。如果位图的值为 00000000000000000000000000000001, 这意味着只有最后一个内存块 (从右往左数) 被占用, 其他内存块都是空闲的。

优缺点

空间效率高: 对于表示大量资源的状态, 位图占用的空间相对较小。例如, 要管理 1024 个内存块的状态, 只需要 1024 位 (128 字节) 的位图, 而不是使用其他更复杂的数据结构来记录每个内存块的状态。

操作简单快速: 判断一个资源是否被占用或释放一个资源, 只需要对相应的位进行简单的位操作 (如读取位的值、将位设置为 0 或 1)。在位图中设置一个位 (标记资源被占用) 或清除一个位 (标记资源被释放) 的操作通常是非常快速的, 一般可以在常数时间内完成。

分辨率有限: 位图的粒度是由位来决定的, 它不能很好地处理小于位所代表资源单位的情况。例如, 如果位代表的是 4KB 的内存块, 就很难直接管理更小的内存分配请求。

查找空闲资源效率可能较低: 当需要寻找连续的空闲资源 (如连续的内存块) 时, 可能需要遍历位图中的多个位。如果位图管理的资源数量很大, 这种遍历可能会比较耗时。在最坏的情况下, 可能需要遍历整个位图来找到足够长的连续空闲位。

(2) 列表管理

定义

列表管理是指使用链表 (Linked List) 或数组 (Array) 等数据结构来管理资源。以链表为例, 链表中的每个节点包含资源的相关信息 (如内存块的起始地址、大小等) 以及指向下一个节点的指针。

例如, 在内存管理的链表中, 一个节点可能包含内存块的起始地址、内存块的大小和一个指向下一个内存块节点的指针。当需要分配内存时, 遍历链表, 找到合适大小的内存块进行分配; 当内存块被释放时, 将其插入到合适的位置 (根据大小、地址等因素)。

优缺点

灵活性高: 可以方便地记录资源的各种详细信息, 如内存块的大小、起始地址、所属进程等。对于复杂的资源管理场景, 能够更好地满足需求。例如, 在动态内存分配系统中, 可以记录每个内存块的具体大小, 以便更好地处理大小不同的内存分配请求。

易于插入和删除操作: 在链表结构中, 插入和删除节点相对容易。当一个资源被释放或者新资源被添加时, 不需要像位图那样可能需要大量的位移动操作。例如, 在释放一个内存块时, 只需要调整链表中相关节点的指针即可将该内存块节点从已分配列表移动到空闲列表。

空间开销较大: 与位图相比, 列表管理需要存储更多的信息。除了资源本身的状态信息, 还需要存储节点之间的连接信息 (如指针)。例如, 一个简单的链表节点除了存储内存块的大小和地址信息外, 还需要存储一个指针 (在 32 位系统中, 指针通常占用 4 字节), 这增加了额外的空间开销。

遍历效率可能较低: 在查找特定资源或者满足一定条件的资源时, 可能需要遍历整个列表。特别是在列表很长的情况下, 这种遍历操作可能会比较耗时。例如, 在一个很长的空闲内存块链表中, 寻找一个足够大的空闲内存块可能需要遍历多个节点。

2. 代码操作

(1) 位图管理

假设我们以 0x1000 4kb 为单位进行管理, 则 128m 需要 32768 个字节。然后通过往里面填入 0, 1, 如果我们需要 100kb 的空间, 我们就从 a 中找出连续 25 个进行标记:

```

j = 0
retry:
for(i = 0; i < 25 ;i++)
{
if(a[j+i] != 0 )
{
j++;
if(j<32768 -25) goto retry;
// "无可利用内存"
}
}
//找到了从j到j+24个连续为0的空闲地址
//然后把这些地方标记为1，利用j值计算对应的地址放大0x1000倍即可

```

如果释放地址的话：比如从0x00123000开始的100kb不需要了：

```

j = 0x00123000 / 0x1000;
for(i = 0; i<25; i++)
{
a[j + i ] = 0;
}

```

这个方法虽然看起来操作简单，但是其管理表所占用的内存地址空间会很大，比如管理3gb内存时，需要786432，即768kb的大小。虽然管理表可以利用位来代替char类型进行管理，使得整体大小降低1/8， 经需96kb即可。

