MP4: Virtual Memory Management and Memory Allocation

Rajarshi Das
UIN: 334002541
CSCE 611: Operating Systems

Assigned Tasks

Main: Completed

System Design

The goal of machine problem 4 is to complete the memory manager for our operating system. The design is built upon previous machine problems.
For our system, the memory layout looks like the following (adapted from the MP2 handout): -

There are a few parts of our operating system that needs to extended for supporting Virtual Memory. These include: -
1. Extending the page table to support very large numbers and sizes of address spaces.
2. Supporting Virtual Memory.
3. Implementing a simple new and delete operators for C++ programs.

We use recursive page table lookup with the last entry in the page directory points back to the front of the directory. This helps our page management significantly and we don’t need to manage any heavy data structures to know the page directory or the page table for our logical address.

Code Description

For MP3, the following files have been changed:-

- cont_frame_pool.C
- page_table.H
- page_table.C
- vm_pool.H
- vm_pool.C

The following sub-sections provide details on the changes.

cont_frame_pool.C : release_frame_pool

This now has a slight modification to release only those frames which are in the state != FREE. This is different from before in that earlier the function blindly made the frames Free. This leads to modifying HoS frames as well, which is not intended when supporting virtual memory.

```
void
ContFramePool::release_frame_pool(unsigned long _first_frame_no) {
   if(get_state(_first_frame_no - base_frame_no) != ContFramePool::FrameState::HoS) {
      Console::puts("RELEASE_FRAMES: Incorrect HoS state\n");
      assert(0);
   }
   /**
    * Instead of freeing the entire pool, free only those frames that are not free.
    */
   set_state(_first_frame_no, ContFramePool::FrameState::Free);
   numFreeFrames++;
   unsigned long fno = _first_frame_no + 1;
   while(get_state(fno - base_frame_no) != ContFramePool::FrameState::Free) {
      set_state(fno, ContFramePool::FrameState::Free);
      numFreeFrames++; fno++;
   }
}
```

page_table.H :

A static pointer to the virtual memory mem pool has been added.

```
   /* THESE MEMBERS ARE COMMON TO ENTIRE PAGING SUBSYSTEM */
   static PageTable     * current_page_table; /* pointer to currently loaded page table object */
   static unsigned int    paging_enabled;     /* is paging turned on (i.e. are addresses logical)? */
   static ContFramePool * kernel_mem_pool;    /* Frame pool for the kernel memory */
   static ContFramePool * process_mem_pool;   /* Frame pool for the process memory */
   static unsigned long   shared_size;        /* size of shared address space */
   static VMPool        * vm_pool_hptr;       /* Head Pointer to the Virtual Memory Pool LL*/

   /* DATA FOR CURRENT PAGE TABLE */
   unsigned long        * page_directory;     /* where is page directory located? */
```

page_table.C : init_paging()

This is same as MP3. Initializes the page table static objects.

```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size)
{
  PageTable::kernel_mem_pool = _kernel_mem_pool;
  PageTable::process_mem_pool = _process_mem_pool;
  PageTable::shared_size = _shared_size;

  if(DEBUGGER_EN) Console::puts("Page Table Initialization: DONE.\n");
}
```

page_table: constructor page_table( )

