Волохань Николай КН-403
Лабораторная работа №2

Упражнение 1. В файле `kern/pmap.c` необходимо реализовать код для следующих функций (возможно, в указанном порядке).

`boot_alloc()`
`mem_init()` (только до вызова `check_page_free_list(1)`)
`page_init()`
`page_alloc()`
`page_free()`

`check_page_free_list()` и `check_page_alloc()` протестируйте свой физический распределитель страниц. Вы должны загрузить JOS и посмотреть, `check_page_alloc()` сообщает ли об успехе. Исправьте свой код так, чтобы он проходил. Возможно, вам будет полезно добавить свои собственные `assert()` файлы, чтобы убедиться в правильности ваших предположений.

boot_alloc

boot_alloc — это "распределитель памяти". Виртуальные адреса от 0xf0000000 до 0xf03fffff сопоставляются физическим адресам от 0x00000000 до 0x003fffff, и это диапазон адресов, которым boot_alloc предполагается управлять.

boot_alloc отслеживает, что выделено, а что нет с помощью переменной static char *nextfree. Адреса, которые меньше nextfree, считаются выделенными, а адреса, равные или выше nextfree (до 0xf03fffff), считаются нераспределенными. Как следствие, nextfree указывает на первый свободный адрес. Таким образом, "Выделение одной страницы" означает добавление 4096 к адресу, который хранится в nextfree (nextfree += 4096). Когда мы получим запрос на выделение, скажем, 3 страниц, мы сохраним текущее значение nextfree в result, добавим 3 * 4096 к nextfree и вернем result. таким образом, область памяти между result и nextfree зарезервирована для того, кто вызывал boot_alloc. Также стоит упомянуть, что адреса nextfree и result должны быть кратны 4096 (размер страницы). Таким образом, первый адрес, возвращаемый этой функцией, равен 0xf0119000).

```
104         // LAB 2: Your code here.
105         result = nextfree;
106         nextfree = ROUNDUP(result + n, PGSIZE);
107         if ((uintptr_t) nextfree >= KERNBASE + PTSIZE) {
108                 cprintf("boot_alloc: out of memory\n");
109                 panic("boot_alloc: failed to allocate %d bytes", n);
110         }
111         return result;
112 }
```

mem_init

Уберем строку panic и добавим код инициализации страниц:

```
132         // Remove this line when you're ready to test this function.
133         //panic("mem_init: This function is not finished\n");
134         pages = (struct PageInfo *)boot_alloc(sizeof(struct
    PageInfo) * npages);
```

page_init

Цель этой функции - создать и инициализировать структуру данных, которая будет отслеживать метаданные всех свободных фреймов страниц (физических страниц) в системе. У нас уже есть aray, который содержит метаданные всех фреймов страницы struct PageInfo *pages. Итак, мы будем использовать элементы этого массива и добавим все свободные фреймы страницы в связанный список, представленный static struct PageInfo *page_free_list

Как мы узнаем, какие фреймы страниц свободны в расширенной памяти (физическая память выше 1 M)?

Как отмечалось ранее, boot_alloc отслеживает всю выделенную память, начиная с 0xf0119000 и если мы вызовем boot_alloc(0) мы получим первый адрес, который не был выделен boot_alloc. boot_alloc возвращает виртуальные адреса, а нас интересует физический адрес фрейма страницы, поэтому нам нужно будет использовать PADDR макрос для преобразования его в физический адрес, а затем разделить его на размер страницы PGSIZE (4096), чтобы получить соответствующий индекс в pages массив.

```
239 void
240 page_init(void)
241 {
242         // The example code here marks all physical pages as free.
243         // However this is not truly the case.  What memory is free?
244         //  1) Mark physical page 0 as in use.
245         //     This way we preserve the real-mode IDT and BIOS
    structures
246         //     in case we ever need them.  (Currently we don't,
    but...)
247         pages[0].pp_ref = 1;
248         pages[0].pp_link = NULL;
249         //  2) The rest of base memory, [PGSIZE, npages_basemem *
    PGSIZE)
250         //     is free.
251         for (int i = 1 ; i < npages_basemem; i++) {
252                 pages[i].pp_ref = 0;
253                 pages[i].pp_link = page_free_list;
254                 page_free_list = &pages[i];
255         }
```

```
256         //  3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which
    must
257         //     never be allocated.
258         uint32_t first_free_pa = (uint32_t) PADDR(boot_alloc(0));
259         assert(first_free_pa % PGSIZE == 0);
260         int free_pa_pg_indx = first_free_pa / PGSIZE;
261         for (int i = npages_basemem ; i < free_pa_pg_indx; i++) {
262                 pages[i].pp_ref = 1;
263                 pages[i].pp_link = NULL;
264         }
265         //  4) Then extended memory [EXTPHYSMEM, ...).
266         //     Some of it is in use, some is free. Where is the
    kernel
267         //     in physical memory?  Which pages are already in use
    for
268         //     page tables and other data structures?
269         //
270         // Change the code to reflect this.
271         // NB: DO NOT actually touch the physical memory
    corresponding to
272         // free pages!
273         for (int i = free_pa_pg_indx; i < npages; i++) {
274                 pages[i].pp_ref = 0;
275                 pages[i].pp_link = page_free_list;
276                 page_free_list = &pages[i];
277         }
278 }
```

