SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

BEIHANG UNIVERSITY

Edited from Dreamwave

init/init.c

```
#include <asm/asm.h>
#include <pmap.h>
#include <env.h>
#include <printf.h>
#include <kclock.h>
#include <trap.h>
extern char aoutcode[];
extern char boutcode[];

void mips_init()
{
    printf("init.c:\tmips_init() is called\n");
    mips_detect_memory();

    mips_vm_init();
    page_init();

    env_init();
    //count();
    /*you can create some processes(env) here. in terms of binary code, please
refer current directory/code_a.c
     * code_b.c*/
    //ENV_CREATE(user_pingpong);
    //ENV_CREATE(user_fktest);
    ENV_CREATE(user_icode);
    //ENV_CREATE(user_testfdsharing);
    //ENV_CREATE(user_testspawn);
    //ENV_CREATE(user_testpipe);
    //ENV_CREATE(user_testpiperace);
    ENV_CREATE(fs_serv);

    /*you may want to create process by MACRO, please read env.h file, in which
you will find it. this MACRO is very
     * interesting, have fun please*/
    trap_init();
    kclock_init();
    panic("^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^");
    while(1);
    panic("init.c:\tend of mips_init() reached!");
}

void bcopy(const void *src, void *dst, size_t len)
{
    void *max;

    max = dst + len;
    // copy machine words while possible
    while (dst + 3 < max)
    {
        *(int *)dst = *(int *)src;
        dst+=4;
        src+=4;
    }
    // finish remaining 0-3 bytes
```

```
    while (dst < max)
    {
        *(char *)dst = *(char *)src;
        dst+=1;
        src+=1;
    }
}

void bzero(void *b, size_t len)
{
    void *max;

    max = b + len;

    //printf("init.c:\tzero from %x to %x\n",(int)b,(int)max);

    // zero machine words while possible

    while (b + 3 < max)
    {
        *(int *)b = 0;
        b+=4;
    }

    // finish remaining 0-3 bytes
    while (b < max)
    {
        *(char *)b++ = 0;
    }

}
```

include/queue.h

```
#ifndef _SYS_QUEUE_H_
#define _SYS_QUEUE_H_

/*
 * This file defines three types of data structures: lists, tail queues,
 * and circular queues.
 *
 * A list is headed by a single forward pointer(or an array of forward
 * pointers for a hash table header). The elements are doubly linked
 * so that an arbitrary element can be removed without a need to
 * traverse the list. New elements can be added to the list before
 * or after an existing element or at the head of the list. A list
 * may only be traversed in the forward direction.
 *
 * A tail queue is headed by a pair of pointers, one to the head of the
 * list and the other to the tail of the list. The elements are doubly
 * linked so that an arbitrary element can be removed without a need to
 * traverse the list. New elements can be added to the list before or
 * after an existing element, at the head of the list, or at the end of
 * the list. A tail queue may only be traversed in the forward direction.
```

```
 *
 * A circle queue is headed by a pair of pointers, one to the head of the
 * list and the other to the tail of the list. The elements are doubly
 * linked so that an arbitrary element can be removed without a need to
 * traverse the list. New elements can be added to the list before or after
 * an existing element, at the head of the list, or at the end of the list.
 * A circle queue may be traversed in either direction, but has a more
 * complex end of list detection.
 *
 * For details on the use of these macros, see the queue(3) manual page.
 */

/*
 * List declarations.
 */
#define LIST_HEAD(name, type)                       \
struct name {                                   \
    struct type *lh_first;  /* first element */         \
}

#define LIST_HEAD_INITIALIZER(head)                 \
    { NULL }

#define LIST_ENTRY(type)                            \
struct {                                    \
    struct type *le_next;   /* next element */          \
    struct type **le_prev;  /* address of previous next element */  \
}

/*
 * List functions.
 */

#define LIST_EMPTY(head)    ((head)->lh_first == NULL)

#define LIST_FIRST(head)    ((head)->lh_first)

#define LIST_FOREACH(var, head, field)                  \
    for ((var) = LIST_FIRST((head));                \
        (var);                              \
        (var) = LIST_NEXT((var), field))

#define LIST_INIT(head) do {                        \
    LIST_FIRST((head)) = NULL;                  \
} while (0)

#define LIST_INSERT_AFTER(listelm, elm, field) do {         \
    if ((LIST_NEXT((elm), field) = LIST_NEXT((listelm), field)) != NULL)\
        LIST_NEXT((listelm), field)->field.le_prev =        \
            &LIST_NEXT((elm), field);               \
    LIST_NEXT((listelm), field) = (elm);                \
    (elm)->field.le_prev = &LIST_NEXT((listelm), field);        \
} while (0)

#define LIST_INSERT_BEFORE(listelm, elm, field) do {            \
    (elm)->field.le_prev = (listelm)->field.le_prev;        \
    LIST_NEXT((elm), field) = (listelm);                \
    *(listelm)->field.le_prev = (elm);              \
```

```
    (listelm)->field.le_prev = &LIST_NEXT((elm), field);        \
} while (0)

#define LIST_INSERT_HEAD(head, elm, field) do {             \
    if ((LIST_NEXT((elm), field) = LIST_FIRST((head))) != NULL) \
        LIST_FIRST((head))->field.le_prev = &LIST_NEXT((elm), field);\
    LIST_FIRST((head)) = (elm);                 \
    (elm)->field.le_prev = &LIST_FIRST((head));         \
} while (0)

#define LIST_NEXT(elm, field)   ((elm)->field.le_next)

#define LIST_REMOVE(elm, field) do {                    \
    if (LIST_NEXT((elm), field) != NULL)                \
        LIST_NEXT((elm), field)->field.le_prev =        \
            (elm)->field.le_prev;               \
    *(elm)->field.le_prev = LIST_NEXT((elm), field);        \
} while (0)

/*
 * Tail queue definitions.
 */
#define TAILQ_HEAD(name, type)                      \
struct name {                                   \
    struct type *tqh_first; /* first element */         \
    struct type **tqh_last; /* addr of last next element */     \
}

#define TAILQ_ENTRY(type)                       \
struct {                                    \
    struct type *tqe_next;  /* next element */          \
    struct type **tqe_prev; /* address of previous next element */  \
}

/*
 * Tail queue functions.
 */
#define TAILQ_INIT(head) {                      \
    (head)->tqh_first = NULL;                   \
    (head)->tqh_last = &(head)->tqh_first;              \
}

#define TAILQ_INSERT_HEAD(head, elm, field) {               \
    if (((elm)->field.tqe_next = (head)->tqh_first) != NULL)    \
        (head)->tqh_first->field.tqe_prev =         \
            &(elm)->field.tqe_next;             \
    else                                \
        (head)->tqh_last = &(elm)->field.tqe_next;      \
    (head)->tqh_first = (elm);                  \
    (elm)->field.tqe_prev = &(head)->tqh_first;         \
}

#define TAILQ_INSERT_TAIL(head, elm, field) {               \
    (elm)->field.tqe_next = NULL;                   \
    (elm)->field.tqe_prev = (head)->tqh_last;           \
    *(head)->tqh_last = (elm);                  \
    (head)->tqh_last = &(elm)->field.tqe_next;          \
}
```

```
#define TAILQ_INSERT_AFTER(head, listelm, elm, field) {         \
    if (((elm)->field.tqe_next = (listelm)->field.tqe_next) != NULL)\
        (elm)->field.tqe_next->field.tqe_prev =         \
            &(elm)->field.tqe_next;             \
    else                                \
        (head)->tqh_last = &(elm)->field.tqe_next;      \
    (listelm)->field.tqe_next = (elm);              \
    (elm)->field.tqe_prev = &(listelm)->field.tqe_next;     \
}

#define TAILQ_INSERT_BEFORE(listelm, elm, field) {          \
    (elm)->field.tqe_prev = (listelm)->field.tqe_prev;      \
    (elm)->field.tqe_next = (listelm);              \
    *(listelm)->field.tqe_prev = (elm);             \
    (listelm)->field.tqe_prev = &(elm)->field.tqe_next;     \
}

#define TAILQ_REMOVE(head, elm, field) {                \
    if (((elm)->field.tqe_next) != NULL)                \
        (elm)->field.tqe_next->field.tqe_prev =         \
            (elm)->field.tqe_prev;              \
    else                                \
        (head)->tqh_last = (elm)->field.tqe_prev;       \
    *(elm)->field.tqe_prev = (elm)->field.tqe_next;         \
}

/*
 * Circular queue definitions.
 */
#define CIRCLEQ_HEAD(name, type)                    \
struct name {                               \
    struct type *cqh_first;     /* first element */     \
    struct type *cqh_last;      /* last element */      \
}

#define CIRCLEQ_ENTRY(type)                     \
struct {                                \
    struct type *cqe_next;      /* next element */      \
    struct type *cqe_prev;      /* previous element */      \
}

/*
 * Circular queue functions.
 */
#define CIRCLEQ_INIT(head) {                        \
    (head)->cqh_first = (void *)(head);             \
    (head)->cqh_last = (void *)(head);              \
}

#define CIRCLEQ_INSERT_AFTER(head, listelm, elm, field) {       \
    (elm)->field.cqe_next = (listelm)->field.cqe_next;      \
    (elm)->field.cqe_prev = (listelm);              \
    if ((listelm)->field.cqe_next == (void *)(head))        \
        (head)->cqh_last = (elm);               \
    else                                \
        (listelm)->field.cqe_next->field.cqe_prev = (elm);  \
    (listelm)->field.cqe_next = (elm);              \
}
```

```
#define CIRCLEQ_INSERT_BEFORE(head, listelm, elm, field) {      \
    (elm)->field.cqe_next = (listelm);              \
    (elm)->field.cqe_prev = (listelm)->field.cqe_prev;      \
    if ((listelm)->field.cqe_prev == (void *)(head))        \
        (head)->cqh_first = (elm);              \
    else                                \
        (listelm)->field.cqe_prev->field.cqe_next = (elm);  \
    (listelm)->field.cqe_prev = (elm);              \
}

#define CIRCLEQ_INSERT_HEAD(head, elm, field) {             \
    (elm)->field.cqe_next = (head)->cqh_first;          \
    (elm)->field.cqe_prev = (void *)(head);             \
    if ((head)->cqh_last == (void *)(head))             \
        (head)->cqh_last = (elm);               \
    else                                \
        (head)->cqh_first->field.cqe_prev = (elm);      \
    (head)->cqh_first = (elm);                  \
}

#define CIRCLEQ_INSERT_TAIL(head, elm, field) {             \
    (elm)->field.cqe_next = (void *)(head);             \
    (elm)->field.cqe_prev = (head)->cqh_last;           \
    if ((head)->cqh_first == (void *)(head))            \
        (head)->cqh_first = (elm);              \
    else                                \
        (head)->cqh_last->field.cqe_next = (elm);       \
    (head)->cqh_last = (elm);                   \
}

#define CIRCLEQ_REMOVE(head, elm, field) {              \
    if ((elm)->field.cqe_next == (void *)(head))            \
        (head)->cqh_last = (elm)->field.cqe_prev;       \
    else                                \
        (elm)->field.cqe_next->field.cqe_prev =         \
            (elm)->field.cqe_prev;              \
    if ((elm)->field.cqe_prev == (void *)(head))            \
        (head)->cqh_first = (elm)->field.cqe_next;      \
    else                                \
        (elm)->field.cqe_prev->field.cqe_next =         \
            (elm)->field.cqe_next;              \
}
#endif  /* !_SYS_QUEUE_H_ */
```

