Memory Manager Project

Ryan, Gavin, Caleb, and Aaron

Problem

OS161 does not free memory, if you use it long enough you may run out of memory

```
panic: locktest: thread_fork failed: Out of memory
sys161: 644245094741152598 cycles (35371080 run, 644245094705781518 global-idle)
sys161:   cpu0: 35371080 kern, 0 user, 0 idle)
sys161: 7090 irqs 0 exns 0r/0w disk 12r/8369w console 0r/0w/1m emufs 0r/0w net
sys161: Elapsed real time: 14.535033 seconds (4.43236e+10 mhz)
sys161: Elapsed virtual time: 13.646103937 seconds (25 mhz)
```

Solution

An incredibly basic memory management system

One function to allocate, one function to deallocate, page structure, and page table structure

All information is stored in the page table, this table is passed into the allocate and deallocate functions to record the information

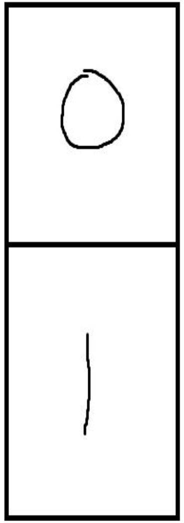
Overview of our system

Our test system uses only two pages we reference them with numbers 0 and 1

Our main focus was to create a simple system that was scalable so that the RAM and page sizes could change but was able to still run

The page table holds all of the information about the RAM that is used

Splitting RAM into pages allows many programs to have access at once



Limitations

Our mockup implementation of a memory manager assumes that all available pages are contiguous. This means that all of the pages will be in physical memory right next to each other. In a real computer, they would be fragmented with only certain regions available for general-purpose use.

Limitations

Our implementation also doesn’t allow a single page to be fragmented into multiple allocations. However, this is not as serious of a problem as you might think.

Limitations

It also doesn’t simulate user-space memory in any way. Our focus was only on memory used in the kernel.

Our Implementation

The first thing that we did is define the memory size of our simulated computer:

```
#define MEM_SIZE 8192
#define PAGE_SIZE 4096
```

We made the memory size be 8K with a page size of 4K (very common). This gives our computer two pages of RAM to work with.

We defined the sizes in bytes to make the memory manager easier to implement.

Page Table Structure

We also defined a page table structure similar to what would be used on a real processor:

```c
//Basic structure for a memory table
struct page_table
{
    int total_pages;
    int pages_available;
    int size_available;
    struct page pages[MEM_SIZE/PAGE_SIZE];
} *my_page_table;
```

It contains information about the total number of pages and how many of them are available. It also contains our individual page structures.

We used the int data type because simulated 32-bit addresses should work fine for our project

Page Structure

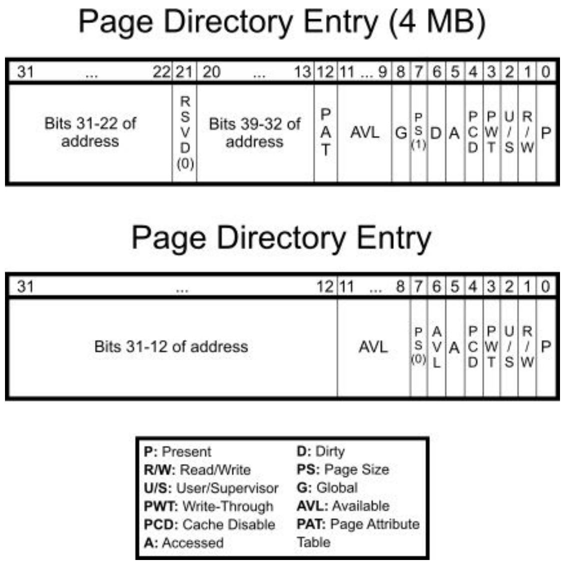
This is what our page structure looks like:

```c
//Basic page structure
struct page
{
    int isused;
    int start_address;
    int pagesused;
    int sizealloc;
};
```

Each page will have one of these structures. It contains important information that the kernel needs to allocate memory using the page.

Actual Page Data Structures

You can see that our example data structures are not that different from the actual page table and page table entry data structures used in the very common x86 processors.