page_alloc и page_free

Глобальная переменная page_free_list указывает на начало связанного списка свободных страниц.

```
293 struct PageInfo *
294 page_alloc(int alloc_flags)
295 {
296         // Fill this function in
297         struct PageInfo* pp = page_free_list;
298         if (!pp) {
299                 return NULL;
300         }
301         page_free_list = pp->pp_link;
302         pp->pp_link = NULL;
303         if (alloc_flags & ALLOC_ZERO) {
304                 memset(page2kva(pp), 0, PGSIZE);
305         }
306         return pp;
307 }
```

```
313 void
314 page_free(struct PageInfo *pp)
315 {
316         // Fill this function in
317         // Hint: You may want to panic if pp->pp_ref is nonzero or
318         // pp->pp_link is not NULL.
319         assert(pp->pp_ref == 0);
320         assert(pp->pp_link == NULL);
321         pp->pp_link = page_free_list;
322         page_free_list = pp;
323 }
```

```
Booting from Hard Disk..
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
kernel panic at kern/pmap.c:712: assertion failed: page_insert(kern_pgdir, pp1,0
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Упражнение 3.Хотя GDB может обращаться к памяти QEMU только по виртуальному адресу, часто бывает полезно иметь возможность проверять физическую память при настройке виртуальной памяти. Ознакомьтесь с командами монитора QEMU из руководства по лабораторным инструментам, особенно с командой `xp`, которая позволяет проверять физическую память. Чтобы получить доступ к монитору QEMU, нажмите **`Ctrl-a c`** в терминале (та же привязка возвращается к последовательной консоли).

Используйте **`xp`** команду в мониторе QEMU и **`x`** команду в GDB для проверки памяти по соответствующим физическим и виртуальным адресам и убедитесь, что вы видите те же данные.

Наша исправленная версия QEMU предоставляет **`info pg`** команду, которая также может оказаться полезной: она показывает компактное, но подробное представление текущих таблиц страниц, включая все отображенные диапазоны памяти, разрешения и флаги. Стандартный QEMU также предоставляет **`info mem`** команду, которая показывает обзор того, какие диапазоны виртуальных адресов сопоставлены и с какими разрешениями.

Вопрос

1. Предполагая, что следующий код ядра JOS верен, какого типа должна быть переменная `x`: `uintptr_t` или `physaddr_t`?

```
    mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

В коде мы оперируем виртуальными адресами, поэтому тип x должен быть uintptr_t.

Упражнение 4. В файле `kern/pmap.c` необходимо реализовать код для следующих функций.

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

`check_page()` вызывается из `mem_init()` и проверяет ваши процедуры управления таблицами страниц. Вы должны убедиться, что она сообщает об успехе, прежде чем продолжить.

pgdir_walk

Функция pgdir_walk() используется для поиска записи страницы PTE, соответствующей виртуальному адресу va.

```
358 pte_t *
359 pgdir_walk(pde_t *pgdir, const void *va, int create)
360 {
361         // Fill this function in
362         int pde_index = PDX(va);
363         int pte_index = PTX(va);
364         pde_t *pde = &pgdir[pde_index];
365         if (!(*pde & PTE_P)) {
366                 if (create) {
367                         struct PageInfo *page =
    page_alloc(ALLOC_ZERO);
368                         if (!page) return NULL;
369 
370                         page->pp_ref++;
371                         *pde = page2pa(page) | PTE_P | PTE_U |
    PTE_W;
372                 } else {
373                         return NULL;
374                 }
375         }
376 
377         pte_t *p = (pte_t *) KADDR(PTE_ADDR(*pde));
378         return &p[pte_index];
379 }
```