This is similar to the constructor in MP3. However, in order to support a recursive page table lookup, the last page directory entry points to the front of the page directory. Only the first and last entries are made valid (present). The direct mapped region is also allocated here.

```
PageTable::PageTable()
{
  paging_enabled = 0; // Ensure paging is disabled

   // Shared page size
   unsigned long shared_pages = PageTable::shared_size / PAGE_SIZE;

   // Allocate Single Frame to First Level - Page Directory
   page_directory = (unsigned long*) (kernel_mem_pool->get_frames(1) * PAGE_SIZE);

   // Allocate Single Frame to Second Level - Page Table
   unsigned long* page_table_ptr = (unsigned long*) (process_mem_pool->get_frames(1) * PAGE_SIZE);

   // Update the First Page Directory Entry and mark the remaining as invalid in directory
   page_directory[0] = (unsigned long)page_table_ptr | RW_MASK_EN | VALID_MASK_EN;

   // Since we use recursive PT lookup, point the last entry to the front of the Page Dir.
   page_directory[shared_pages - 1] = (unsigned long)page_directory | RW_MASK_EN | VALID_MASK_EN;

   unsigned long idx {0};

   for(idx = 1; idx < shared_pages - 1; ++idx) {
      page_directory[idx] = (unsigned long) 0 | RW_MASK_EN;
   }

   // Setup direct mapped pages
   unsigned long page_address {0};

   // Shared memory stuff
   for(idx = 0; idx < shared_pages; ++idx, page_address+=PAGE_SIZE) {
      page_table_ptr[idx] = page_address | RW_MASK_EN | VALID_MASK_EN;
   }
  if(DEBUGGER_EN) Console::puts("Setup Page.\n");
}
```

page_table.C : load( )

This is used to load the page table. Same as in MP3.

```
void
PageTable::load()
{
  current_page_table = this;
  write_cr3((unsigned long)(current_page_table->page_directory)); // Reference: osdever.net
  if(DEBUGGER_EN) {Console::puts("Page Table Loaded!\n");}
}
```

page_table.C : enable_paging( )

Used to enable the paging. Same as in MP3.

```
void
PageTable::enable_paging()
{
  write_cr0(read_cr0() | 0x80000000); // Reference: osdever.net
  paging_enabled = 1;
  if(DEBUGGER_EN) {Console::puts("Paging Enabled!\n");}
}
```

page_table.C : handle_fault()

This is the page table fault handler. The higher level mechanisms are same as in MP3, with changes added to accommodate virtual memory. Some debug prints are also added. We check the virtual memory pool for legitimacy of the virtual address. If it isn't an assertion failure is raised. The page table walk is similar to before. In this case, we allocate a page table when the page directory is allocated for the first time. This avoids a second page fault due to the page table not being allocated.

The remaining mechanisms are similar to MP3.

```
void
PageTable::handle_fault(REGS * _r)
{
   if(DEBUGGER_EN) {Console::puts("PAGE_FAULT_HANDLER: Handling Page Fault.\n");}

   // Extract the error code
    unsigned long error_code = _r->err_code;

    if( (error_code & 0x1) == 0) { // Page not present
        //Console::puts("PAGE_FAULT_HANDLER: ERROR_CODE = 0x0.\n");
        // Obtain pointer to the current page directory
        unsigned long* directory_ptr = (unsigned long*) read_cr3();

        unsigned long faulted_page_addr = read_cr2();
        if(DEBUGGER_EN) {
            Console::puts("Faulted Address: ");
            Console::putui(faulted_page_addr);
            Console::puts("\n");
        }

        unsigned long pde_idx = (faulted_page_addr >> PDE_OFFSET); // Page Table Directory Index
        unsigned long pte_idx = (faulted_page_addr >> PTE_OFFSET) & (PTE_IDX_MASK); // Page Table Entry Index
```

```
    unsigned long* page_table_ptr = nullptr;
    unsigned long* page_dir_ptr = nullptr;

    // Check if logical address is valid and legit
    bool is_legal = false;

    // Parse VM Pool
    VMPool* current = PageTable::vm_pool_hptr;

    while (current != nullptr) {
        if (current->is_legitimate(faulted_page_addr) == true) {
            is_legal = true;
            break;
        }
        current = current->vm_pool_next_ptr;
    }

    if((current != nullptr) && (is_legal == false)) {
      Console::puts("PAGE_FAULT_HANDLER: Address is not legitimate.\n");
      assert(0);
    }

    // Check page fault
    if ((directory_ptr[pde_idx] & VALID_MASK_EN) == 0) { // Level-1 page fault : PDE not present
        if(DEBUGGER_EN) Console::puts("Page Fault due to no PDE.\n");
        int idx {0};
        page_table_ptr = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);

        // 1023 | 1023 | Offset
        unsigned long* page_dir_ptr = (unsigned long *)(PDE_FLAG_CLEAR_MASK);

        page_dir_ptr[pde_idx] = (unsigned long)(page_table_ptr) | RW_MASK_EN | VALID_MASK_EN;

        for(idx = 0; idx < 1024; ++idx) {
            page_table_ptr[idx] = UK_MASK_EN;
        }

        // While at it, allocate PTE to avoid another exception.
        page_dir_ptr = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);

        // Effectively: {1023 | PDE | Offset}
        unsigned long* pte = (unsigned long *)((0x3FF << 22) | (pde_idx << PTE_OFFSET));

        pte[pte_idx] = ( (unsigned long)(page_dir_ptr) | RW_MASK_EN | VALID_MASK_EN);

    }
    else { // Level 2 page fault: PTE not present

        else { // Level 2 page fault: PTE not present
            if(DEBUGGER_EN) Console::puts("Page Fault due to no PTE.\n");
            page_dir_ptr = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);

            unsigned long* pte = (unsigned long *)((0x3FF << 22) | (pde_idx << PTE_OFFSET) );

            pte[pte_idx] = ( (unsigned long)(page_dir_ptr) | 0b11 );
        }
     }

    Console::puts("PAGE_FAULT_HANDLER: Done.\n");
}
```