Source: wiki.osdev.org

Allocation Function

```
//Allocations happen here, give it size of allocation and the page_table
void *my_alloc(int request_size, struct page_table *my_table)
{
```

Our allocation function takes a size in bytes and a pointer to the page table. The result will be stored in the page table.

An int type should be sufficient for any allocation our simulated kernel could need to make.

However, it is not ideal that it is necessary to check the result in the page table. Further development could allow the allocation function to return a pointer to simulated allocated memory.

Allocation Checks

```
int pages_requested = (request_size/PAGE_SIZE) + 1;
printf("Pages left: %d Requested: %d\n", my_table->pages_available, pages_requested);
//If there is not enough room
if (request_size > my_table->size_available || pages_requested > my_table->pages_available)
{
    printf("Allocation failed, request size too large\n");
    return NULL;
}
```

First, we do an initial check to see if there could be an available page that is large enough.

Try to Find a Page

A while loop searches through the page table trying to find an available page that can fulfill the request

```c
    //Otherwise we want to loop through memory looking for a contiguous spot to sit
    int page_num = 0;

    //Go until it searches the entire table unsuccessfully or it returns a page number
    while (page_num != my_table->total_pages)
    {
        int empty_blocks = 0;
        int current_page = page_num;
        //This loop checks if there is enough space from the current page number
        while(empty_blocks < pages_requested)
        {
            //Check to see if block is empty, if it's not break
            //Otherwise check the next block
            if (my_table->pages[current_page].isused == 0)
            {
                empty_blocks++;
            } else
            {
                break;
            }
        }

        //Check to see if there are enough blocks to leave the outer loop
        if (empty_blocks >= pages_requested)
        {
            break;
        } else {
            page_num++;
        }
    }
```

Try to Find a Page

The nested while loop checks that there are enough contiguous pages for the request

```c
    //Otherwise we want to loop through memory looking for a contiguous spot to sit
    int page_num = 0;

    //Go until it searches the entire table unsuccessfully or it returns a page number
    while (page_num != my_table->total_pages)
    {
        int empty_blocks = 0;
        int current_page = page_num;
        //This loop checks if there is enough space from the current page number
        while(empty_blocks < pages_requested)
        {
            //Check to see if block is empty, if it's not break
            //Otherwise check the next block
            if (my_table->pages[current_page].isused == 0)
            {
                empty_blocks++;
            } else
            {
                break;
            }
        }

        //Check to see if there are enough blocks to leave the outer loop
        if (empty_blocks >= pages_requested)
        {
            break;
        } else {
            page_num++;
        }
    }
```

Check That the Allocation Worked

```
//No contiguous spot in memory found return NULL
if (page_num == my_table->total_pages){
    printf("Allocation failed, no spot is available\n");
    return NULL;
}
```

We use an if statement to check if the page number is invalid. Then, the function prints an error and returns.

Edit the Page Table

Finally, we update the paging table for the pages we allocated. The for loop is needed because the allocation can span multiple pages. We also need to update how many pages are available.

```c
    int start_addr = page_num*PAGE_SIZE;
    //Otherwise it holds the spot in memory, and we must update the pages to show it's being used
    my_table->pages_available -= pages_requested;
    my_table->size_available -= request_size;
    for (int i = 0; i < pages_requested; i++)
    {
        my_table->pages[page_num].isused = 1;
        my_table->pages[page_num].start_address = start_addr;
        my_table->pages[page_num].pagesused = pages_requested;
        my_table->pages[page_num].sizealloc = request_size;
    }
    printf("Successfully Allocated!\n");
    return NULL;
}
```

We also print that the allocation was successful for testing purposes.

Deallocating Function

```
//Deallocating function, you give it the starting page and the page_table and it updates the pages to show they're not used
void *my_dealloc(int start_page, struct page_table *my_table)
```

Our simple version of a free() function

```
//Check to make sure there is something actually allocated at the address (a soft check, this is a dangerous dealloc)
if (my_table->pages[start_page].pagesused == 0)
{
    printf("Deallocation failed, page given is already empty\n");
    return NULL;
}
```

Safety check

What would happen without?