boot_map_region

Задача состоит в том, чтобы сопоставить виртуальный адрес с соответствующим ему физическим адресом.

```
392 static void
393 boot_map_region(pde_t *pgdir, uintptr_t va, size_t size, physaddr_t
    pa, int perm)
394 {
395         // Fill this function in
396         int pages = PGNUM(size);
397         for (int i = 0; i < pages; i++) {
398                 pte_t *pte = pgdir_walk(pgdir, (void *)va, 1);
399                 if (!pte) {
400                         panic("boot_map_region panic: out of
    memory");
401                 }
402                 *pte = pa | perm | PTE_P;
403                 va += PGSIZE, pa += PGSIZE;
404         }
405 }
```

page_lookup

Находит запись страницы, соответствующей виртуальному адресу va, и возвращает структуру pageInfo, соответствующую записи страницы.

```
450 struct PageInfo *
451 page_lookup(pde_t *pgdir, void *va, pte_t **pte_store)
452 {
453         // Fill this function in
454         pte_t *pte = pgdir_walk(pgdir, va, 0);
455         if (!pte || !(*pte & PTE_P)) {
456                 return NULL;
457         }
458
459         if (pte_store) {
460                 *pte_store = pte;
461         }
462
463         return pa2page(PTE_ADDR(*pte));
464 }
```

page_remove

Удаляет физическое отображение страницы, соответствующее виртуальному адресу va, из таблицы страниц.

```
481 void
482 page_remove(pde_t *pgdir, void *va)
483 {
484         // Fill this function in
485         pte_t *pte;
486         struct PageInfo *page = page_lookup(pgdir, va, &pte);
487         if (!page || !(*pte & PTE_P)) {
488                 return;
489         }
490         *pte = 0;
491         page_decref(page);
492         tlb_invalidate(pgdir, va);
493 }
```

page_insert

Сопоставляет виртуальный адрес va с физической страницей, соответствующей pp.

```
432 int
433 page_insert(pde_t *pgdir, struct PageInfo *pp, void *va, int perm)
434 {
435         // Fill this function in
436         pte_t *pte = pgdir_walk(pgdir, va, 1);
437         if (!pte) {
438                 return -E_NO_MEM;
439         }
440
441         pp->pp_ref++;
442         if (*pte & PTE_P) {
443                 page_remove(pgdir, va);
444         }
445
446         *pte = page2pa(pp) | perm | PTE_P;
447         return 0;
448 }
```

```
Booting from Hard Disk..
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic at kern/pmap.c:681: assertion failed: check_va2pa(pgdir, UPAGES + i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Упражнение 5. Заполните недостающий код в `mem_init()` после вызова `check_page()`.

Теперь ваш код должен пройти проверки `check_kern_pgdir()` и `check_page_installed_pgdir()`.

```
150         //////////////////////////////////////////////////////////////
151         // Allocate an array of npages 'struct PageInfo's and store
    it in 'pages'.
152         // The kernel uses this array to keep track of physical
    pages: for
153         // each physical page, there is a corresponding struct
    PageInfo in this
154         // array.  'npages' is the number of physical pages in
    memory.  Use memset
155         // to initialize all fields of each struct PageInfo to 0.
156         // Your code goes here:
157         pages = (struct PageInfo *) boot_alloc(npages *
    sizeof(struct PageInfo));
158         uintptr_t pages_region_sz = (uintptr_t)boot_alloc(0) -
    (uintptr_t)pages;
159         memset(pages, 0, pages_region_sz);
160
```

```
177         //////////////////////////////////////////////////////////////
178         // Map 'pages' read-only by the user at linear address
    UPAGES
179         // Permissions:
180         //    - the new image at UPAGES -- kernel R, user R
181         //      (ie. perm = PTE_U | PTE_P)
182         //    - pages itself -- kernel RW, user NONE
183         // Your code goes here:
184         boot_map_region(kern_pgdir, UPAGES, PTSIZE, PADDR(pages),
    PTE_U);
185
```