include/pmap.h

```
#ifndef _PMAP_H_
#define _PMAP_H_

#include "types.h"
#include "queue.h"
#include "mmu.h"
#include "printf.h"

LIST_HEAD(Page_list, Page);
typedef LIST_ENTRY(Page) Page_LIST_entry_t;

struct Page {
    Page_LIST_entry_t pp_link;  /* free list link */

   // Ref is the count of pointers (usually in page table entries)
   // to this page.  This only holds for pages allocated using
   // page_alloc.  Pages allocated at boot time using pmap.c's "alloc"
   // do not have valid reference count fields.

    u_short pp_ref;
};

extern struct Page *pages;
static inline u_long
page2ppn(struct Page *pp)
{
    return pp - pages;
}

static inline u_long
page2pa(struct Page *pp)
{
    return page2ppn(pp)<<PGSHIFT;
}

static inline struct Page *
pa2page(u_long pa)
{
    if (PPN(pa) >= npage)
        panic("pa2page called with invalid pa: %x", pa);
    return &pages[PPN(pa)];
}

static inline u_long
page2kva(struct Page *pp)
{
    return KADDR(page2pa(pp));
}

static inline u_long
va2pa(Pde *pgdir, u_long va)
{
    Pte *p;
```

```
    pgdir = &pgdir[PDX(va)];
    if (!(*pgdir&PTE_V))
        return ~0;
    p = (Pte*)KADDR(PTE_ADDR(*pgdir));
    if (!(p[PTX(va)]&PTE_V))
        return ~0;
    return PTE_ADDR(p[PTX(va)]);
}

void mips_detect_memory();

void mips_vm_init();

void mips_init();
void page_init(void);
void page_check();
int page_alloc(struct Page **pp);
void page_free(struct Page *pp);
void page_decref(struct Page *pp);
int pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte);
int page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm);
struct Page* page_lookup(Pde *pgdir, u_long va, Pte **ppte);
void page_remove(Pde *pgdir, u_long va) ;
void tlb_invalidate(Pde *pgdir, u_long va);

void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm);

extern struct Page *pages;

#endif /* _PMAP_H_ */
```

mm/pmap.c

```
#include "mmu.h"
#include "pmap.h"
#include "printf.h"
#include "env.h"
#include "error.h"

/* These variables are set by mips_detect_memory() */
u_long maxpa;            /* Maximum physical address */
u_long npage;            /* Amount of memory(in pages) */
u_long basemem;          /* Amount of base memory(in bytes) */
u_long extmem;           /* Amount of extended memory(in bytes) */

Pde *boot_pgdir;

struct Page *pages;
static u_long freemem;

static struct Page_list page_free_list; /* Free list of physical pages */
```

```
/* Overview:
    Initialize basemem and npage.
    Set basemem to be 64MB, and calculate corresponding npage value.*/
void mips_detect_memory()
{
    /* Step 1: Initialize basemem.
     * (When use real computer, CMOS tells us how many kilobytes there are).
*/
    maxpa   = 0x4000000;
    npage   = 0x4000;
    basemem = 0x4000000;
    extmem  = 0;

    // Step 2: Calculate corresponding npage value.

    printf("Physical memory: %dK available, ", (int)(maxpa / 1024));
    printf("base = %dK, extended = %dK\n", (int)(basemem / 1024),
           (int)(extmem / 1024));
}

/* Overview:
    Allocate `n` bytes physical memory with alignment `align`, if `clear` is
set, clear the
    allocated memory.
    This allocator is used only while setting up virtual memory system.

   Post-Condition:
    If we're out of memory, should panic, else return this address of memory
we have allocated.*/
static void *alloc(u_int n, u_int align, int clear)
{
    extern char end[];
    u_long alloced_mem;

    /* Initialize `freemem` if this is the first time. The first virtual address
that the
     * linker did *not* assign to any kernel code or global variables. */
    if (freemem == 0) {
        freemem = (u_long)end;
    }

    /* Step 1: Round up `freemem` up to be aligned properly */
    freemem = ROUND(freemem, align);

    /* Step 2: Save current value of `freemem` as allocated chunk. */
    alloced_mem = freemem;

    /* Step 3: Increase `freemem` to record allocation. */
    freemem = freemem + n;

    /* Step 4: Clear allocated chunk if parameter `clear` is set. */
    if (clear) {
        bzero((void *)alloced_mem, n);
    }

    // We're out of memory, PANIC !!
    if (PADDR(freemem) >= maxpa) {
        panic("out of memorty\n");
```

```
        return (void *)-E_NO_MEM;//(void* ) -4
    }

    /* Step 5: return allocated chunk. */
    return (void *)alloced_mem;
}

/* Overview:
    Get the page table entry for virtual address `va` in the given
    page directory `pgdir`.
    If the page table is not exist and the parameter `create` is set to 1,
    then create it.*/
static Pte *boot_pgdir_walk(Pde *pgdir, u_long va, int create)
{

    //pgdir_entryp:虚拟地址
    //*pgdir_entryp:页目录项内容
    //页目录项或者页表项的构成:20 位物理页框号+12 位标志位
    Pde *pgdir_entryp;
    Pte *pgtable, *pgtable_entry;

    /* Step 1: Get the corresponding page directory entry and page table. */
    /* Hint: Use KADDR and PTE_ADDR to get the page table from page directory
     * entry value. */
    pgdir_entryp = &pgdir[PDX(va)];//依然是一个虚拟地址

     /* Step 2: If the corresponding page table is not exist and parameter
`create`
     * is set, create one. And set the correct permission bits for this new
page
     * table. */
    if (create && (!(*pgdir_entryp & PTE_V)) ) {//如果有效位是 0 且 create 被设置,
那么创建一页
        *pgdir_entryp = PADDR((Pde)alloc(BY2PG,BY2PG,1) | PTE_V);
    }

    /* Step 3: Get the page table entry for `va`, and return it. */
    pgtable = (Pte*)KADDR(PTE_ADDR(*pgdir_entryp));//PET_ADDR(pte)实际上只是将页
表项的 12 位标志位抹掉
    pgtable_entry = &pgtable[PTX(va)];
    return pgtable_entry;
}

/*Overview:
    Map [va, va+size) of virtual address space to physical [pa, pa+size) in
the page
    table rooted at pgdir.
    Use permission bits `perm|PTE_V` for the entries.
    Use permission bits `perm` for the entries.

  Pre-Condition:
    Size is a multiple of BY2PG.*/
void boot_map_segment(Pde *pgdir, u_long va, u_long size, u_long pa, int perm)
{
    //将物理地址写入对应的页表项中
    int i, va_temp;
```

```
    Pte *pgtable_entry;

    /* Step 1: Check if `size` is a multiple of BY2PG. */
    if (size&0xfff)
        size = ROUND(size,BY2PG);
    //assert(size%BY2PG==0);

    /* Step 2: Map virtual address space to physical address. */
     /* Hint: Use `boot_pgdir_walk` to get the page table entry of virtual
address `va`. */
    for (i = 0;i<VPN(size);i++) {
        va_temp = va+(i<<PGSHIFT);
        pgtable_entry = boot_pgdir_walk(pgdir,va_temp,1);
        *pgtable_entry = PTE_ADDR((pa+(i<<PGSHIFT))) | (perm|PTE_V);
    }

}

/* Overview:
    Set up two-level page table.

   Hint:
    You can get more details about `UPAGES` and `UENVS` in include/mmu.h. */
void mips_vm_init()
{
    extern char end[];
    extern int mCONTEXT;
    extern struct Env *envs;

    Pde *pgdir;
    u_int n;

    /* Step 1: Allocate a page for page directory(first level page table). */
    pgdir = alloc(BY2PG, BY2PG, 1);//分配页目录
    printf("to memory %x for struct page directory.\n", freemem);
    mCONTEXT = (int)pgdir;

    boot_pgdir = pgdir;

    /* Step 2: Allocate proper size of physical memory for global array `pages`,
     * for physical memory management. Then, map virtual address `UPAGES` to
       * physical address `pages` allocated before. For consideration of
alignment,
     * you should round up the memory size before map. */
    pages = (struct Page *)alloc(npage * sizeof(struct Page), BY2PG, 1);
    printf("to memory %x for struct Pages.\n", freemem);
    n = ROUND(npage * sizeof(struct Page), BY2PG);
    boot_map_segment(pgdir, UPAGES, n, PADDR(pages), PTE_R);

    /* Step 3, Allocate proper size of physical memory for global array `envs`,
     * for process management. Then map the physical address to `UENVS`. */
    envs = (struct Env *)alloc(NENV * sizeof(struct Env), BY2PG, 1);
    n = ROUND(NENV * sizeof(struct Env), BY2PG);
    boot_map_segment(pgdir, UENVS, n, PADDR(envs), PTE_R);
    printf("pmap.c:\t mips vm init success\n");
}

/*Overview:
```

```
    Initialize page structure and memory free list.
    The `pages` array has one `struct Page` entry per physical page. Pages
    are reference counted, and free pages are kept on a linked list.
  Hint:
    Use `LIST_INSERT_HEAD` to insert something to list.*/
void
page_init(void)
{
    /* Step 1: Initialize page_free_list. */
    /* Hint: Use macro `LIST_INIT` defined in include/queue.h. */
    extern char end[];
    LIST_INIT(&page_free_list);
    /* Step 2: Align `freemem` up to multiple of BY2PG. */
    /* In fact ROUND(a,n) = ceiling(a/n)*n,
       For example, a=5,n=4,so a/n=1.25, and ceiling(1.25)=2,
       ROUND(5,4) = 2*4 = 8
       Another example, a=15,n=4,so a/n=3.75, and ceiling(15,4)=4,
       ROUND(15,4) = 4*4 = 16;
       By Contrast, ROUNDDOWN(a,n) = floor(a/n)*n;
    */
    freemem = ROUND(freemem,BY2PG);
    /* Step 3: Mark all memory blow `freemem` as used(set `pp_ref`
     * filed to 1) */
    u_long used = PPN(PADDR(freemem));
    int i=0;
    for (i=0;i<used;i++) {
        pages[i].pp_ref=1;
    }
    /* Step 4: Mark the other memory as free. */
    for (i=used;i<npage;i++) {
        pages[i].pp_ref=0;
        LIST_INSERT_HEAD(&page_free_list,&pages[i],pp_link);
    }
}
void count(void) {
    int i=0;
    struct Page*p;
    p = LIST_FIRST(&page_free_list);
    while (p!=NULL) {
        i++;
        p = LIST_NEXT(p,pp_link);
    }
    printf("%d pages are free\n",i);
}

/*Overview:
    Allocates a physical page from free memory, and clear this page.

  Post-Condition:
    If failed to allocate a new page(out of memory(there's no free page)),
    return -E_NO_MEM.
    Else, set the address of allocated page to *pp, and returned 0.

  Note:
    Does NOT increment the reference count of the page - the caller must do
    these if necessary (either explicitly or via page_insert).

  Hint:
```

```
    Use LIST_FIRST and LIST_REMOVE defined in include/queue.h .*/
int
page_alloc(struct Page **pp)
{
    struct Page *ppage_temp;

    /* Step 1: Get a page from free memory. If fails, return the error code.*/
    if (LIST_EMPTY(&page_free_list)) {
        *pp = 0;
        return -E_NO_MEM;
    }
    ppage_temp = LIST_FIRST(&page_free_list);
    LIST_REMOVE(ppage_temp,pp_link);

    /* Step 2: Initialize this page.
     * Hint: use `bzero`. */
    bzero((void*)page2kva(ppage_temp),BY2PG);//清空的是对应的 4k 空间的那一页，而不是
我们存储页信息的结构体
    *pp = ppage_temp;
    return 0;
}

/*Overview:
    Release a page, mark it as free if it's `pp_ref` reaches 0.
  Hint:
    When to free a page, just insert it to the page_free_list.*/
void
page_free(struct Page *pp)
{
    /* Step 1: If there's still virtual address refers to this page, do nothing.
*/
    if (pp->pp_ref > 0) return;
    /* Step 2: If the `pp_ref` reaches to 0, mark this page as free and return.
*/
    if (pp->pp_ref==0) {
        LIST_INSERT_HEAD(&page_free_list,pp,pp_link);
        return;
    }

    /* If the value of `pp_ref` less than 0, some error must occurred before,
     * so PANIC !!! */
    if (pp->pp_ref<0) panic("cgh:pp->pp_ref is less than zero\n");
}

/*Overview:
    Given `pgdir`, a pointer to a page directory, pgdir_walk returns a pointer
    to the page table entry (with permission PTE_R|PTE_V) for virtual address
'va'.
    给定页目录地址，pgdir_walk 函数返回一个对应于 va 的且有效位被置为 PTE_R|PTE_V 的指向
    页表项的指针

  Pre-Condition:
    The `pgdir` should be two-level page table structure.

  Post-Condition:
    If we're out of memory, return -E_NO_MEM.
     Else, we get the page table entry successfully, store the value of page
```

```
table
    entry to *ppte, and return 0, indicating success.

  Hint:
     We use a two-level pointer to store page table entry and return a state
code to indicate
    whether this function execute successfully or not.
    This function have something in common with function `boot_pgdir_walk`.*/
int
pgdir_walk(Pde *pgdir, u_long va, int create, Pte **ppte)
{
    Pde *pgdir_entryp;
    Pte *pgtable;
    struct Page *ppage;

    /* Step 1: Get the corresponding page directory entry and page table. */
    pgdir_entryp = &pgdir[PDX(va)];
    /* Step 2: If the corresponding page table is not exist(valid) and parameter
`create`
     * is set, create one. And set the correct permission bits for this new
page
     * table.
     * When creating new page table, maybe out of memory. */
    if (create && (*pgdir_entryp & PTE_V)==0) {
        if (page_alloc(&ppage)) {
            *ppte = 0;
            return -E_NO_MEM;//没有空间了，则返回失败信息
        }
        ppage->pp_ref++;//让页引用变为 1
        *pgdir_entryp = PADDR( (Pde)page2kva(ppage) |(PTE_R | PTE_V) );//设置对
应的标志位，这里的写入只是写入了一个 32 位值
    }

    /* Step 3: Set the page table entry to `*ppte` as return value. */
    if((*pgdir_entryp)==0) {
        *ppte = 0;
        return 0;
    }
    pgtable = (Pte*)KADDR(PTE_ADDR(*pgdir_entryp));
    *ppte = &pgtable[PTX(va)];
    return 0;
}

/*Overview:
    Map the physical page 'pp' at virtual address 'va'.
    The permissions (the low 12 bits) of the page table entry should be set to
'perm|PTE_V'.
    将物理页 pp 映射到虚拟地址 va

  Post-Condition:
    Return 0 on success
    Return -E_NO_MEM, if page table couldn't be allocated

  Hint:
     If there is already a page mapped at `va`, call page_remove() to release
this mapping.
    The `pp_ref` should be incremented if the insertion succeeds.*/
```

```
int
page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
{
    u_int PERM;
    Pte *pgtable_entry;
    PERM = perm | PTE_V;

    /* Step 1: Get corresponding page table entry. */
    pgdir_walk(pgdir, va, 0, &pgtable_entry);

    if (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
        if (pa2page(*pgtable_entry) != pp) {
            page_remove(pgdir, va);
        } else  {
            tlb_invalidate(pgdir, va);
            *pgtable_entry = (page2pa(pp) | PERM);
            return 0;
        }
    }

    /* Step 2: Update TLB. */
    tlb_invalidate(pgdir, va);

    /* Step 3: Do check, re-get page table entry to validate the insertion. */
    if (pgdir_walk(pgdir, va, 1, &pgtable_entry) != 0) {
        return -E_NO_MEM;    // panic ("page insert failed .\n");
    }

    *pgtable_entry = (page2pa(pp) | PERM);
    pp->pp_ref++;
    return 0;
}

/*Overview:
    Look up the Page that virtual address `va` map to.

  Post-Condition:
    Return a pointer to corresponding Page, and store it's page table entry to
*ppte.
    If `va` doesn't mapped to any Page, return NULL.*/
struct Page *
page_lookup(Pde *pgdir, u_long va, Pte **ppte)
{
    struct Page *ppage;
    Pte *pte;

    /* Step 1: Get the page table entry. */
    pgdir_walk(pgdir, va, 0, &pte);

    /* Hint: Check if the page table entry doesn't exist or is not valid. */
    if (pte == 0) {
        return 0;
    }
    if ((*pte & PTE_V) == 0) {
        return 0;    //the page is not in memory.
    }

    /* Step 2: Get the corresponding Page struct. */
```

```
    /* Hint: Use function `pa2page`, defined in include/pmap.h . */
    ppage = pa2page(*pte);
    if (ppte) {
        *ppte = pte;
    }

    return ppage;
}

// Overview:
//  Decrease the `pp_ref` value of Page `*pp`, if `pp_ref` reaches to 0, free
this page.
void page_decref(struct Page *pp) {
    if(--pp->pp_ref == 0) {
        page_free(pp);
    }
}

// Overview:
//  Unmaps the physical page at virtual address `va`.
void
page_remove(Pde *pgdir, u_long va)
{
    Pte *pagetable_entry;
    struct Page *ppage;

    /* Step 1: Get the page table entry, and check if the page table entry is
valid. */
    ppage = page_lookup(pgdir, va, &pagetable_entry);

    if (ppage == 0) {
        return;
    }

     /* Step 2: Decrease `pp_ref` and decide if it's necessary to free this
page. */

    /* Hint: When there's no virtual address mapped to this page, release it.
*/
    ppage->pp_ref--;
    if (ppage->pp_ref == 0) {
        page_free(ppage);
    }

    /* Step 3: Update TLB. */
    *pagetable_entry = 0;
    tlb_invalidate(pgdir, va);
    return;
}

// Overview:
//  Update TLB.
void
tlb_invalidate(Pde *pgdir, u_long va)
{
    if (curenv) {
        tlb_out(PTE_ADDR(va) | GET_ENV_ASID(curenv->env_id));
```

```
    } else {
        tlb_out(PTE_ADDR(va));
    }
}

void
page_check(void)
{
    struct Page *pp, *pp0, *pp1, *pp2;
    struct Page_list fl;

    // should be able to allocate three pages
    pp0 = pp1 = pp2 = 0;
    assert(page_alloc(&pp0) == 0);
    assert(page_alloc(&pp1) == 0);
    assert(page_alloc(&pp2) == 0);

    assert(pp0);
    assert(pp1 && pp1 != pp0);
    assert(pp2 && pp2 != pp1 && pp2 != pp0);

    // temporarily steal the rest of the free pages
    fl = page_free_list;
    // now this page_free list must be empty!!!!
    LIST_INIT(&page_free_list);

    // should be no free memory
    assert(page_alloc(&pp) == -E_NO_MEM);

    // there is no free memory, so we can't allocate a page table
    assert(page_insert(boot_pgdir, pp1, 0x0, 0) < 0);

    // free pp0 and try again: pp0 should be used for page table
    page_free(pp0);
    assert(page_insert(boot_pgdir, pp1, 0x0, 0) == 0);
    assert(PTE_ADDR(boot_pgdir[0]) == page2pa(pp0));

    printf("va2pa(boot_pgdir, 0x0) is %x\n",va2pa(boot_pgdir, 0x0));
    printf("page2pa(pp1) is %x\n",page2pa(pp1));

    assert(va2pa(boot_pgdir, 0x0) == page2pa(pp1));
    assert(pp1->pp_ref == 1);

    // should be able to map pp2 at BY2PG because pp0 is already allocated for
page table
    assert(page_insert(boot_pgdir, pp2, BY2PG, 0) == 0);
    assert(va2pa(boot_pgdir, BY2PG) == page2pa(pp2));
    assert(pp2->pp_ref == 1);

    // should be no free memory
    assert(page_alloc(&pp) == -E_NO_MEM);

    printf("start page_insert\n");
    // should be able to map pp2 at BY2PG because it's already there
    assert(page_insert(boot_pgdir, pp2, BY2PG, 0) == 0);
    assert(va2pa(boot_pgdir, BY2PG) == page2pa(pp2));
    assert(pp2->pp_ref == 1);
```