page_table.C : register_pool()

Registers the virtual memory pool with the page table. The implementation is based on a linked list (similar to mem pools before). The new pool is added at the tail of the linked list.

```
void
PageTable::register_pool(VMPool * _vm_pool)
{
    // First time initialization of VM
    if(PageTable::vm_pool_hptr == nullptr) {
        PageTable::vm_pool_hptr = _vm_pool;
    }
    else { // Non-first virtual mempool
        VMPool* current = PageTable::vm_pool_hptr;
        // Parse Linked List
        while (current->vm_pool_next_ptr != nullptr) {
            current = current->vm_pool_next_ptr;
        }
        current->vm_pool_next_ptr = _vm_pool;
    }
    Console::puts("registered VM pool\n");
}
```

page_table.C : free_page()
This method is used to release the frame. The page table's release_frames gets called internally once the frame number is determined. After releasing, the TLB is flushed to ensure we do not read garbage values later.

```
void
PageTable::free_page(unsigned long _page_no) {
    // Get the page directory index
    unsigned long directory_idx = (_page_no & 0xFFC00000) >> PDE_OFFSET;

    // Find page table idx
    unsigned long tbl_idx = (_page_no & 0x003FF000 ) >> PTE_OFFSET;

    unsigned long* page_table_ptr = (unsigned long *)((0x000003FF << PDE_OFFSET) | (directory_idx << PTE_OFFSET));

    unsigned long frame_number = ((page_table_ptr[tbl_idx] & PDE_FLAG_CLEAR_MASK) / PAGE_SIZE );

    // Release frame
    process_mem_pool->release_frames(frame_number);

    // Mark PTE invalid
    page_table_ptr[tbl_idx] = (unsigned long) 0 | RW_MASK_EN;

    // Flush the TLB - we don't want residual stuff
    load();
    Console::puts("freed page\n");
}
```

vm_pool.H

This contains the definition of the class VMPool. It has all the necessary class members required to manage the virtual memory (pool). It also contains a structure AllocRegion that is used to keep track of the allocated virtual memory regions.

```
/*--------------------------------------------------------------------------*/
/* DATA STRUCTURES */
/*--------------------------------------------------------------------------*/

/* Forward declaration of class PageTable */
/* We need this to break a circular include sequence. */
class PageTable;

// Keep track of each region in VM
struct AllocRegion
{
   unsigned long  base_address;
   unsigned long  size;
};

/*--------------------------------------------------------------------------*/
/* V M  P o o l  */
/*--------------------------------------------------------------------------*/
/*--------------------------------------------------------------------------*/
/* V M  P o o l  */
/*--------------------------------------------------------------------------*/

class VMPool { /* Virtual Memory Pool */
private:
   /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */
   unsigned long base_address;
   unsigned long size;
   unsigned long num_vmem_regions;           // Number of virtual memory regions
   unsigned long available_mem;              // Size of memory region available

   ContFramePool*             frame_pool;
   PageTable*                 page_table;
   struct AllocRegion*        alloc_regions;    // Pointer to virtual memory region allocation

public:
   VMPool* vm_pool_next_ptr;
```