```
186         //////////////////////////////////////////////////////////////
187         // Use the physical memory that 'bootstack' refers to as the
    kernel
188         // stack.  The kernel stack grows down from virtual address
    KSTACKTOP.
189         // We consider the entire range from [KSTACKTOP-PTSIZE,
    KSTACKTOP)
190         // to be the kernel stack, but break this into two pieces:
191         //     * [KSTACKTOP-KSTKSIZE, KSTACKTOP) -- backed by
    physical memory
192         //     * [KSTACKTOP-PTSIZE, KSTACKTOP-KSTKSIZE) -- not
    backed; so if
193         //       the kernel overflows its stack, it will fault
    rather than
194         //       overwrite memory.  Known as a "guard page".
195         //     Permissions: kernel RW, user NONE
196         // Your code goes here:
197         uintptr_t backed_stack = KSTACKTOP-KSTKSIZE;
198         boot_map_region(kern_pgdir, backed_stack, KSTKSIZE,
    PADDR(bootstack), PTE_W);
```

```
200         //////////////////////////////////////////////////////////////////
201         // Map all of physical memory at KERNBASE.
202         // Ie.  the VA range [KERNBASE, 2^32) should map to
203         //      the PA range [0, 2^32 - KERNBASE)
204         // We might not have 2^32 - KERNBASE bytes of physical
    memory, but
205         // we just set up the mapping anyway.
206         // Permissions: kernel RW, user NONE
207         // Your code goes here:
208         uintptr_t pa_end = 0xffffffff - KERNBASE + 1;
209         boot_map_region(kern_pgdir, KERNBASE, pa_end, 0, PTE_W);
210
```

```
running JOS: (0.8s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
nikolai@nikolai-VirtualBox:~/6.828/lab$
```

Вопросы:

2. Какие записи (строки) в каталоге страниц были заполнены на этом этапе? Какие адреса они отображают и куда указывают? Другими словами, заполните эту таблицу как можно подробнее:

Запись	Базовый виртуальный адрес	Указывает на (логически):
1023	?	Таблица страниц для верхних 4 МБ физической памяти
1022	?	?
.	?	?
.	?	?
.	?	?
2	0x00800000	?
1	0x00400000	?
0	0x00000000	[смотрите следующий вопрос]

Запись	Базовый виртуальный адрес	Указывает на (логически):
1023	0xFFC00000	Таблица страниц для верхних 4 МБ физической памяти
1022	0xFF800000	Таблица страниц для следующих 4 МБ физической памяти
…	…	…
960	0xF0000000	Таблица страниц для нижних 4 МБ физической памяти
959	0xEFC00000	Стек ядра
…	…	…

956	0xEF000000	Таблица страниц, содержащая структуру pages
…	…	…
0	0x00000000	[смотрите следующий вопрос]

3. Мы разместили ядро и пользовательскую среду в одном адресном пространстве. Почему пользовательские программы не смогут читать или записывать память ядра? Какие конкретные механизмы защищают память ядра?

Они не смогут этого сделать, потому что на страницах, принадлежащих ядру, отключен фрагмент PTE_U. Это означает, что, если пользовательская программа попытается прочитать эту страницу, процессор сгенерирует ошибку страницы, которая вернет управление обратно операционной системе.

4. Какой максимальный объем физической памяти может поддерживать данная операционная система? Почему?

Поскольку JOS использует 4 МБ свободного пространства для хранения всей информации о структуре pageInfo, каждая структура имеет размер 8B, поэтому мы можем хранить 512 тыс. Структур pageInfo. Размер страницы составляет 4 КБ, поэтому у нас может быть больше всего 512K * 4KB = 2GB физической памяти.

5. Сколько места потребуется для управления памятью, если у нас действительно был максимальный объем физической памяти? Как распределяются эти накладные расходы?

Нам нужно 4 МБ PageInfo для управления памятью и 2 МБ для таблицы страниц, 4 КБ для каталога страниц, если у нас есть 2 ГБ физической памяти.

6. Пересмотрите настройку таблицы страниц в `kern/entry.S` и `kern/entrypgdir.c`. Сразу после включения подкачки EIP по-прежнему остается небольшим числом (чуть более 1 МБ). В какой момент мы переходим к работе на EIP выше KERNBASE? Что позволяет нам продолжать выполнение с низким EIP между включением подкачки и началом работы с EIP выше KERNBASE? Почему необходим этот переход?

Переход к высокому EIP происходит при переходе к relocated. Мы можем работать с низким EIP после включения подкачки, потому что настройка каталога страниц в entrypgdir.c сопоставляет виртуальные адреса от 0 до 4 МБ с физическими от 0 до 4 МБ. Переход необходим, потому что остальная часть ядра связана по высоким адресам.