Deallocating Function

Goals: Keep the page table up to date

```
//Set the current page
int cur_page = start_page;
my_table->pages_available += my_table->pages[start_page].pagesused;
my_table->size_available += my_table->pages[start_page].sizealloc;
    //Starting at the page given until the number of pages allocated, fix the page information
    for(int i = 0; i < my_table->pages[start_page].pagesused; i++)
    {
        my_table->pages[cur_page].isused = 0;
        my_table->pages[cur_page].start_address = -1;
        my_table->pages[cur_page].pagesused = 0;
        my_table->pages[cur_page].sizealloc = 0;
    }
    printf("Successfully Deallocated!\n");
```

Test 1 - 2

```
// BEGIN: Test Case 1
printf("Test Case 1: Allocate 1 page\n");
my_alloc(2000, my_page_table);
printf("Pages available: %d\n", my_page_table->pages_available);
printf("Size available: %d\n", my_page_table->size_available);
printf("\n");
// END: Test Case 1 :

// BEGIN: Test Case 2 : Allocate memory that exceeds available size
printf("Test Case 2: Allocate memory that exceeds available size\n");
my_alloc(10000, my_page_table);
printf("Pages available: %d\n", my_page_table->pages_available);
printf("Size available: %d\n", my_page_table->size_available);
printf("\n");
```

```
Test Case 1: Allocate 1 page
Pages left: 2 Requested: 1
Successfully Allocated!
Pages available: 1
Size available: 6192
```

```
Test Case 2: Allocate memory that exceeds available size
Pages left: 1 Requested: 3
Allocation failed, request size too large
Pages available: 1
Size available: 6192
```

Test 3 - 4

```c
// BEGIN: Test Case 3 : Deallocate memory
printf("Test Case 3: Deallocate memory\n");
my_dealloc(0, my_page_table);
printf("Pages available: %d\n", my_page_table->pages_available);
printf("Size available: %d\n", my_page_table->size_available);
printf("\n");
// END: Test Case 3

// BEGIN: Test Case 4 : Deallocate already deallocated memory
printf("Test Case 4: Deallocate already deallocated memory\n");
my_dealloc(0, my_page_table);
printf("Pages available: %d\n", my_page_table->pages_available);
printf("Size available: %d\n", my_page_table->size_available);
printf("\n");
// END: Test Case 4
```

```
Test Case 3: Deallocate memory
Successfully Deallocated!
Pages available: 2
Size available: 8192
```

```
Test Case 4: Deallocate already deallocated memory
Deallocation failed, page given is already empty
Pages available: 2
Size available: 8192
```

Test 5 - 6

```c
// BEGIN: Test Case 4 : Deallocate already deallocated memory
printf("Test Case 4: Deallocate already deallocated memory\n");
my_dealloc(0, my_page_table);
printf("Pages available: %d\n", my_page_table->pages_available);
printf("Size available: %d\n", my_page_table->size_available);
printf("\n");
// END: Test Case 4

// BEGIN: Test Case 5 : Allocate memory after deallocation
printf("Test Case 5: Allocate memory after deallocation\n");
my_alloc(3000, my_page_table);
printf("Pages available: %d\n", my_page_table->pages_available);
printf("Size available: %d\n", my_page_table->size_available);
printf("\n");
// END: Test Case 5


printf("Deallocating all memory\n")
my_dealloc(0, my_page_table);

print("\n");

// BEGIN: Test Case 6 : Allocate memory that requires multiple pages
printf("Test Case 6: Allocate memory that requires multiple pages\n");
my_alloc(6000, my_page_table);
printf("Pages available: %d\n", my_page_table->pages_available);
printf("Size available: %d\n", my_page_table->size_available);
printf("\n");
// END: Test Case 6
```

```
Test Case 5: Allocate memory after deallocation
Pages left: 2 Requested: 1
Successfully Allocated!
Pages available: 1
Size available: 5192

Successfully Deallocated!
Test Case 6: Allocate memory that requires multiple pages
Pages left: 2 Requested: 2
Successfully Allocated!
Pages available: 0
Size available: 2192
```

