CS1217 - Spring 2023 - Lab 3

Bhumika Mittal, Saptarishi Dhanuka

Contributions of individual team members:
Bhumika: 5-8
Saptarishi: 1-5

Part 1

Exercise 1

Boot_alloc
We panic if we go beyond the 4 MB limit. Return the old value of nextfree and update nextfree
Mem_init
Boot_alloc the size of npages and initialise the contents to 0 using memset
Page_init
We follow the comments given. If it lies in the IO hole or it lies beyond the hole but before the next free page then we mark it as used, otherwise free and add it to linked list of free pages
Page_alloc
Extract a page from the page_free_list if it exists and follow the comments and hints. If needed then fill the page with \0 bytes using memset and page2kva().
Page_free
Return page to list of free pages

Exercise 2
Reading - *read*

Exercise 3

```
(gdb) x/4x 0xf0100006
0xf0100006:     0x4ffe0000      0xc766e452      0x00047205      0xb8123400
(gdb) x/4x 0xf0102840
0xf0102840:     0x000000c3      0x00000000      0x00000000      0x00000000
```

```
(gdb) x/4x 0xf0100000
0xf0100000:     0x1badb002      0x00000000      0xe4524ffe      0x7205c766
(gdb) x/4x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
(gdb) x/4x 0x00100020
0x100020:       0x0100010d      0xc0220f80      0x10002fb8      0xbde0fff0
(gdb) x/4x 0xf0100020
0xf0100020:     0x0100010d      0xc0220f80      0x10002fb8      0xbde0fff0
(gdb) x/4x 0xf0100025
0xf0100025:     0xb8c0220f      0xf010002f      0x00bde0ff      0xbc000000
```

```
(qemu) xp/4x 0x00100006
0000000000100006: 0x4ffe0000 0xc766e452 0x00047205 0xb8123400
(qemu) xp/4x 0xf0100006
00000000f0100006: 0x00000000 0x00000000 0x00000000 0x00000000
(qemu) xp/4x 0x00102840
0000000000102840: 0x000000c3 0x00000000 0x00000000 0x00000000
```

```
Type 'help' for a list of commands.
K> QEMU 2.3.0 monitor - type 'help' for more information
(qemu) xp/4x 0x00100000
0000000000100000: 0x1badb002 0x00000000 0xe4524ffe 0x7205c766
(qemu) xp/4x 0x00100020
0000000000100020: 0x0100010d 0xc0220f80 0x10002fb8 0xbde0fff0
(qemu) xp/4x 0x00100025
0000000000100025: 0xb8c0220f 0xf010002f 0x00bde0ff 0xbc000000
```

First two are from gdb and we inspect memory at VAs and second image is from QEMU monitor where we inspect memory at corresponding PAs and we can see that they have same contents.

Info pg

```
(qemu) info pg
VPN range     Entry         Flags        Physical page
[00000-003ff]  PDE[000]     ----A----P
  [00000-00000]  PTE[000]     --------WP 00000
  [00001-0009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [000a0-000b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [000b8-000b8]  PTE[0b8]     ---DA---WP 000b8
  [000b9-000ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [00100-00103]  PTE[100-103] ----A---WP 00100-00103
  [00104-00110]  PTE[104-110] --------WP 00104-00110
  [00111-00111]  PTE[111]     ---DA---WP 00111
  [00112-00114]  PTE[112-114] --------WP 00112-00114
  [00115-00156]  PTE[115-156] ---DA---WP 00115-00156
  [00157-00157]  PTE[157]     --------WP 00157
  [00158-003ff]  PTE[158-3ff] ---DA---WP 00158-003ff
[f0000-f03ff]  PDE[3c0]     ----A---WP
  [f0000-f0000]  PTE[000]     --------WP 00000
  [f0001-f009f]  PTE[001-09f] ---DA---WP 00001-0009f
  [f00a0-f00b7]  PTE[0a0-0b7] --------WP 000a0-000b7
  [f00b8-f00b8]  PTE[0b8]     ---DA---WP 000b8
  [f00b9-f00ff]  PTE[0b9-0ff] --------WP 000b9-000ff
  [f0100-f0103]  PTE[100-103] ----A---WP 00100-00103
  [f0104-f0110]  PTE[104-110] --------WP 00104-00110
  [f0111-f0111]  PTE[111]     ---DA---WP 00111
  [f0112-f0114]  PTE[112-114] --------WP 00112-00114
  [f0115-f0156]  PTE[115-156] ---DA---WP 00115-00156
  [f0157-f0157]  PTE[157]     --------WP 00157
  [f0158-f03ff]  PTE[158-3ff] ---DA---WP 00158-003ff
```