vm_pool.C : Constructor VMPool

Creates a virtual memory pool. Updates the details of the memory pool from the arguments passed to it. The allocated region structure is also updated with the base address of the region, and the page size. Here, the allocation is done in multiples of PAGE_SIZE only. This can create internal fragmentation but makes our handling much easier.

```
VMPool::VMPool(unsigned long  _base_address,
               unsigned long  _size,
               ContFramePool *_frame_pool,
               PageTable     *_page_table) {

    // Fill in the parameters
    base_address = _base_address;
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;

    vm_pool_next_ptr = nullptr;
    num_vmem_regions = 0;

    // Register the pool
    page_table->register_pool(this);

    // 1st Entry
    AllocRegion* region_ptr = (AllocRegion*)base_address;

    region_ptr[0].base_address = base_address;
    region_ptr[0].size = PageTable::PAGE_SIZE;
    alloc_regions = region_ptr;

    // Number of regions should be increased by 1 now
    num_vmem_regions += 1;

    // Available amount of virtual memory
    available_mem = available_mem - PageTable::PAGE_SIZE;

    Console::puts("Constructed VMPool object.\n");
}
```

vm_pool.C : allocate( )

Allocates the virtual memory pages. This is called from the new() operator. The size of the allocation is determined, and the region is marked for the given allocation.

If the requested size is > the available memory, an assertion failure occurs. Again, allocating in multiples of PAGE_SIZE makes the implementation easier.

```
unsigned long VMPool::allocate(unsigned long _size) {

    // Requested more memory than available
    if( _size > available_mem ) {
        Console::puts("VMPOOL: Allocation failed. Not enough memory.\n");
        assert(0);
    }

    unsigned long allocated_pages {0};

    // Size of pages to allocate
    allocated_pages = (_size + PageTable::PAGE_SIZE - 1) / PageTable::PAGE_SIZE; // Works since the numbers are positive

    // Directory entry for allocated region - based on the previous vm_region
    alloc_regions[num_vmem_regions].base_address = alloc_regions[num_vmem_regions-1].base_address +  alloc_regions[num_vmem_regions-1].size;
    alloc_regions[num_vmem_regions].size = allocated_pages*PageTable::PAGE_SIZE;

    // Number of regions should be increased by 1 now
    num_vmem_regions += 1;

    // Available amount of virtual memory
    available_mem -= allocated_pages * PageTable::PAGE_SIZE;

    Console::puts("Allocated region of memory.\n");

    // return (allocated base addr)
    return alloc_regions[num_vmem_regions-1].base_address;
}
```

vm_pool.C : release( )

This is the function to release the virtual memory. This is called from within the C++ delete operation. First, the region is determined. If there is an issue in determining the region, an assertion failure occurs. This helps manage rogue/ unintended delete operations. Each page is freed, by calling the free_page function of the page table. The tracking variables are also updated.

```
void VMPool::release(unsigned long _start_address) {
    int index = 0;
    int region_no = -1;
    unsigned long pages_to_free {0};

    // Identify the region
    for(index = 1; index < num_vmem_regions; ++index) {
        if(alloc_regions[index].base_address  == _start_address ) {
            region_no = index;
        }
    }
    if(region_no == -1) {
        Console::puts("Region not found.\n");
        assert(0);
    }
    // Pages to free
    pages_to_free = alloc_regions[region_no].size / PageTable::PAGE_SIZE;
    while(pages_to_free > 0) {
        // Free the page

        //Console::puts("Freeing Page at address: ");
        //Console::putui(_start_address);
        //Console::puts("\n");
        page_table->free_page(_start_address);

        // Freeing done, should change start address now
        _start_address += PageTable::PAGE_SIZE;

        // Onto next
        pages_to_free--;
    }

    // Free the information of regions
    for( index = region_no; index < num_vmem_regions; ++index) {
        alloc_regions[index] = alloc_regions[index+1];
    }

    // Recompute available memory
    available_mem += alloc_regions[region_no].size;

    // Recompute the number of vmem regions
    num_vmem_regions--;

    //Console::puts("Released region of memory.\n");
}
```