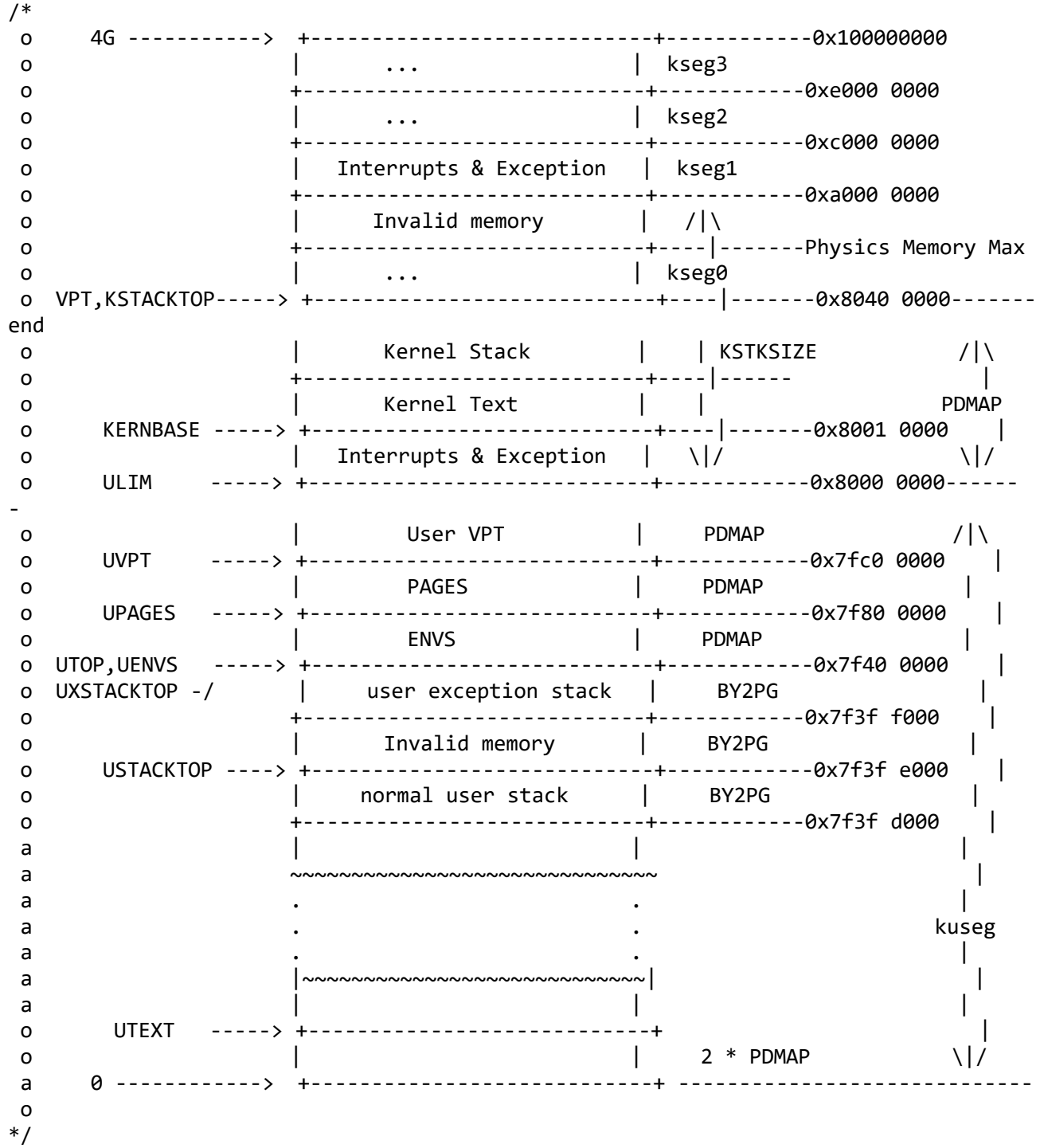
```
// pp2 should NOT be on the free list
// could happen in ref counts are handled sloppily in page_insert
assert(page_alloc(&pp) == -E_NO_MEM);

// should not be able to map at PDMAP because need free page for page table
assert(page_insert(boot_pgdir, pp0, PDMAP, 0) < 0);

// insert pp1 at BY2PG (replacing pp2)
assert(page_insert(boot_pgdir, pp1, BY2PG, 0) == 0);

// should have pp1 at both 0 and BY2PG, pp2 nowhere, ...
assert(va2pa(boot_pgdir, 0x0) == page2pa(pp1));
assert(va2pa(boot_pgdir, BY2PG) == page2pa(pp1));
// ... and ref counts should reflect this
assert(pp1->pp_ref == 2);
printf("pp2->pp_ref %d\n",pp2->pp_ref);
assert(pp2->pp_ref == 0);
printf("end page_insert\n");

// pp2 should be returned by page_alloc
assert(page_alloc(&pp) == 0 && pp == pp2);

// unmapping pp1 at 0 should keep pp1 at BY2PG
page_remove(boot_pgdir, 0x0);
assert(va2pa(boot_pgdir, 0x0) == ~0);
assert(va2pa(boot_pgdir, BY2PG) == page2pa(pp1));
assert(pp1->pp_ref == 1);
assert(pp2->pp_ref == 0);

// unmapping pp1 at BY2PG should free it
page_remove(boot_pgdir, BY2PG);
assert(va2pa(boot_pgdir, 0x0) == ~0);
assert(va2pa(boot_pgdir, BY2PG) == ~0);
assert(pp1->pp_ref == 0);
assert(pp2->pp_ref == 0);

// so it should be returned by page_alloc
assert(page_alloc(&pp) == 0 && pp == pp1);

// should be no free memory
assert(page_alloc(&pp) == -E_NO_MEM);

// forcibly take pp0 back
assert(PTE_ADDR(boot_pgdir[0]) == page2pa(pp0));
boot_pgdir[0] = 0;
assert(pp0->pp_ref == 1);
pp0->pp_ref = 0;

// give free list back
page_free_list = fl;

// free the pages we took
page_free(pp0);
page_free(pp1);
page_free(pp2);

/*u_long* va = 0x12450;
u_long* pa;
```

```
    page_insert(boot_pgdir,pp,va,PTE_R);
    pa = va2pa(boot_pgdir,va);
    printf("va: %x -> pa: %x\n",va,pa);

    *va = 0x88888;
    printf("va value: %x\n",*va);
    printf("pa value: %x\n",*((u_long*)((u_long)pa+(u_long)ULIM)));*/
    printf("page_check() succeeded!\n");
}
void pageout(int va, int context)
{
    u_long r;
    struct Page *p = NULL;

    if (context < 0x80000000) {
        panic("tlb refill and alloc error!");
    }

    if ((va > 0x7f400000) && (va < 0x7f800000)) {
        panic(">>>>>>>>>>>>>>>>>>>>>>>>it's env's zone");
    }

    if (va < 0x10000) {
        panic("^^^^^^TOO LOW^^^^^^^^^^");
    }

    if ((r = page_alloc(&p)) < 0) {
        panic ("page alloc error!");
    }

    p->pp_ref++;

    page_insert((Pde *)context, p, VA2PFN(va), PTE_R);
    printf("pageout:\t@@@___0x%x___@@@  ins a page \n", va);
}
```