(2)列表管理

- 利用列表的管理方法，把从xxx开始的yyy字节地址空间都是空着的这种信息存于列表当中进行管理，利用结构体：

```

struct FREEINFO {    //可用信息
    unsigned int addr, size;
};

struct MEMMAN {      //进行内存管理
    int frees, maxfrees, lostsize, losts;
    struct FREEINFO free[MEMMAN_FREES[1000]];
}

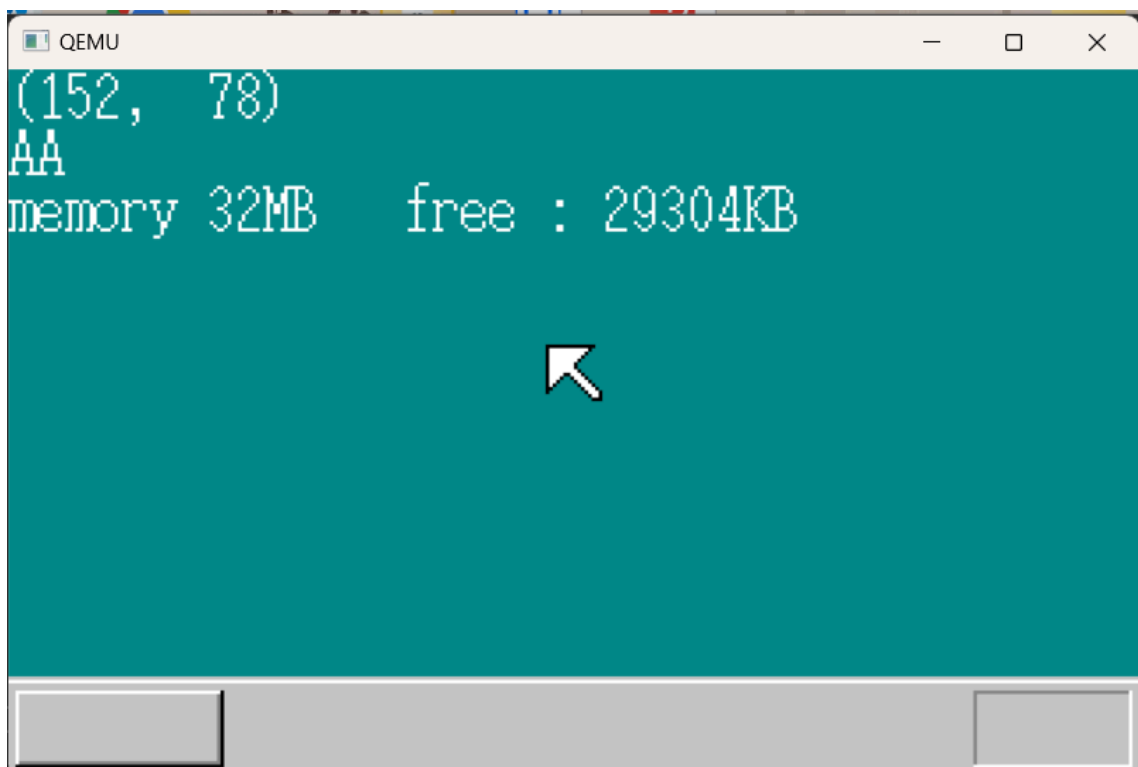
```

例如：memman.free[0].addr = 0x00400000 ; memman.free[0].size = 0x07c00000 代表着从0x00400000开始有124mb空间可用。