Test 7: Thread and real world application test

```
pthread_t thread1, thread2;
int thread_num1 = 1;
int thread_num2 = 2;

printf("Test Case 7: Using threads to allocate and deallocate memory\n");
pthread_create(&thread1, NULL, thread_func, &thread_num1);
pthread_create(&thread2, NULL, thread_func, &thread_num2);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
```

Thread testing Function

- Important testing features
  - Random and differing allocation/ deallocations amounts.
  - Random execution time
- This is to ensure that the test is more accurate to real world applications, because realistically you can not control the either of these factors.

```c
// Thread function to allocate and deallocate memory at random intervals, with random sizes.
void *thread_func(void *arg)
{
    int *thread_num = (int *)arg;
    int request_size = 0;
    int sleep_time = 0;

    // Allocate and deallocate memory 5 times



    for (int i = 0; i < 5; i++)
    {
        // Generate a random request size
        request_size = rand() % 10000 + 1;

        // Generate a random sleep time
        sleep_time = rand() % 5 + 1;


        // Allocate memory
        printf("Thread %d: Allocating %d bytes\n", *thread_num, request_size);
        my_alloc(request_size, my_page_table);
        printf("Thread %d: Pages available: %d\n", *thread_num, my_page_table->pages_available);
        printf("Thread %d: Size available: %d\n", *thread_num, my_page_table->size_available);
        printf("\n");

        // Sleep for a random amount of time
        sleep(sleep_time);


        // Deallocate memory
        printf("Thread %d: Deallocating starting at page %d\n", *thread_num, 0);
        my_dealloc(0, my_page_table);
        printf("Thread %d: Pages available: %d\n", *thread_num, my_page_table->pages_available);
        printf("Thread %d: Size available: %d\n", *thread_num, my_page_table->size_available);
        printf("\n");



        // Sleep for a random amount of time
        sleep(sleep_time);
    }

    return NULL;
}
```

```
Test Case 7: Using threads to allocate and deallocate memory
Thread 1: Allocating 9384 bytes
Pages left: 2 Requested: 3
Allocation failed, request size too large
Thread 1: Pages available: 2
Thread 1: Size available: 8192

Thread 2: Allocating 2778 bytes
Pages left: 2 Requested: 1
Successfully Allocated!
Thread 2: Pages available: 1
Thread 2: Size available: 5414

Thread 2: Deallocating
Successfully Deallocated!
Thread 2: Pages available: 2
Thread 2: Size available: 8192

Thread 1: Deallocating
Deallocation failed, nothing allocated at this address
Thread 1: Pages available: 2
Thread 1: Size available: 8192

Thread 2: Allocating 7794 bytes
Pages left: 2 Requested: 2
Successfully Allocated!
Thread 2: Pages available: 0
Thread 2: Size available: 398

Thread 2: Deallocating
Successfully Deallocated!
Thread 2: Pages available: 2
Thread 2: Size available: 8192

Thread 1: Allocating 5387 bytes
Pages left: 2 Requested: 2
Successfully Allocated!
Thread 1: Pages available: 0
Thread 1: Size available: 2805

Thread 2: Allocating 6650 bytes
Pages left: 0 Requested: 2
Allocation failed, request size too large
Thread 2: Pages available: 0
Thread 2: Size available: 2805

Thread 2: Deallocating
Successfully Deallocated!
Thread 2: Pages available: 2
Thread 2: Size available: 8192

Thread 1: Deallocating
Deallocation failed, nothing allocated at this address
Thread 1: Pages available: 2
```

```
Thread 2: Allocating 2363 bytes
Pages left: 2 Requested: 1
Successfully Allocated!
Thread 2: Pages available: 1
Thread 2: Size available: 5829

Thread 1: Allocating 8691 bytes
Pages left: 1 Requested: 3
Allocation failed, request size too large
Thread 1: Pages available: 1
Thread 1: Size available: 5829

Thread 2: Deallocating
Successfully Deallocated!
Thread 2: Pages available: 2
Thread 2: Size available: 8192

Thread 2: Allocating 7764 bytes
Pages left: 2 Requested: 2
Successfully Allocated!
Thread 2: Pages available: 0
Thread 2: Size available: 428

Thread 1: Deallocating
Successfully Deallocated!
Thread 1: Pages available: 2
Thread 1: Size available: 8192

Thread 2: Deallocating
Deallocation failed, nothing allocated at this address
Thread 2: Pages available: 2
Thread 2: Size available: 8192

Thread 1: Allocating 541 bytes
Pages left: 2 Requested: 1
Successfully Allocated!
Thread 1: Pages available: 1
Thread 1: Size available: 7651

Thread 1: Deallocating
Successfully Deallocated!
Thread 1: Pages available: 2
Thread 1: Size available: 8192

Thread 1: Allocating 9173 bytes
Pages left: 2 Requested: 3
Allocation failed, request size too large
Thread 1: Pages available: 2
Thread 1: Size available: 8192

Thread 1: Deallocating
Deallocation failed, nothing allocated at this address
Thread 1: Pages available: 2
Thread 1: Size available: 8192
```