include/mmu.h

```
#ifndef _MMU_H_
#define _MMU_H_

/*
 * This file contains:
 *
 *  Part 1.  MIPS definitions.
 *  Part 2.  Our conventions.
 *  Part 3.  Our helper functions.
 */

/*
 * Part 1.  MIPS definitions.
 */
#define BY2PG      4096       // bytes to a page
#define PDMAP      (4*1024*1024)   // bytes mapped by a page directory entry
#define PGSHIFT    12
#define PDSHIFT    22      // log2(PDMAP)
#define PDX(va)    ((((u_long)(va))>>22) & 0x03FF)
#define PTX(va)    ((((u_long)(va))>>12) & 0x03FF)
#define PTE_ADDR(pte)   ((u_long)(pte)&~0xFFF)

// page number field of address
#define PPN(va)    (((u_long)(va))>>12)
#define VPN(va)    PPN(va)

#define  VA2PFN(va)        (((u_long)(va))  &  0xFFFFF000  )  //  va  2  PFN  for
EntryLo0/1
#define PTE2PT     1024
//$#define VA2PDE(va)       (((u_long)(va)) & 0xFFC00000 ) // for context

/* Page Table/Directory Entry flags
 *   these are defined by the hardware
 */
#define PTE_G      0x0100  // Global bit
#define PTE_V      0x0200  // Valid bit
#define PTE_R       0x0400  // Dirty bit ,'0' means only read ,otherwise make
interrupt
#define PTE_D      0x0002  // fileSystem Cached is dirty
#define PTE_COW    0x0001  // Copy On Write
#define PTE_UC     0x0800  // unCached
#define PTE_LIBRARY    0x0004  // share memmory
```

```
/*
 * Part 2.  Our conventions.
 */
```



```
#define KERNBASE 0x80010000

#define VPT (ULIM + PDMAP )
#define KSTACKTOP (VPT-0x100)
#define KSTKSIZE (8*BY2PG)
#define ULIM 0x80000000

#define UVPT (ULIM - PDMAP)
#define UPAGES (UVPT - PDMAP)
#define UENVS (UPAGES - PDMAP)

#define UTOP UENVS
#define UXSTACKTOP (UTOP)
#define TIMESTACK 0x82000000

#define USTACKTOP (UTOP - 2*BY2PG)
#define UTEXT 0x00400000

#define E_UNSPECIFIED   1   // Unspecified or unknown problem
#define E_BAD_ENV       2       // Environment doesn't exist or otherwise
               // cannot be used in requested action
#define E_INVAL     3   // Invalid parameter
#define E_NO_MEM    4   // Request failed due to memory shortage
#define E_NO_FREE_ENV   5       // Attempt to create a new environment beyond
               // the maximum allowed
#define E_IPC_NOT_RECV  6   // Attempt to send to env that is not recving.

// File system error codes -- only seen in user-level
#define E_NO_DISK   7   // No free space left on disk
#define E_MAX_OPEN  8   // Too many files are open
#define E_NOT_FOUND 9   // File or block not found
#define E_BAD_PATH  10  // Bad path
#define E_FILE_EXISTS   11  // File already exists
#define E_NOT_EXEC  12  // File not a valid executable

#define MAXERROR 12

#ifndef __ASSEMBLER__

/*
 * Part 3.  Our helper functions.
 */
#include "types.h"
void bcopy(const void *, void *, size_t);
void bzero(void *, size_t);

extern char bootstacktop[], bootstack[];

extern u_long npage;

typedef u_long Pde;
typedef u_long Pte;

extern volatile Pte* vpt[];
extern volatile Pde* vpd[];

#define PADDR(kva)                      \
({                              \
```

```
    u_long a = (u_long) (kva);              \
    if (a < ULIM)                   \
        panic("PADDR called with invalid kva %08lx", a);\
    a - ULIM;                       \
})

// translates from physical address to kernel virtual address
#define KADDR(pa)                       \
({                              \
    u_long ppn = PPN(pa);                   \
    if (ppn >= npage)                   \
        panic("KADDR called with invalid pa %08lx", (u_long)pa);\
    (pa) + ULIM;                    \
})

#define assert(x)   \
    do {    if (!(x)) panic("assertion failed: %s", #x); } while (0)

#define TRUP(_p)                        \
({                              \
    register typeof((_p)) __m_p = (_p);         \
    (u_int) __m_p > ULIM ? (typeof(_p)) ULIM : __m_p;   \
})


extern void tlb_out(u_int entryhi);
#endif //!__ASSEMBLER__
#endif // !_MMU_H_
```

Page insert and Page remove

```
1 // Overview:
2 // Map the physical page 'pp' at virtual address 'va'.
3 // The permissions (the low 12 bits) of the page table entry
4 // should be set to 'perm|PTE_V'.
5 //
6 // Post-Condition:
7 // Return 0 on success
8 // Return -E_NO_MEM, if page table couldn't be allocated
9 //
10 // Hint:
11 // If there is already a page mapped at `va`, call page_remove()
12 // to release this mapping.The `pp_ref` should be incremented
13 // if the insertion succeeds.

14 int
15 page_insert(Pde *pgdir, struct Page *pp, u_long va, u_int perm)
16 {
17 ^^Iu_int PERM;
18 ^^IPte *pgtable_entry;
19 ^^IPERM = perm | PTE_V;
20
21 ^^I/* Step 1: Get corresponding page table entry. */
22 ^^Ipgdir_walk(pgdir, va, 0, &pgtable_entry);
23
24 ^^Iif (pgtable_entry != 0 && (*pgtable_entry & PTE_V) != 0) {
25 ^^I^^Iif (pa2page(*pgtable_entry) != pp) {
26 ^^I^^I^^Ipage_remove(pgdir, va);
27 ^^I^^I}
28 ^^I^^Ielse{
29 ^^I^^I^^Itlb_invalidate(pgdir, va);
30 ^^I^^I^^I*pgtable_entry = (page2pa(pp) | PERM);
31 ^^I^^I^^Ireturn 0;
32 ^^I^^I}
33 ^^I}
34
35 ^^I/* Step 2: Update TLB. */
36 ^^Itlb_invalidate(pgdir, va);
37
38 ^^I/* Step 3: Do check, re-get page table entry to validate
39 ^^I * the insertion. */
40 ^^Iif (pgdir_walk(pgdir, va, 1, &pgtable_entry) != 0) {
41 ^^I^^Ireturn -E_NO_MEM; // panic ("page insert failed .\n");
```

```
42 ^^I}
43
44 ^^I*pgtable_entry = (page2pa(pp) | PERM);
45 ^^Ipp->pp_ref++;
46 ^^Ireturn 0;
47 }
```

TLB 汇编函数

```
1 #include <asm/regdef.h>
2 #include <asm/cp0regdef.h>
3 #include <asm/asm.h>
4
5   LEAF(tlb_out)
6   nop
7   mfc0   k1,CP0_ENTRYHI
8   mtc0   a0,CP0_ENTRYHI
9   nop
10  tlbp
11  nop
12  nop
13  nop
14  nop
15  mfc0   k0,CP0_INDEX
16  bltz   k0,NOFOUND
17  nop
18  mtc0   zero,CP0_ENTRYHI
19  mtc0   zero,CP0_ENTRYLO0
20  nop
21  tlbwi
22  NOFOUND:
23
24  mtc0 k1,CP0_ENTRYHI
25
26  j ra
27  nop
28  END(tlb_out)
```