Info mem

```
(qemu) info mem
0000000000000000-0000000000400000 0000000000400000 -r-
00000000f0000000-00000000f0400000 0000000000400000 -rw
(qemu)
```

Question 1

x should have type `uintptr_t` since we are assigning it a pointer and pointers in C are virtual addresses. Moreover it can be dereferenced to give the value 10

Exercise 4

Pgdir_walk

We get the address of the page table through the appropriate index of the directory and check its existence and present bit. If we need to create a page then we allocate it and increase the reference count on the struct PageInfo. Then we actually put it into the page directory entry along with permissions. Then we get the physical address of the page table and then the kernel address of that physical address. Then we can use that kernel virtual address and return the page table entry at page_index.

Boot_map_region

We go page by page with the VAs and PAs, and we get the pte for the VA and map VA to PA with permissions

Page_insert

First we get the pte for the VA, creating it on demand if needed. If a page already present at pgtable_entry corresponding to VA then increase the reference count first to avoid the bug of freeing it in case it hits 0 due to page_remove. Then remove the earlier page and and map pp at virtaddr va with permissions. If page isn’t there earlier, then just map and increase reference count.

Page_lookup

First get the mapped page with pgdir_walk. Check existence and permissions then store at the address of the pte if needed. Return the mapped page after passing it through pa2page()

Page_remove

First look for the page that we need to remove. Get its pte through walking, check permissions and existence. Put the pte_store value as 0, decrement reference count and invalidate the TLB.

Exercise 5

1. Added the following lines:

```
boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages), PTE_U);
boot_map_region(kern_pgdir, KSTACKTOP - KSTKSIZE, KSTKSIZE, PADDR(bootstack),
PTE_W);
boot_map_region(kern_pgdir, KERNBASE, 0xffffffff - KERNBASE + 1, 0, PTE_W);
```

   - First one we map the UPAGES (va) as the start address and map entire page table which is available to user using this command.
   - Second one as per the comments maps [KSTACKTOP-KSTKSIZE, KSTACKTOP) to bootstack and the writable bit is enabled and not the user bit so the user can't access it
   - Third one as per the comment, the VA range [KERNBASE, 2^32) should map to the PA range [0, 2^32 - KERNBASE). Here also the writable bit is enabled and not the user bit so the user can't access it.

2. In mem_init I added the following code to print the kern_pgdir

```
    int i;

    uintptr_t va = 0;

    for (i = 0; i < NPDENTRIES; i++, va+= PGSIZE*1024)

    {

        cprintf("Entry: %u, Base VA: %x, Points to: %x\n", i, va,
pgdir_walk(kern_pgdir, (void *)va, 0));

    }
```

We can also just refer back to what we set up previously to fill in the table

Entry	Base Virtual Address	Points to (logically):
1023	0xffc00000	Page table for top 4MB of phys memory
1022	0xff800000	Page table for the second-last chunk of 4 MB
.	.	.
.	.	.
960	0xf0000000 (KERNBASE)	Page table for first 4 MB of phys mem [0,4)
959	0xefc00000 (MMIOLIM or KSTACKTOP-PTSIZE)	Kernel stack and invalid memory
958	0xef800000 (ULIM)	Unmapped
957	0xef400000 (UVPT)	User read-only virtual page table
956	0xef000000 (UPAGES)	Read-only copies of the Page structures
:	:	Unmapped
.	0x00C00000	Unmapped
2	`0x00800000`	Unmapped
1	`0x00400000`	Unmapped
0	`0x00000000`	[see next question]

3. The pages which are there in the kernel's memory don't have the PTE_U set and hence user programs can't access those pages.