- 使用后，直接将这一段信息从可用空间中删去即可。释放时，需查看前后是否可以合并，能合并就合并，不能就单独添加进入即可。

打开整理后的源文件 test2，先完成test1补充代码的整合，再将bootpack.c代码补充完整。

点击.bat文件，运行截图如下：

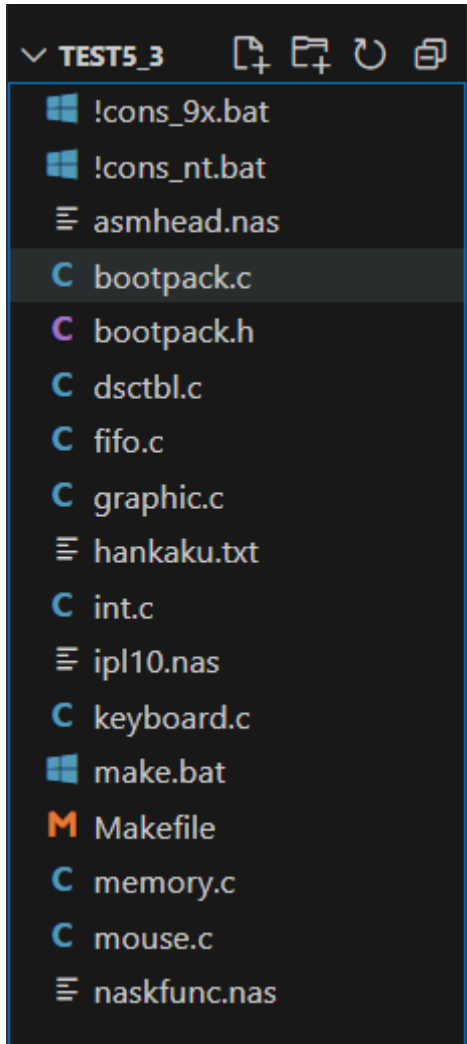


3.3 内存管理优化

1. 分割管理和内存优化

由于之前的内存管理程序中，都是以1字节为单位进行管理的，这就有可能导致出现过多的不连续的小段的内存空间地址，也会重复多次的内存进行分配和释放，不利于我们系统的运行稳定性和利用率等等。因此，下面我们将以0x1000（4kb）为单位进行内存的分配于与管理。如果分配的不足4kb则向上取整，直接默认为4kb。

在此之前，先将内存管理的代码写入到memory.c当中，以便进行后续的分割管理,代码模块如下：



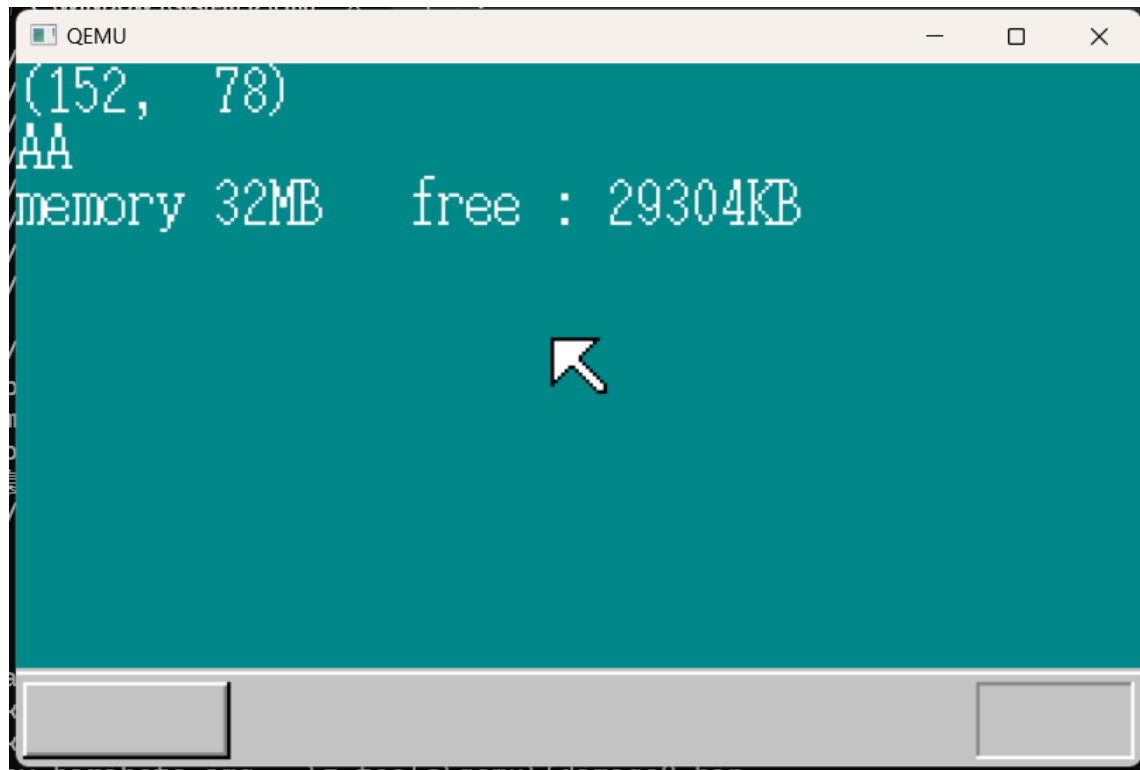
2. 代码操作

在之前的代码中，memman_alloc和memman_free能够以1字节为单位进行内存管理，这种方式虽然不错，但是有一点不足——在反复进行内存分配和内存释放之后，内存中就会出现很多不连续的小段未使用空间，这样就会把man—>free消耗殆尽。

因此，我们要编写一些总是以0x1000字节为单位进行内存分配和释放的函数，它们会把指定的内存大小按0x1000字节为单位向上舍入（roundup）。（0x1000字节的大小正好是4KB）

打开整理后的源文件 test3，先完成test1和test2补充代码的整合，再将bootpack.c代码补充完整。

点击.bat文件，运行截图如下：



四.实验代码及运行截图

4.1 内存容量检查

4.2 内存管理

4.3 内存管理优化

五. 实验分析

5.1 内存容量检查

5.2 内存管理

5.3 内存管理优化

六. 实验中遇到的问题及解决办法

七.总结