lib/env.c

```
/* Notes written by Qian Liu <qianlxc@outlook.com>
  If you find any bug, please contact with me.*/

#include <mmu.h>
#include <error.h>
#include <env.h>
#include <kerelf.h>
#include <sched.h>
#include <pmap.h>
#include <printf.h>

struct Env *envs = NULL;        // All environments
struct Env *curenv = NULL;          // the current env

static struct Env_list env_free_list;   // Free list

extern Pde *boot_pgdir;
extern char *KERNEL_SP;


/* Overview:
 *  This function is for making an unique ID for every env.
 *
 * Pre-Condition:
 *  Env e is exist.
 *
 * Post-Condition:
 *  return e's envid on success.
 */

u_int mkenvid(struct Env *e)
{
    static u_long next_env_id = 0;

    /*Hint: lower bits of envid hold e's position in the envs array. */
    u_int idx = e - envs;

    /*Hint:  high bits of envid hold an increasing number. */
    return (++next_env_id << (1 + LOG2NENV)) | idx;
}

/* Overview:
 *  Converts an envid to an env pointer.
 *  If envid is 0 , set *penv = curenv;otherwise set *penv = envs[ENVX(envid)];
 *
 * Pre-Condition:
 *  Env penv is exist,checkperm is 0 or 1.
 *
 * Post-Condition:
 *  return 0 on success,and sets *penv to the environment.
 *  return -E_BAD_ENV on error,and sets *penv to NULL.
 */
```

```
int envid2env(u_int envid, struct Env **penv, int checkperm)
{
    struct Env *e;

    /* Hint:
     *  If envid is zero, return the current environment.*/
    if (envid == 0) {
        *penv = curenv;
        return 0;
    }

    e = &envs[ENVX(envid)];

    if (e->env_status == ENV_FREE || e->env_id != envid) {
        *penv = 0;
        return -E_BAD_ENV;
    }

    /* Hint:
     *  Check that the calling environment has legitimate permissions
     *  to manipulate the specified environment.
     *  If checkperm is set, the specified environment
     *  must be either the current environment.
     *  or an immediate child of the current environment.ok */

    if (checkperm && e != curenv && e->env_parent_id != curenv->env_id){
        *penv = 0;
        return -E_BAD_ENV;
    }
    *penv = e;
    return 0;
}

/* Overview:
 *  Mark all environments in 'envs' as free and insert them into the env_free_list.
 *  Insert in reverse order,so that the first call to env_alloc() return envs[0].
 *
 * Hints:
 *  You may use these defines to make it:
 *      LIST_INIT,LIST_INSERT_HEAD
 */
void
env_init(void)
{
    int i;
    /*Step 1: Initial env_free_list. */
    LIST_INIT(&env_free_list);

    /*Step 2: Travel the elements in 'envs', init every element(mainly initial its
status, mark it as free)
     * and inserts them into the env_free_list as reverse order. */
    for(i=NENV-1;i>=0;i--){
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list,&envs[i],env_link);
    }

}
```

```
/* Overview:
 *  Initialize the kernel virtual memory layout for environment e.
 *  Allocate a page directory, set e->env_pgdir and e->env_cr3 accordingly,
 *  and initialize the kernel portion of the new environment's address space.
 *  Do NOT map anything into the user portion of the environment's virtual address
space.
 */
/***Your Question Here***/

static int
env_setup_vm(struct Env *e)
{

    int i, r;
    struct Page *p = NULL;
    Pde *pgdir;

    /*Step 1: Allocate a page for the page directory and add its reference.
     *pgdir is the page directory of Env e. */
    if ((r = page_alloc(&p)) < 0) {
        panic("env_setup_vm - page_alloc error\n");
        return r;
    }
    p->pp_ref++;
    pgdir = (Pde *)page2kva(p);


    /*Step 2: Zero pgdir's field before UTOP. */
    for (i = 0; i < PDX(UTOP); i++) {
        pgdir[i] = 0;
    }


    /*Step 3: Copy kernel's boot_pgdir to pgdir. */

    /* Hint:
     *  The VA space of all envs is identical above UTOP
     *  (except at VPT and UVPT, which we've set below).
     *  See ./include/mmu.h for layout.
     *  Can you use boot_pgdir as a template?
     */
    for (i = PDX(UTOP); i <= PDX(~0); i++) {
        pgdir[i] = boot_pgdir[i];
    }
    e->env_pgdir = pgdir;
    e->env_cr3   = PADDR(pgdir);


    /*Step 4: VPT and UVPT map the env's own page table, with
     *different permissions. */
    e->env_pgdir[PDX(VPT)]   = e->env_cr3;
    e->env_pgdir[PDX(UVPT)]  = e->env_cr3 | PTE_V | PTE_R;
    return 0;
}
```

```
/* Overview:
 *  Allocates and Initializes a new environment.
 *  On success, the new environment is stored in *new.
 *
 * Pre-Condition:
 *  If the new Env doesn't have parent, parent_id should be zero.
 *  env_init has been called before this function.
 *
 * Post-Condition:
 *  return 0 on success, and set appropriate values for Env new.
 *  return -E_NO_FREE_ENV on error, if no free env.
 *
 * Hints:
 *  You may use these functions and defines:
 *      LIST_FIRST,LIST_REMOVE,mkenvid (Not All)
 *  You should set some states of Env:
 *      id , status , the sp register, CPU status , parent_id
 *      (the value of PC should NOT be set in env_alloc)
 */

int
env_alloc(struct Env **new, u_int parent_id)
{
    int r;
    struct Env *e;

    /*Step 1: Get a new Env from env_free_list*/
    if((e=LIST_FIRST(&env_free_list))==NULL){
        printf("Sorry,alloc env failed!\n");
        return -E_NO_FREE_ENV;
    }


    /*Step 2: Call certain function(has been implemented) to init kernel memory
layout for this new Env.
     *The function mainly maps the kernel address to this new Env address. */
    env_setup_vm(e);


    /*Step 3: Initialize every field of new Env with appropriate values*/
    e->env_parent_id = parent_id;
    e->env_status = ENV_RUNNABLE;
    e->env_runs = 0;
    e->env_id = mkenvid(e);

    /*Step 4: focus on initializing env_tf structure, located at this new Env.
     * especially the sp register,CPU status. */
    e->env_tf.cp0_status = 0x10001004;
    e->env_tf.regs[29] = USTACKTOP;

    /*Step 5: Remove the new Env from Env free list*/
    *new = e;
    LIST_REMOVE(e,env_link);
    return 0;
}
```

```
/* Overview:
 *   This is a call back function for kernel's elf loader.
 * Elf loader extracts each segment of the given binary image.
 * Then the loader calls this function to map each segment
 * at correct virtual address.
 *
 *   `bin_size` is the size of `bin`. `sgsize` is the
 * segment size in memory.
 *
 * Pre-Condition:
 *   bin can't be NULL.
 *   Hint: va may NOT aligned 4KB.
 *
 * Post-Condition:
 *   return 0 on success, otherwise < 0.
 */
static int load_icode_mapper(u_long va, u_int32_t sgsize,
                             u_char *bin, u_int32_t bin_size, void *user_data)
{
    struct Env *env = (struct Env *)user_data;
    struct Page *p = NULL;
    u_long i;
    int r;
    u_long offset = va - ROUNDDOWN(va, BY2PG);
    //printf("the va:%x,the bin_size:%d,the sgsize:%d\n",va,bin_size,sgsize);
    /*Step 1: load all content of bin into memory. */
    for (i = 0; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a page and increase the reference count of it. */
        if(page_alloc(&p)<0){
            printf("Sorry,alloc page failed!\n");
            return -E_NO_MEM;
        }
        p->pp_ref++;
        if(i==0)
            bcopy(bin,(char *)page2kva(p)+offset,((BY2PG-offset)<bin_size-
i)?(BY2PG-offset):(bin_size - i));
        else
            bcopy(bin+i-offset,(char *)page2kva(p),(BY2PG<bin_size-
i)?BY2PG:(bin_size-i));
        r = page_insert(env->env_pgdir,p,va+i,PTE_V|PTE_R);
        if(r<0){
            printf("Sorry,insert a page is failed!\n");
            return -E_NO_MEM;
        }

    }
    /*Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`.
    * i has the value of `bin_size` now. */
    //i = ROUND(bin_size,BY2PG);
    while (i < sgsize) {
        if(page_alloc(&p)<0){
            printf("Sorry,alloc page failed!\n");
            return -E_NO_MEM;
        }
        p->pp_ref++;
        r = page_insert(env->env_pgdir,p,va+i,PTE_V|PTE_R);
        if(r<0){
```

```
            printf("Sorry,alloc page failed!\n");
            return -E_NO_MEM;
        }
        //bzero(page2kva(p)+offset,BY2PG);
        i+=BY2PG;
    }
    return 0;
}


/* Overview:
 *  Sets up the the initial stack and program binary for a user process.
 *  This function loads the complete binary image by using elf loader,
 *  into the environment's user memory. The entry point of the binary image
 *  is given by the elf loader. And this function maps one page for the
 *  program's initial stack at virtual address USTACKTOP - BY2PG.
 *
 * Hints:
 *  All mappings are read/write including those of the text segment.
 *  You may use these :
 *      page_alloc, page_insert, page2kva , e->env_pgdir and load_elf.
 */
static void
load_icode(struct Env *e, u_char *binary, u_int size)
{
    /* Hint:
     *  You must figure out which permissions you'll need
     *  for the different mappings you create.
     *  Remember that the binary image is an a.out format image,
     *  which contains both text and data.
     */
    struct Page *p = NULL;
    u_long entry_point;
    u_long r;
    u_long perm;

    /*Step 1: alloc a page. */
    if(page_alloc(&p)<0){
        printf("Sorry,alloc page failed!\n");
        return;
    }

    /*Step 2: Use appropriate perm to set initial stack for new Env. */
    /*Hint: The user-stack should be writable? */
    perm = PTE_V|PTE_R;
    page_insert(e->env_pgdir,p,USTACKTOP-BY2PG,perm);
    /*Step 3:load the binary by using elf loader. */
    r = load_elf(binary,size,&entry_point,e,load_icode_mapper);
    if(r<0){
        printf("Sorry.load entire image failed!\n");
        return;
    }

    /***Your Question Here***/
    /*Step 4:Set CPU's PC register as appropriate value. */
    e->env_tf.pc = entry_point;
}
```

```
/* Overview:
 *  Allocates a new env with env_alloc, loads te named elf binary into
 *  it with load_icode. This function is ONLY called during kernel
 *  initialization, before running the first user_mode environment.
 *
 * Hints:
 *  this function wrap the env_alloc and load_icode function.
 */

void
env_create(u_char *binary, int size)
{
    struct Env *e;
    /*Step 1: Use env_alloc to alloc a new env. */
    if(env_alloc(&e,0)<0){
        printf("Sorry,env can't create because alloc env failed!\n");
        return;
    }
    /*Step 2: Use load_icode() to load the named elf binary. */
    load_icode(e,binary,size);
}



/* Overview:
 *  Frees env e and all memory it uses.
 */

void
env_free(struct Env *e)
{
    Pte *pt;
    u_int pdeno, pteno, pa;

    /* Hint: Note the environment's demise.*/
    printf("[%08x] free env %08x\n", curenv ? curenv->env_id : 0, e->env_id);

    /* Hint: Flush all mapped pages in the user portion of the address space */
    for (pdeno = 0; pdeno < PDX(UTOP); pdeno++) {
        /* Hint: only look at mapped page tables. */
        if (!(e->env_pgdir[pdeno] & PTE_V)) {
            continue;
        }
        /* Hint: find the pa and va of the page table. */
        pa = PTE_ADDR(e->env_pgdir[pdeno]);
        pt = (Pte *)KADDR(pa);
        /* Hint: Unmap all PTEs in this page table. */
        for (pteno = 0; pteno <= PTX(~0); pteno++)
            if (pt[pteno] & PTE_V) {
                page_remove(e->env_pgdir, (pdeno << PDSHIFT) | (pteno <<
PGSHIFT));
            }
        /* Hint: free the page table itself. */
        e->env_pgdir[pdeno] = 0;
        page_decref(pa2page(pa));
    }
```

```
    /* Hint: free the page directory. */
    pa = e->env_cr3;
    e->env_pgdir = 0;
    e->env_cr3 = 0;
    page_decref(pa2page(pa));
    /* Hint: return the environment to the free list. */
    e->env_status = ENV_FREE;
    LIST_INSERT_HEAD(&env_free_list, e, env_link);
}



/* Overview:
 *  Frees env e, and schedules to run a new env
 *  if e is the current env.
 */
void
env_destroy(struct Env *e)
{
    /* Hint: free e. */
    env_free(e);

    /* Hint: schedule to run a new environment. */
    if (curenv == e) {
        curenv = NULL;
        /* Hint:Why this? */
        bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
              (void *)TIMESTACK - sizeof(struct Trapframe),
              sizeof(struct Trapframe));
        printf("i am killed ... \n");
        sched_yield();
    }
}



extern void env_pop_tf(struct Trapframe *tf, int id);
extern void lcontext(u_int contxt);

/* Overview:
 *  Restores the register values in the Trapframe with the
 *  env_pop_tf, and context switch from curenv to env e.
 *
 * Post-Condition:
 *  Set 'e' as the curenv running environment.
 *
 * Hints:
 *  You may use these functions:
 *      env_pop_tf and lcontext.
 */
void
env_run(struct Env *e)
{
    /*Step 1: save register state of curenv. */
    /* Hint: if there is a environment running,you should do
    *  context switch.You can imitate env_destroy() 's behaviors.*/
```

```
    struct Trapframe *old = (struct Trapframe *)(TIMESTACK-sizeof(struct
Trapframe));
    if(curenv){
        bcopy(old,&(curenv->env_tf),sizeof(struct Trapframe));
        //curenv->env_tf.pc += 4;//aim to mips 32
        curenv->env_tf.pc = old->cp0_epc;
        //printf("cp0_epc:%x\n",curenv->env_tf.pc);
    }
    //printf("id:%d\n",e->env_id);


    /*Step 2: Set 'curenv' to the new environment. */
    curenv = e;
    curenv->env_runs ++;
    //printf("what the runs:%d\n",curenv->env_runs);


    /*Step 3: Use lcontext() to switch to its address space. */
    lcontext(KADDR(curenv->env_cr3));


    /*Step 4: Use env_pop_tf() to restore the environment's
     * environment   registers and drop into user mode in the
     * the environment.
     */
    /* Hint: You should use GET_ENV_ASID there.Think why? */
    //printf("检测 current env id:%d\n",curenv->env_id);
    //printf("检测 2 in env_run\n");
    env_pop_tf(&(curenv->env_tf),GET_ENV_ASID(curenv->env_id));
    //printf("检测,current env:%d\n",curenv->env_id);
}
```