4. This operating system can support a max of 256 MB of physical memory. This is because as per pmap.c, all physical memory needs to be mapped from KERNBASE (0xf0000000) to 2^32 bytes, which gives a size of 256 MB. We can't go to the usual 32-bit space of 4 GB due to design constraints

   There's also a comment in pmap.h which states that the maximum is 256 MB

```
/* This macro takes a kernel virtual address -- an address that points above
 * KERNBASE, where the machine's maximum 256MB of physical memory is mapped --
 * and returns the corresponding physical address.  It panics if you pass it a
 * non-kernel virtual address.
 */
```

5. If we have 256 MB of maximum physical memory then the overhead is as follows. I assume that this question just talks about the overhead in the 256 MB of physical memory.
   First we have the struct PageInfo * pages array.
   256 MB needs 256 MB/ 4KB = 65536 pages. Each struct is 8 bytes, thus leading to 524288 bytes = 512 KB here.
   Each physical page also needs a page table entry which is 4 bytes, leading to 262144 bytes = 256 KB here
   The page directory has 1024 entries of 4 bytes each, leading to 4 KB here
   Hence total overhead = 512 + 256 + 4 = 772 KB.

6.

```
# Load the physical address of entry_pgdir into cr3.  entry_pgdir
# is defined in entrypgdir.c.
movl    $(RELOC(entry_pgdir)), %eax
movl    %eax, %cr3
# Turn on paging.
movl    %cr0, %eax
orl $(CR0_PE|CR0_PG|CR0_WP), %eax
movl    %eax, %cr0

# Now paging is enabled, but we're still running at a low EIP
# (why is this okay?).  Jump up above KERNBASE before entering
# C code.
mov $relocated, %eax
jmp *%eax
```

```
=> 0x10002d:    jmp    *%eax
0x0010002d in ?? ()
(gdb) info reg eip
eip            0x10002d            0x10002d
(gdb) si
=> 0xf010002f:  mov    $0x0,%ebp
0xf010002f in ?? ()
(gdb) info reg eip
eip            0xf010002f          0xf010002f
(gdb)
```

We transition after the jmp *%eax instruction as we can see from the above code and gdb output
In entrypgdir.c we can see that the comments say

```
// The entry.S page directory maps the first 4MB of physical memory
// starting at virtual address KERNBASE (that is, it maps virtual
// addresses [KERNBASE, KERNBASE+4MB) to physical addresses [0, 4MB)).
// We choose 4MB because that's how much we can map with one page
// table and it's enough to get us through early boot.  We also map
// virtual addresses [0, 4MB) to physical addresses [0, 4MB); this
// region is critical for a few instructions in entry.S and then we
// never use it again.
```

This is what makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE since those instructions in the

in-between area lie in the virtual addresses [0, 4 MB) which are mapped and hence can run properly. If it had just mapped the above KERNBASE addresses then instructions right after enabling paging would crash.
The transition in between is necessary since we still need to execute instructions to jump above KERNBASE even after turning on paging, which can only happen when the old physical address is still a valid virtual address even after paging.
Moreover, we have to jump to above KERNBASE in the first place because that's where the kernel is linked so that the lower parts of the processor's address space can be used by user programs.

EXERCISE 6

1. Check and enable if the processor supports PTE_PS - cr4 register enables superpaging. edx register in cpuid contains the info about the support for the processor.

```
//check and enable if the processor supports PTE_PS

static int check_pse()
{
   uint32_t edx, unused;
   cpuid(1, &unused, &unused, &unused, &edx);
   char enabled = (edx & 8);

   if (enabled)
   {
       uint32_t cr4 = rcr4();
       lcr4(cr4 | CR4_PSE);
   }

   return 0;
}
```

2. We now add another parameter in the boot_map_region which sets the superpage bit to 1 or 0. Superpage is basically a 4MB page (this helps in doing a more space-efficient job).