vm_pool.C : is_legitimate( )

Checks whether the address is legitimate by checking the address range of the virtual memory region.

```
bool VMPool::is_legitimate(unsigned long _address) {
    //Console::puts("Checking whether address is part of an allocated region.\n");
    return (_address >= base_address) && (_address <= (base_address + size));
}
```

Testing

Like any test-suite, the goal of these tests is to make sure the program functionality is correct, assertions are getting thrown as expected, and corner scenarios are covered. This is not the most through test-suite and leaves us with room for exploration in the future. The test scenarios are shown below.

Test #	Feature(s)	Testcase	Description	Expected Outcome	Status
1	Page Table	Page Table Test Fault at address 4MB. The number of integers accessed: 2048	Tests that the page table works as expected. Does not use the virtual memory pools.	1. Exception 14 should be handled without assertion failure. 2. All memory checks should pass.	PASS
2		Page Table Test Fault at address 4MB. The number of integers accessed: 4096	Tests that the page table works as expected. Does not use the virtual memory pools.	1. Exception 14 should be handled without assertion failure. 2. All memory checks should pass.	PASS
3	Virtual Memory Pool	Page Table Test Fault at address 16 MB. The number of integers accessed: 2048	Tests that the page table works as expected. Does not use the virtual memory pools.	1. Exception 14 should be handled without assertion failure. 2. All memory checks should pass.	PASS
4		Virtual Memory Pool Test	Tests that different memory pools regions:- 1. code 2. heap Can be initialized, used, and freed. Also test the C++ new and delete operations	1. Exception 14 should be handled without assertion failure. 2. All memory checks should pass.	PASS

Test Outputs:

Test 1.

```
Linux
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool initialized
Frame Pool initialized
POOLS INITIALIZED!
Installed exception handler at ISR <14>
Hello World!
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
PAGE_FAULT_HANDLER: Done.
DONE WRITING TO MEMORY. Now testing...
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
```

Test 2.

```
Linux
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool initialized
Frame Pool initialized
POOLS INITIALIZED!
Installed exception handler at ISR <14>
Hello World!
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
```

```
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
DONE WRITING TO MEMORY. Now testing...
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
```

Output of **Test 3**. is similar to Test 1, hence not added.

Test 4.

```
Linux
qemu-system-x86_64 -kernel kernel.bin -serial stdio
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Frame Pool initialized
Frame Pool initialized
POOLS INITIALIZED!
Installed exception handler at ISR <14>
Hello World!
registered VM pool
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
PAGE_FAULT_HANDLER: Done.
PAGE_FAULT_HANDLER: Done.
Constructed VMPool object.
registered VM pool
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
PAGE_FAULT_HANDLER: Done.
Constructed VMPool object.
VM Pools successfully created!
I am starting with an extensive test
of the VM Pool memory allocator.
Please be patient...
Testing the memory allocation on code_pool...
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
Allocated region of memory.
```

```
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
freed page
freed page
freed page
freed page
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
freed page
freed page
freed page
freed page
Testing the memory allocation on heap_pool...
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
```

```
freed page
freed page
freed page
freed page
freed page
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
freed page
freed page
freed page
freed page
Allocated region of memory.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
EXCEPTION DISPATCHER: exc_no = <14>
PAGE_FAULT_HANDLER: Done.
freed page
freed page
freed page
freed page
freed page
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
```