lib/kernel_elfloader.c

```
/* This is a simplefied ELF loader for kernel.
 * You can contact me if you find any bugs.
 *
 * Luming Wang<wlm199558@126.com>
 */

#include <kerelf.h>
#include <types.h>
#include <pmap.h>

/* Overview:
 *   Check whether it is a ELF file.
 *
 * Pre-Condition:
 *   binary must longer than 4 byte.
 *
 * Post-Condition:
 *   Return 0 if `binary` isn't an elf. Otherwise
 * return 1.
 */
int is_elf_format(u_char *binary)
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;

    if (ehdr->e_ident[0] == EI_MAG0 &&
        ehdr->e_ident[1] == EI_MAG1 &&
        ehdr->e_ident[2] == EI_MAG2 &&
        ehdr->e_ident[3] == EI_MAG3) {
        return 0;
    }

    return 1;
}

/* Overview:
 *   load an elf format binary file. Map all section
 * at correct virtual address.
 *
 * Pre-Condition:
 *   `binary` can't be NULL and `size` is the size of binary.
 *
 * Post-Condition:
 *   Return 0 if success. Otherwise return < 0.
 *   If success, the entry point of `binary` will be stored in `start`
 */
int load_elf(u_char *binary, int size, u_long *entry_point, void *user_data,
             int (*map)(u_long va, u_int32_t sgsize,
                        u_char *bin, u_int32_t bin_size, void *user_data))
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
    Elf32_Phdr *phdr = NULL;
    /* As a loader, we just care about segment,
     * so we just parse program headers.
     */
```

```
    u_char *ptr_ph_table = NULL;
    Elf32_Half ph_entry_count;
    Elf32_Half ph_entry_size;
    int r;

    // check whether `binary` is a ELF file.
    if (size < 4 || !is_elf_format(binary)) {
        return -1;
    }

    ptr_ph_table = binary + ehdr->e_phoff;
    ph_entry_count = ehdr->e_phnum;
    ph_entry_size = ehdr->e_phentsize;

    while (ph_entry_count--) {
        phdr = (Elf32_Phdr *)ptr_ph_table;

        if (phdr->p_type == PT_LOAD) {
            r = map(phdr->p_vaddr, phdr->p_memsz,
                    binary + phdr->p_offset, phdr->p_filesz, user_data);

            if (r < 0) {
                return r;
            }
        }

        ptr_ph_table += ph_entry_size;
    }

    *entry_point = ehdr->e_entry;
    return 0;
}
```

boot/start.S

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>

.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
       .set   noat
       .set   noreorder
       /*
        * Register saving is delayed as long as we don't know
        * which registers really need to be saved.
        */
1:
       mfc0   k1,CP0_CAUSE
       la  k0,exception_handlers
       /*
        * Next lines assumes that the used CPU type has max.
        * 32 different types of exceptions. We might use this
        * to implement software exceptions in the future.
        */

       andi   k1,0x7c
       addu   k0,k1
       lw  k0,(k0)
       NOP
       jr  k0
       nop
END(except_vec3)
       .set   at


.data
          .globl mCONTEXT
mCONTEXT:
          .word 0

          .globl delay
delay:
          .word 0

          .globl tlbra
tlbra:
          .word 0

          .section .data.stk
KERNEL_STACK:
          .space 0x8000
.text
LEAF(_start)

   .set   mips2
   .set   reorder

   /* Disable interrupts */
   mtc0   zero, CP0_STATUS
```

```
        /* Disable watch exception. */
        mtc0    zero, CP0_WATCHLO
        mtc0    zero, CP0_WATCHHI

    /* disable kernel mode cache */
    mfc0   t0, CP0_CONFIG
    and t0, ~0x7
    ori t0, 0x2
    mtc0   t0, CP0_CONFIG

    /* set up stack */
    li  sp, 0x80400000

    li     t0,0x80400000
    sw     t0,mCONTEXT

    /* jump to main */
    jal main

loop:
    j   loop
    nop
END(_start)
```

lib/traps.c

```
#include <trap.h>
#include <env.h>
#include <printf.h>

extern void handle_int();
extern void handle_reserved();
extern void handle_tlb();
extern void handle_sys();
extern void handle_mod();
unsigned long exception_handlers[32];
void trap_init(){
    int i;
    for(i=0;i<32;i++)
    set_except_vector(i, handle_reserved);
    set_except_vector(0, handle_int);
    set_except_vector(1, handle_mod);
    set_except_vector(2, handle_tlb);
    set_except_vector(3, handle_tlb);
    set_except_vector(8, handle_sys);
}
void *set_except_vector(int n, void * addr){
    unsigned long handler=(unsigned long)addr;
    unsigned long old_handler=exception_handlers[n];
    exception_handlers[n]=handler;
    return (void *)old_handler;
}


struct pgfault_trap_frame{
        u_int fault_va;
        u_int err;
        u_int sp;
        u_int eflags;
        u_int pc;
        u_int empty1;
        u_int empty2;
        u_int empty3;
        u_int empty4;
        u_int empty5;
};
```

```
void
page_fault_handler(struct Trapframe *tf)
{
        u_int va;
        u_int *tos, d;
    struct Trapframe PgTrapFrame;
    extern struct Env * curenv;
//printf("^^^^cp0_BadVAddress:%x\n",tf->cp0_badvaddr);


    bcopy(tf, &PgTrapFrame,sizeof(struct Trapframe));
    if(tf->regs[29] >= (curenv->env_xstacktop - BY2PG) && tf->regs[29] <=
(curenv->env_xstacktop - 1))
    {
        //panic("fork can't nest!!");
        tf->regs[29] = tf->regs[29] - sizeof(struct  Trapframe);
        bcopy(&PgTrapFrame, tf->regs[29], sizeof(struct Trapframe));
    }
    else
    {

        tf->regs[29] = curenv->env_xstacktop - sizeof(struct  Trapframe);
//      printf("page_fault_handler(): bcopy():
src:%x\tdes:%x\n",(int)&PgTrapFrame,(int)(curenv->env_xstacktop -
sizeof(struct  Trapframe)));

        bcopy(&PgTrapFrame, curenv->env_xstacktop - sizeof(struct  Trapframe),
sizeof(struct Trapframe));
    }
//  printf("^^^^cp0_epc:%x\tcurenv->env_pgfault_handler:%x\n",tf->cp0_epc,cur
env->env_pgfault_handler);

    tf->cp0_epc = curenv->env_pgfault_handler;


    return;
}
```

lib/sched.c

```
#include <env.h>
#include <pmap.h>
#include <printf.h>

/* Overview:
 *  Implement simple round-robin scheduling.
 *  Search through 'envs' for a runnable environment ,
 *  in circular fashion statrting after the previously running env,
 *  and switch to the first such environment found.
 *
 * Hints:
 *  The variable which is for counting should be defined as 'static'.
 */

void sched_yield(void)
{
    static u_long count = 0;
    while (1){
        count = (count+1)%NENV;
        if (envs[count].env_status==ENV_RUNNABLE) {
            env_run((envs+count));
        }
    }

}
```

user/syscall_wrap.S

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>

/* Overview:
 *  `msyscall` push all the arguments into the stack, running
 * `syscall` instruction.
 *
 * Pre-Condition:
 *  The first, second, third and fourth arguments are passed
 * by registers(a0~a3). The remains are stored on the stack.
 *
 * Post-Condition:
 *  All arguments should be stored on the stack. Syscall number
 * should be passed by register v0.
 *
 * Hint:
 *  Interestingly, MIPS 32 ABI(application binary interface) defined that
 * allocating space, which shoud be large enough to contain all the arguments,
 * on the stack is always required.
 *  So, we needn't allocate space on the stack again. In another word,
 * we shouldn't change the value of $sp. All we need to do is store
 * registers(a0~a3) on the stack.
 *  Remember passing syscall number by register v0 :)
 */


LEAF(msyscall)

sw  a0,0(sp)
sw  a1,4(sp)
sw  a2,8(sp)
sw  a3,12(sp)
move    v0, a0

syscall

jr  ra

END(msyscall)

LEAF(getDate)
    lw t0,0x95000000
    lw v0,0x95000010
    jr  ra
    nop
END(getDate)

LEAF(exitShell)
    lw t0,0x90000010
    jr  ra
    nop
END(exitShell)
```

lib/syscall.S

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>
#include <stackframe.h>
#include <unistd.h>


NESTED(handle_sys,TF_SIZE, sp)

SAVE_ALL
CLI

//1: j 1b
nop
.set at
lw t1, TF_EPC(sp)
lw v0, TF_REG2(sp)

subu v0, v0, __SYSCALL_BASE
sltiu t0, v0, __NR_SYSCALLS+1

addiu t1, 4
sw  t1, TF_EPC(sp)
beqz    t0,  illegal_syscall//undef
nop
sll t0, v0,2
la  t1, sys_call_table
addu    t1, t0
lw  t2, (t1)
beqz    t2, illegal_syscall//undef
nop
lw  t0,TF_REG29(sp)

lw  t1, (t0)
lw  t3, 4(t0)
lw  t4, 8(t0)
lw  t5, 12(t0)
lw  t6, 16(t0)
lw  t7, 20(t0)

subu    sp, 20

sw  t1, 0(sp)
sw  t3, 4(sp)
sw  t4, 8(sp)
sw  t5, 12(sp)
sw  t6, 16(sp)
sw  t7, 20(sp)

move    a0, t1
move    a1, t3
move    a2, t4
move    a3, t5

jalr    t2
nop
```

```
addu    sp, 20

sw  v0, TF_REG2(sp)

j   ret_from_exception//extern?
nop

illegal_syscall: j illegal_syscall
           nop
END(handle_sys)

    .extern sys_putchar
    .extern sys_getenvid
    .extern sys_yield
    .extern sys_env_destroy
    .extern sys_set_pgfault_handler
    .extern sys_mem_alloc
    .extern sys_mem_map
    .extern sys_mem_unmap
    .extern sys_env_alloc
    .extern sys_set_env_status
    .extern sys_set_trapframe
    .extern sys_panic
    .extern sys_ipc_can_send
    .extern sys_ipc_recv
    .extern sys_cgetc