3. For the 4MB pages, we increment va and pa by 4MB and set ***pgtable_entry = pa | perm | PTE_P | PTE_PS;***

4. For 4KB pages, it just increment va and pa by 4KB.

5. Then, we implement a simple if-else logic blocks with the boot_map_region function.

EXERCISE 7

1. In monitor.c, we first include pmap.h to get the required functions.

```
#include <kern/pmap.h> // for page2pa(), struct PageInfo, etc
```

2. In the struct Command, we add the following commands that can be used by the user.

```
   {"showmappings", "Display the page mappings of the given virtual
address range", mon_showmappings},

   {"setperm", "Set the permission of the given virtual address range",
mon_setperm},

   {"dumpv", "Dump the content of the given virtual or physical address
range", mon_dumpv},

   {"dumpp", "Dump the content of the given virtual or physical address
range", mon_dumpp},

   {"pagesize", "Prints the page size", mon_pagesize},

   {"loadv", "Modify a byte at given virtual memory", mon_loadv},

   {"loadp", "Modify a byte at given physical memory", mon_loadp},
```

3. Now, for showmappings, we would need some helper functions (names are self-explanatory)

```
// convert hex string to int
uint32_t convert2int(char *str)
{
   uint32_t result = 0;
   int i = 0;
   while (str[i] != '\0')
   {
       if (str[i] >= '0' && str[i] <= '9')
       {
           result = result * 16 + (str[i] - '0');
       }
       else if (str[i] >= 'a' && str[i] <= 'f')
       {
           result = result * 16 + (str[i] - 'a' + 10);
       }
       else if (str[i] >= 'A' && str[i] <= 'F')
       {
           result = result * 16 + (str[i] - 'A' + 10);
       }
       i++;
   }
   return result;
}

// given a virtual address, return the physical address
int virtual2physical(uint32_t virtual_address)
{
   pte_t *pte = pgdir_walk(kern_pgdir, (void *)virtual_address, 0); // get
the page table entry

   if (!pte) // if the page table entry is not present, return -1
       return -1;

   return PTE_ADDR(*pte) + (virtual_address & 0xFFF); // return the
physical address
}

// given a physical address, return the virtual address
int physical2virtual(uint32_t physical_address)
{
   return (uint32_t)KADDR(physical_address); // return the virtual address
}

// extract permission bits from the page table entry in form of three bits
(P, W, U)
// output should be in the form of a string - "P W U" (without the quotes)
- if the permission is present, else "-"
char *extractperm(pte_t pte)
{
```

```
    static char perm[4];
    perm[0] = (pte & PTE_P) ? 'P' : '-';
    perm[1] = (pte & PTE_W) ? 'W' : '-';
    perm[2] = (pte & PTE_U) ? 'U' : '-';
    perm[3] = '\0';
    return perm;
}

// return the permission of the given virtual address
char *getperm(uint32_t virtual_address)
{
    pte_t *pte = pgdir_walk(kern_pgdir, (void *)virtual_address, 0); // get
the page table entry
    if (!pte) // if the page table entry is not present, return -1
        return NULL;
    // return the permission bits of the page table entry
    return extractperm(*pte);
}
```

4. Now, we can simply check for the existence of the page in the PTE. If it exists then print the required details.

```
// show the page mappings of the given virtual address range
int mon_showmappings(int argc, char **argv, struct Trapframe *tf)
{
    // sanity check - make sure the user entered the correct number of
arguments
    if (argc == 1)
    {
        cprintf("The function takes two arguments - beginning address and
end address\n");
        return 0;
    }

    // convert the arguments to integers
    uint32_t start = convert2int(argv[1]);
    uint32_t end = convert2int(argv[2]);

    cprintf("start address: %x, end address: %x\n", start, end);
    for (; start <= end; start += PGSIZE)
    {
        pte_t *pte = pgdir_walk(kern_pgdir, (void *)start, 1); // get the
page table entry

        if (!pte)
            panic("page table entry not found - out of memory?");

        if (*pte & PTE_P) // if the page is present and has a physical
address
        {
```

```
            cprintf("virtual page: %x ", start);                     //
print the virtual address - the start address
            cprintf("physical page: %x ", virtual2physical(start)); //
print the physical address
            cprintf("permission: %s \n", getperm(start));            //
print the physical address
        }
        else
            cprintf("page don't exist: %x\n", start); // if the page is not
present, say that the page does not exist
    }
    return 0;
    // test -- showmappings 0xf0110000 0xf011a000
}
```