Linear Inverted Page Table Implementation

- We have also modified our original implementation to fit the linear inverted page table model described in class

- Very similar to normal page table, however:
  - Index into the table using physical address, not virtual
  - Linear search through table to find virtual page number (can be slow)
  - Saves a lot of memory

```c
// Basic page structure
struct page {
    int is_used;
    unsigned int vpn;
    int pages_used;
    int size_alloc;
};
```

IPT Allocation

- The my_malloc function now returns an unsigned int that is meant to mimic a pointer (memory address) being passed back to the programmer

- This “pointer” can be used to free up the memory later on when the programmer is finished
    - The virtual address is calculated by adding 14 to the physical address and shifting that value 3 bits to the left

```
for (int i = 0; i < pages_requested; i++) {
    my_table->pages[page_num].is_used = 1;
    my_table->pages[page_num].vpn = start_addr;
    my_table->pages[page_num].pages_used = pages_requested;
    my_table->pages[page_num].size_alloc = request_size;
    page_num++;
}

printf("Successfully Allocated!\n");

// Release the lock
pthread_mutex_unlock(&mutex);

return v_addr;
```

IPT Deallocation

- Now we can use the pointer that we got earlier to deallocate the memory when we are done. Take the pointer (address), and pass it to the deallocation function to free the memory.

```
// Function to deallocate memory
void *my_dealloc(unsigned int v_addr, struct page_table *my_table) {
    pthread_mutex_lock(&mutex);
```

- The simulated MMU extracts the VPN from the virtual address and then translates the VPN to a physical address to be deallocated by the operating system

IPT Deallocation Linear Search

- Since the inverted page table is indexed using the physical address, we must search through each index to see if we can find the corresponding VPN.
  - This can take a lot of time especially if the page table is large

```
// Check if there is something actually allocated at the address
for (int i = 0; i < my_table->pages_available; i++) {
    if (my_table->pages[i].vpn == v_addr) {
        // Update the page table to show the deallocated memory
        int pages_used = my_table->pages[i].pages_used;
        int size_alloc = my_table->pages[i].size_alloc;
```

- The trade off is saving memory. The only entries in the page table are the ones that currently exist in physical memory
  - Because the IPT has entry for each physical frame, not logical

IPT Testing

- You can see that sometimes there is still memory wasted due to the nature of paging
- There are sub paging systems that exist to grab the fragmented memory but we did not implement that here for the sake of time
- All tests still produce expected results
  - There is no noticeable time change from regular paging because of the small sample

```
Starting main MemSize: 8192 PageSize: 4096
Pages available: 2
Test Case 1: Allocate 1 page
Pages left: 2 Requested: 1
Successfully Allocated!
Pages available: 1
Size available: 6192
Pointer address (virtual): 0x70

Test Case 2: Allocate memory that exceeds available size
Pages left: 1 Requested: 3
Allocation failed, request size too large
Pages available: 1
Size available: 6192

Test Case 3: Deallocate memory
Successfully Deallocated!
Pages available: 2
Size available: 8192

Test Case 4: Deallocate already deallocated memory
Deallocation failed, nothing allocated at this address
Pages available: 2
Size available: 8192

Test Case 5: Allocate memory after deallocation
Pages left: 2 Requested: 1
Successfully Allocated!
Pages available: 1
Size available: 5192

Deallocating all memory
Successfully Deallocated!
```