.macro syscalltable
.word sys_putchar
.word sys_getenvid
.word sys_yield
.word sys_env_destroy
.word sys_set_pgfault_handler
.word sys_mem_alloc
.word sys_mem_map
.word sys_mem_unmap
.word sys_env_alloc
.word sys_set_env_status
.word sys_set_trapframe
.word sys_panic
.word sys_ipc_can_send
.word sys_ipc_recv
.word sys_cgetc
.endm


EXPORT(sys_call_table)
syscalltable
.size sys_call_table, . - sys_call_table
```

lib/syscall_all.c

```
#include "../drivers/gxconsole/dev_cons.h"
#include <mmu.h>
#include <env.h>
#include <printf.h>
#include <pmap.h>
#include <sched.h>

extern char *KERNEL_SP;
extern struct Env *curenv;

/* Overview:
 *  This function is used to print a character on screen.
 *
 * Pre-Condition:
 *  `c` is the character you want to print.
 */
void sys_putchar(int sysno, int c, int a2, int a3, int a4, int a5)
{
    printcharc((char) c);
    return ;
}




/* Overview:
 *  This function enables you to copy content of `srcaddr` to `destaddr`.
 *
 * Pre-Condition:
 *  `destaddr` and `srcaddr` can't be NULL. Also, the `srcaddr` area
 *  shouldn't overlap the `destaddr`, otherwise the behavior of this
 *  function is undefined.
 *
 * Post-Condition:
 *  the content of `destaddr` area(from `destaddr` to `destaddr`+`len`) will
 * be same as that of `srcaddr` area.
 */


void *memcpy(void *destaddr, void const *srcaddr, u_int len)
{
    char *dest = destaddr;
    char const *src = srcaddr;

    while (len-- > 0) {
        *dest++ = *src++;
    }

    return destaddr;
}
```

```
/* Overview:
 *  This function provides the environment id of current process.
 *
 * Post-Condition:
 *  return the current environment id
 */
u_int sys_getenvid(void)
{
    return curenv->env_id;
}

/* Overview:
 *  This function enables the current process to give up CPU.
 *
 * Post-Condition:
 *  Deschedule current environment. This function will never return.
 */
void sys_yield(void)
{
    //类似于 env_destroy，保存 kernel_sp 中的 Trapframe，随后执行 sched_yield;
    bcopy((void*)(KERNEL_SP-sizeof(struct Trapframe)),(void*)TIMESTACK-
sizeof(struct Trapframe),sizeof(struct Trapframe));
    sched_yield();
}



/* Overview:
 *  This function is used to destroy the current environment.
 *
 * Pre-Condition:
 *  The parameter `envid` must be the environment id of a
 * process, which is either a child of the caller of this function
 * or the caller itself.
 *
 * Post-Condition:
 *  Return 0 on success, < 0 when error occurs.
 */

int sys_env_destroy(int sysno, u_int envid)
{
    /*
        printf("[%08x] exiting gracefully\n", curenv->env_id);
        env_destroy(curenv);
    */
    int r;
    struct Env *e;

    if ((r = envid2env(envid, &e, 1)) < 0) {
        return r;
    }

    printf("[%08x] destroying %08x\n", curenv->env_id, e->env_id);
    env_destroy(e);
    return 0;
}
```

```
/* Overview:
 *  Set envid's pagefault handler entry point and exception stack.
 *
 * Pre-Condition:
 *  xstacktop points one byte past exception stack.
 *
 * Post-Condition:
 *  The envid's pagefault handler will be set to `func` and its
 *  exception stack will be set to `xstacktop`.
 *  Returns 0 on success, < 0 on error.
 */
int sys_set_pgfault_handler(int sysno, u_int envid, u_int func, u_int
xstacktop)
{
    // Your code here.
    struct Env *env;
    int ret = 0;
    if ((ret = envid2env(envid,&env,0))!=0) return ret;
    env->env_xstacktop = xstacktop;
    env->env_pgfault_handler = func;
    return 0;
    //  panic("sys_set_pgfault_handler not implemented");
}


/* Overview:
 *  Allocate a page of memory and map it at 'va' with permission
 * 'perm' in the address space of 'envid'.
 *
 *  If a page is already mapped at 'va', that page is unmapped as a
 * side-effect.
 *
 * Pre-Condition:
 * perm -- PTE_V is required,
 *         PTE_COW is not allowed(return -E_INVAL),
 *         other bits are optional.
 *
 * Post-Condition:
 * Return 0 on success, < 0 on error
 *  - va must be < UTOP
 *  - env may modify its own address space or the address space of its children
 */
int sys_mem_alloc(int sysno, u_int envid, u_int va, u_int perm)
{
    // Your code here.
    struct Env *env;
    struct Page *ppage;
    int ret;
    ret = 0;
    assert(va%BY2PG==0);
    if ((perm & PTE_COW)==PTE_COW || va>=UTOP) return -E_INVAL;
    if ((ret = envid2env(envid,&env,0))!=0) return ret;
    if ((ret = page_alloc(&ppage))!=0) return ret;
    if ((ret = page_insert(env->env_pgdir,ppage,va,perm))!=0) return ret;
    ret = 0;
    return ret;
}
```

```
/* Overview:
 *  Map the page of memory at 'srcva' in srcid's address space
 * at 'dstva' in dstid's address space with permission 'perm'.
 * Perm has the same restrictions as in sys_mem_alloc.
 * (Probably we should add a restriction that you can't go from
 * non-writable to writable?)
 *
 * Post-Condition:
 *  Return 0 on success, < 0 on error.
 *
 * Note:
 *  Cannot access pages above UTOP.
 */
int sys_mem_map(int sysno, u_int srcid, u_int srcva, u_int dstid, u_int
dstva,
                u_int perm)
{
    int ret;
    u_int round_srcva, round_dstva;
    struct Env *srcenv;
    struct Env *dstenv;
    struct Page *ppage;
    Pte *ppte;
    //your code here
    ppage = NULL;
    ret = 0;
    round_srcva = ROUNDDOWN(srcva, BY2PG);
    round_dstva = ROUNDDOWN(dstva, BY2PG);
    if ((perm & PTE_COW)!=0 || dstva>=UTOP) return -E_INVAL;
    if ((ret = envid2env(srcid,&srcenv,0))!=0) return ret;
    if ((ret = envid2env(dstid,&dstenv,0))!=0) return ret;
    ppage = pa2page(va2pa(srcenv->env_pgdir,srcva));
    //ppage = page_lookup(srcenv->env_pgdir,srcva,&ppte);//获取 srcva 映射的 page
    pgdir_walk(srcenv->env_pgdir,srcva,0,&ppte);//获取 srcva 对应的页表项
    if (ppte!=NULL && ((*ppte)&PTE_R==0) && (perm&PTE_R!=0)) return -
E_INVAL;//企图从不可写映射到可写,返回错误
    if ((ret = page_insert(dstenv->env_pgdir,ppage,dstva,perm))!=0) return
ret;
    return ret;
}



/* Overview:
 *  Unmap the page of memory at 'va' in the address space of 'envid'
 * (if no page is mapped, the function silently succeeds)
 *
 * Post-Condition:
 *  Return 0 on success, < 0 on error.
 *
 * Cannot unmap pages above UTOP.
 */
int sys_mem_unmap(int sysno, u_int envid, u_int va)
{
    // Your code here.
    int ret;
    struct Env *env;
```

```
    if (va>=UTOP) return -E_INVAL;
    if ((ret = envid2env(envid,&env,0))!=0) return ret;
    page_remove(env->env_pgdir,va);
    return ret;
    //  panic("sys_mem_unmap not implemented");
}



/* Overview:
 *  Allocate a new environment.
 *
 * Pre-Condition:
 * The new child is left as env_alloc created it, except that
 * status is set to ENV_NOT_RUNNABLE and the register set is copied
 * from the current environment.
 *
 * Post-Condition:
 *  In the child, the register set is tweaked so sys_env_alloc returns 0.
 *  Returns envid of new environment, or < 0 on error.
 */

int sys_env_alloc(int sysno,int iffork)
{
    // Your code here.
    int r;
    struct Env *e;
    Pte* ppte;
    //以 curenv->env_id 作为父进程的 id 来创建一个子进程
    bcopy((void*)KERNEL_SP-sizeof(struct
Trapframe),&(curenv->env_tf),sizeof(struct Trapframe));
    if ((r = env_alloc(&e,curenv->env_id))!=0) return r;
    bcopy(&(curenv->env_tf),&(e->env_tf),sizeof(struct Trapframe));
    if (iffork) {
    u_long i;
    for (i = UTEXT;i<UTOP-2*BY2PG;i=i+BY2PG) {
        ppte=0;
        pgdir_walk(curenv->env_pgdir,i,0,&ppte);
        if (ppte) {
            if ((*ppte & PTE_V)!=0) {
                if ((*ppte & PTE_R)!=0 || (*ppte & PTE_COW)!=0) {
                    if ((*ppte & PTE_LIBRARY)==0) {
                        if (r =
page_insert(curenv->env_pgdir,pa2page(PTE_ADDR(*ppte)),i,PTE_R|PTE_V|PTE_COW)
) return r;
                        if (r =
page_insert(e->env_pgdir,pa2page(PTE_ADDR(*ppte)),i,PTE_R|PTE_V|PTE_COW))
return r;
                    } else {
                        if (r =
page_insert(e->env_pgdir,pa2page(PTE_ADDR(*ppte)),i,PTE_R|PTE_V|PTE_LIBRARY))
return r;
                    }
                } else {
                    if (r=page_insert(e->env_pgdir,pa2page(PTE_ADDR(*ppte)),i,
PTE_V)) return r;
                }
```

```
            }
        }
    }
    }

    e->env_status = ENV_NOT_RUNNABLE;
    e->env_tf.pc = e->env_tf.cp0_epc;
    e->env_tf.regs[2] = 0;//返回值寄存器设置为 0
    //printf("sys_env_alloc return enf_id:%d\n",e->env_id);
        //panic("sys_env_alloc not implemented");
    return e->env_id;
}



/* Overview:
 *  Set envid's env_status to status.
 *
 * Pre-Condition:
 *  status should be one of `ENV_RUNNABLE`, `ENV_NOT_RUNNABLE` and
 * `ENV_FREE`. Otherwise return -E_INVAL.
 *
 * Post-Condition:
 *  Returns 0 on success, < 0 on error.
 *  Return -E_INVAL if status is not a valid status for an environment.
 *  The status of environment will be set to `status` on success.
 */

int sys_set_env_status(int sysno, u_int envid, u_int status)
{
    // Your code here.
    struct Env *env;
    int ret;
    if(ret=(envid2env(envid,&env,0))) return ret;
    if((status!=ENV_FREE)&&(status!=ENV_NOT_RUNNABLE)&&(status!=ENV_RUNNABLE)
) {
        return -E_INVAL;
    }
    env->env_status = status;
    return 0;
    //  panic("sys_env_set_status not implemented");
}



/* Overview:
 *  Set envid's trap frame to tf.
 *
 * Pre-Condition:
 *  `tf` should be valid.
 *
 * Post-Condition:
 *  Returns 0 on success, < 0 on error.
 *  Return -E_INVAL if the environment cannot be manipulated.
 *
 * Note: This hasn't be used now?
 */
```

```
int sys_set_trapframe(int sysno, u_int envid, struct Trapframe *tf)
{
    struct Env *env;
    int ret;
    if(ret=envid2env(envid,&env,0)) {
        return ret;
    }
    env->env_tf=*tf;

    return 0;
}


/* Overview:
 *  Kernel panic with message `msg`.
 *
 * Pre-Condition:
 *  msg can't be NULL
 *
 * Post-Condition:
 *  This function will make the whole system stop.
 */

void sys_panic(int sysno, char *msg)
{
    // no page_fault_mode -- we are trying to panic!
    panic("%s", TRUP(msg));
}