5. There are three types of permissions: P- present, W - Writable, U - User. We know create a function to change the existing permission (comments explain the code and logic).

```
// function to print the permission bits of the page table entry
void printPermission(pte_t *pte)
{
    cprintf("PTE_P: %d, PTE_W: %d, PTE_U: %d \r \n", (*pte & PTE_P) ? 1 :
0, (*pte & PTE_W) ? 1 : 0, (*pte & PTE_U) ? 1 : 0);
}

// change the permission of the given virtual address
int mon_setperm(int argc, char **argv, struct Trapframe *tf)
{
    if (argc == 1)
    {
        cprintf("The function takes three arguments - addresss, clear/set
and permission\n");
        return 0;
    }
    uint32_t addr = convert2int(argv[1]); // convert the address to integer

    pte_t *pte = pgdir_walk(kern_pgdir, (void *)addr, 1); // get the page
table entry
    if (!pte)
        panic("page table entry not found - out of memory?");

    cprintf("%x before setperm: ", addr); // print the virtual address
    printPermission(pte);                 // print the permissions

    uint32_t perm = 0; // permission bits

    if (argv[3][0] == 'P')
        perm = PTE_P;
    if (argv[3][0] == 'W')
        perm = PTE_W;
```

```
    if (argv[3][0] == 'U')
        perm = PTE_U;
    if (argv[2][0] == '0') // clear the permission
        *pte = *pte & ~perm;
    else // set the permission
        *pte = *pte | perm;

    cprintf("%x after setperm: ", addr);
    printPermission(pte);

    return 0;
}
```

6. To dump the content of the virtual page, we can use dumpv function which works as follows:

```
// dump the contents of the given virtual address range
int mon_dumpv(int argc, char **argv, struct Trapframe *tf)
{
    // sanity check
    if (argc == 1)
    {
        cprintf("The function takes two arguments - beginning address
and end address\n");
        return 0;
    }
    // convert the arguments to integers
    uint32_t start = convert2int(argv[1]);
    uint32_t end = convert2int(argv[2]);

    // check the validity of the addresses
    if (start > end)
    {
        cprintf("The start address should be less than the end
address\n");
        return 0;
    }

    // check if the addresses are in the virtual address range
    if (start < UTOP)
    {
        cprintf("The start address should be greater than UTOP\n");
        return 0;
    }

    if (end < UTOP)
    {
        cprintf("The end address should be greater than UTOP\n");
        return 0;
    }
```

```
    cprintf("start address: %x, end address: %x\n", start, end);
    // loop through the virtual address range and print the contents
    while (start <= end)
    {
        cprintf("Virtual Address: %x contains %x %x %x %x \r \n",
start, *(uint32_t *)start, *(uint32_t *)(start + 1), *(uint32_t
*)(start + 2), *(uint32_t *)(start + 3));
        start += 4;
    }
    return 0;
    // test -- dumpv 0xf0110000 0xf011a000
}
```

7. Similarly, we just translate the given pa to va and use the same logic to dump the content of the given physical address.

```
// dump the contents of the given physical address range
int mon_dumpp(int argc, char **argv, struct Trapframe *tf)
{
    // sanity check
    if (argc == 1)
    {
        cprintf("The function takes two arguments - beginning address
and end address\n");
        return 0;
    }
    // convert the physical address to virtual address
    uint32_t start = physical2virtual(convert2int(argv[1]));
    uint32_t end = physical2virtual(convert2int(argv[2]));

    cprintf("start address: %x, end address: %x\n", start, end);
    // loop through the physical address range and print the contents
    while (start <= end)
    {
        cprintf("Physical Address: %x contains %x %x %x %x \r \n",
start, *(uint32_t *)start, *(uint32_t *)(start + 1), *(uint32_t
*)(start + 2), *(uint32_t *)(start + 3));
        start += 4;
    }
    return 0;
    // test -- dumpp 0x100000 0x100001
}
```

8. To debug, we introduce the functions: pagesize (checks the page size), loadv and loadp (modify data at the byte granularity for the given virtual or physical address).

```
// function to print the page size of the given virtual address - for
debugging
int mon_pagesize(int argc, char **argv, struct Trapframe *tf)
{
    // sanity check
    if (argc == 1)
    {
        cprintf("The function takes one argument - address\n");
        return 0;
    }
    // convert the address to integer
    uint32_t addr = convert2int(argv[1]);

    // get the page table entry
    pte_t *pte = pgdir_walk(kern_pgdir, (void *)addr, 1);
    if (!pte)
        panic("page table entry not found - out of memory?");

    // print the page size
    if (*pte & PTE_PS)
        cprintf("Page size: 4MB \r \n");
    else
        cprintf("Page size: 4KB \r \n");

    return 0;
    // test -- pagesize 0xf0110000
}

// Modify a byte at given virtual memory
int mon_loadv(int argc, char **argv, struct Trapframe *tf)
{
    // sanity check
    if (argc == 1)
    {
        cprintf("The function takes three arguments - address, value and
size\n");
        return 0;
    }
    // convert the address to integer
    uint32_t addr = convert2int(argv[1]);
    // convert the value to integer
    uint32_t value = convert2int(argv[2]);
    // convert the size to integer
    uint32_t size = convert2int(argv[3]);

    // get the page table entry
    pte_t *pte = pgdir_walk(kern_pgdir, (void *)addr, 1);
    if (!pte)
        panic("page table entry not found - out of memory?");

    // check if the page is present
    if (!(*pte & PTE_P))
        panic("page not present");
```

```
    // check if the page is writable
    if (!(*pte & PTE_W))
        panic("page not writable");

    // check if the size is valid
    if (size != 1 && size != 2 && size != 4)
        panic("invalid size");

    // check if the address is aligned
    if (addr % size != 0)
        panic("address not aligned");

    // check if the value is valid
    if (size == 1 && value > 0xff)
        panic("invalid value");
    if (size == 2 && value > 0xffff)
        panic("invalid value");
    if (size == 4 && value > 0xffffffff)
        panic("invalid value");

    // modify the byte
    *(uint32_t *)addr = value;

    return 0;
    // test -- loadv 0xf0110000 0x12345678 4
}

// Modify a byte at given physical memory
int mon_loadp(int argc, char **argv, struct Trapframe *tf)
{
    // sanity check
    if (argc == 1)
    {
        cprintf("The function takes three arguments - address, value and 
size\n");
        return 0;
    }
    // convert the address to integer
    uint32_t addr = physical2virtual(convert2int(argv[1]));
    // convert the value to integer
    uint32_t value = convert2int(argv[2]);
    // convert the size to integer
    uint32_t size = convert2int(argv[3]);

    // get the page table entry
    pte_t *pte = pgdir_walk(kern_pgdir, (void *)addr, 1);
    if (!pte)
        panic("page table entry not found - out of memory?");

    // check if the page is present
    if (!(*pte & PTE_P))
        panic("page not present");

    // check if the page is writable
```

```
    if (!(*pte & PTE_W))
        panic("page not writable");

    // check if the size is valid
    if (size != 1 && size != 2 && size != 4)
        panic("invalid size");

    // check if the address is aligned
    if (addr % size != 0)
        panic("address not aligned");

    // check if the value is valid
    if (size == 1 && value > 0xff)
        panic("invalid value");
    if (size == 2 && value > 0xffff)
        panic("invalid value");
    if (size == 4 && value > 0xffffffff)
        panic("invalid value");

    // modify the byte
    *(uint32_t *)addr = value;

    return 0;
    // test -- loadp 0xf0110000 0x12345678 4
}
```

EXERCISE 8

We need to make a buddy allocator to implement this. Memory can be divided into fixed size blocks that are all powers of 2 in size. We store this structure in an array of linked lists where the index of the array allows fast access. When a memory allocation request is made, the allocator searches for the smallest free block of memory that can satisfy the request. If the block is larger than the requested size, the allocator splits the block into two "buddies", which sub-blocks of equal size. One of the buddies is allocated to the request, and the other buddy remains free. The allocator then updates the structure to reflect the new allocation. When a memory deallocation request is made, the allocator merges the freed block with its buddy to form a larger block. The allocator continues merging the resulting larger blocks with their buddies until a block of the maximum size is formed, or until a buddy cannot be found.