/* Overview:
 *  This function enables caller to receive message from
 * other process. To be more specific, it will flag
 * the current process so that other process could send
 * message to it.
 *
 * Pre-Condition:
 *  `dstva` is valid (Note: NULL is also a valid value for `dstva`).
 *
 * Post-Condition:
 *  This syscall will set the current process's status to
 * ENV_NOT_RUNNABLE, giving up cpu.
 */

void sys_ipc_recv(int sysno, u_int dstva)
{
    if (dstva>=UTOP) {
        return;
    }
    curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_recving = 1;
    curenv->env_status = ENV_NOT_RUNNABLE;
    sys_yield();
}
```

```
/* Overview:
 *  Try to send 'value' to the target env 'envid'.
 *
 *  The send fails with a return value of -E_IPC_NOT_RECV if the
 * target has not requested IPC with sys_ipc_recv.
 *  Otherwise, the send succeeds, and the target's ipc fields are
 * updated as follows:
 *    env_ipc_recving is set to 0 to block future sends
 *    env_ipc_from is set to the sending envid
 *    env_ipc_value is set to the 'value' parameter
 *  The target environment is marked runnable again.
 *
 * Post-Condition:
 *  Return 0 on success, < 0 on error.
 *
 * Hint: the only function you need to call is envid2env.
 */

int sys_ipc_can_send(int sysno, u_int envid, u_int value, u_int srcva,
                     u_int perm)
{

    int r;
    struct Env *e;
    struct Page *p;
    if ((r = envid2env(envid,&e,0))!=0) return r;
    if (e->env_ipc_recving!=1) return -E_IPC_NOT_RECV;
    e->env_ipc_recving=0;
    e->env_ipc_from=curenv->env_id;
    e->env_ipc_value=value;
    if (srcva!=0) {
        if(r=sys_mem_map(sysno,curenv->env_id,srcva,envid,e->env_ipc_dstva,per
m)) return r;
        e->env_ipc_perm = perm;
    }
    e->env_status=ENV_RUNNABLE;
    return 0;
}
```

user/fork.c

```
// implement fork from user space

#include "lib.h"
#include <mmu.h>
#include <env.h>

//because now we are in user status,so we need use user/syscall_lib.c but not
lib/syscall_all.c
//be careful.
/* ---------------- help functions ---------------- */

/* Overview:
 *  Copy `len` bytes from `src` to `dst`.
 *
 * Pre-Condition:
 *  `src` and `dst` can't be NULL. Also, the `src` area
 *   shouldn't overlap the `dest`, otherwise the behavior of this
 *   function is undefined.a;
 */

void user_bcopy(const void *src, void *dst, size_t len)
{
    void *max;
    //  writef("~~~~~~~~~~~~~~~~ src:%x dst:%x
len:%x\n",(int)src,(int)dst,len);
    max = dst + len;
    // copy machine words while possible
    if (((int)src % 4 == 0) && ((int)dst % 4 == 0)) {
        while (dst + 3 < max) {
            *(int *)dst = *(int *)src;
            dst += 4;
            src += 4;
        }
    }
    // finish remaining 0-3 bytes
    while (dst < max) {
        *(char *)dst = *(char *)src;
        dst += 1;
        src += 1;
    }
    //for(;;);
}

/* Overview:
 *  Sets the first n bytes of the block of memory
 * pointed by `v` to zero.
 *
 * Pre-Condition:
 *  `v` must be valid.
 *
 * Post-Condition:
 *  the content of the space(from `v` to `v`+ n)
 * will be set to zero.
 */
```

```
void user_bzero(void *v, u_int n)
{
    char *p;
    int m;

    p = v;
    m = n;

    while (--m >= 0) {
        *p++ = 0;
    }
}
/*--------------------------------------------------------------*/


/* Overview:
 *  Custom page fault handler - if faulting page is copy-on-write,
 * map in our own private writable copy.
 *
 * Pre-Condition:
 *  `va` is the address which leads to a TLBS exception.
 *
 * Post-Condition:
 *  Launch a user_panic if `va` is not a copy-on-write page.
 * Otherwise, this handler should map a private writable copy of
 * the faulting page at correct address.
 */

static void
pgfault(u_int va)
{
    u_int temp = 0x50000000;
    //first we must make sure that va is align to BY2PG
    va = ROUNDDOWN(va,BY2PG);
    u_int perm = (*vpt)[VPN(va)]& 0xfff;
    //writef("fork.c:pgfault():\t va:%x\n",va);
    if(perm & PTE_COW){
        if(syscall_mem_alloc(0,temp,perm &(~PTE_COW))<0){
            user_panic("syscall_mem_alloc error.\n");
        }
        user_bcopy((void *)va,(void *)temp,BY2PG);
        if(syscall_mem_map(0,temp,0,va,perm &(~PTE_COW))<0){
            user_panic("syscall_mem_map error.\n");
        }
        if(syscall_mem_unmap(0,temp)<0){
            user_panic("syscall_mem_unmap error.\n");
        }
    }else{
        user_panic("va page is not PTE_COW.\n");
    }
}
```

```
/* Overview:
 *  Map our virtual page `pn` (address pn*BY2PG) into the target `envid`
 * at the same virtual address.
 *
 * Post-Condition:
 *  if the page is writable or copy-on-write, the new mapping must be
 * created copy on write and then our mapping must be marked
 * copy on write as well. In another word, both of the new mapping and
 * our mapping should be copy-on-write if the page is writable or
 * copy-on-write.
 *
 * Hint:
 *  PTE_LIBRARY indicates that the page is shared between processes.
 * A page with PTE_LIBRARY may have PTE_R at the same time. You
 * should process it correctly.
 */
static void
duppage(u_int envid, u_int pn)
{
    /* Note:
     *  I am afraid I have some bad news for you. There is a ridiculous,
     * annoying and awful bug here. I could find another more adjectives
     * to qualify it, but you have to reproduce it to understand
     * how disturbing it is.
     *  To reproduce this bug, you should follow the steps bellow:
     *  1. uncomment the statement "writef("");" bellow.
     *  2. make clean && make
     *  3. lauch Gxemul and check the result.
     *  4. you can add serveral `writef("");` and repeat step2~3.
     *  Then, you will find that additional `writef("");` may lead to
     * a kernel panic. Interestingly, some students, who faced a strange
     * kernel panic problem, found that adding a `writef("");` could solve
     * the problem.
     *  Unfortunately, we cannot find the code which leads to this bug,
     * although we have debugged it for serveral weeks. If you face this
     * bug, we would like to say "Good luck. God bless."
     */
    // writef("");
    u_int perm;
    perm = (*vpt)[pn] & 0xfff; //取出标记位
    if((((perm & PTE_R) !=0) || ((perm & PTE_COW)!=0)) && (perm & PTE_V)){
        if(perm & PTE_LIBRARY){
            perm = PTE_V | PTE_R | PTE_LIBRARY;
        }else{
            perm = PTE_V | PTE_R | PTE_COW;
        }
        if(syscall_mem_map(0,pn*BY2PG,envid,pn*BY2PG,perm)<0){
            user_panic("syscall_mem_map for son failed.\n");
        }
        if(syscall_mem_map(0,pn*BY2PG,0,pn*BY2PG,perm)<0){
            user_panic("syscall_mem_map for father failed.\n");
        }
    }else{
        if(syscall_mem_map(0,pn*BY2PG,envid,pn*BY2PG,perm)<0){
            user_panic("syscall_mem_map for son failed.1\n");
        }
    }
```

```
    //user_panic("duppage not implemented");
}

/* Overview:
 *  User-level fork. Create a child and then copy our address space
 * and page fault handler setup to the child.
 *
 * Hint: use vpd, vpt, and duppage.
 * Hint: remember to fix "env" in the child process!
 * Note: `set_pgfault_handler`(user/pgfault.c) is different from
 *       `syscall_set_pgfault_handler`.
 */
extern void __asm_pgfault_handler(void);
int
fork(void)
{
    // Your code here.
    u_int newenvid;
    extern struct Env *envs;
    extern struct Env *env;//将其指向当前的进程，如果子进程无法创建，则指向父进程
    u_int i;
    //The parent installs pgfault using set_pgfault_handler
    set_pgfault_handler(pgfault);
    //alloc a new env
    if((newenvid = syscall_env_alloc())==0){
        //in child env
        env = &envs[ENVX(syscall_getenvid())];
        return 0;
    }
    /*use vpt vpd，我们只需要将父进程中相关的用户空间的页复制到子进程用户空间即可*/
    /*注意创建一个进程的时候会调用 env_vm_init 函数，这个函数有个非常关键的操作,我们创建
    子进程，复制父进程的地址空间只需要复制 UTOP 以下的页即可，因为所有进程 UTOP 以上的页都是
利用
    boot_pgdir 作为模板复制的，不需要再次复制拷贝*/
    /*we need judge whether the pgtable is exist or the page is exist.*/
    for(i=0;i<UTOP-BY2PG;i+=BY2PG){
        if(((*vpd)[VPN(i)/1024])!=0 && ((*vpt)[VPN(i)])!=0){
            duppage(newenvid,VPN(i));
        }
    }
    //搭建异常处理栈，分配一个页，让别的进程不抢占此页
    if(syscall_mem_alloc(newenvid,UXSTACKTOP-BY2PG,PTE_V|PTE_R)<0){
        user_panic("failed alloc UXSTACK.\n");
        return 0;
    }
    //帮助子进程注册错误处理函数
    if(syscall_set_pgfault_handler(newenvid,__asm_pgfault_handler,UXSTACKTOP)
<0){
        user_panic("page fault handler setup failed.\n");
        return 0;
    }
    //we need to set the child env status to ENV_RUNNABLE,we must use
syscall_set_env_status.
    syscall_set_env_status(newenvid,ENV_RUNNABLE);
    writef("OK! newenvid is:%d\n",newenvid);
    return newenvid;
```

```
}
// Challenge!
int
sfork(void)
{
    user_panic("sfork not implemented");
    return -E_INVAL;
}
```

fs/ide.c

```
/*
 * Minimal PIO-based (non-interrupt-driven) IDE driver code.
 * For information about what all this IDE/ATA magic means,
 * see for example "The Guide to ATA/ATAPI documentation" at:
 *  http://www.stanford.edu/~csapuntz/ide.html
 */

#include "fs.h"
#include "lib.h"
#include <mmu.h>

void
ide_read(u_int diskno, u_int secno, void *dst, u_int nsecs)
{
    int offset_begin = secno * 0x200;
    int offset_end = offset_begin + nsecs * 0x200;
    int offset = 0;

    while(offset_begin + offset < offset_end)
    {
        //writef("ide.c: read_sector() offset=%x\n",offset_begin + offset);
        if (read_sector(offset_begin + offset))
        {
            user_bcopy( 0x93004000,dst + offset, 0x200);
            offset += 0x200;
            //user_panic("$$$$$$$$$$$$$$$$$$$$$$");

        } else {
            user_panic("disk I/O error");
        }
    }
}

void
ide_write(u_int diskno, u_int secno, void *src, u_int nsecs)
{
    int offset_begin = secno * 0x200;
    int offset_end = offset_begin + nsecs * 0x200;
    int offset = 0;

    while (offset_begin + offset < offset_end)
    {
        //writef("ide_write(): offset_begin:%x offset:%x
src:%x\n",offset_begin,offset,src);
        user_bcopy(src + offset,0x93004000,  0x200);
        if (write_sector(offset_begin + offset))
        {
            offset += 0x200;
        } else {
            user_panic("disk I/O error");
        }
    }
}
```

fs/ide_asm.S

```
#include <asm/regdef.h>
#include <asm/cp0regdef.h>
#include <asm/asm.h>

LEAF(read_sector)


    sw  a0, 0x93000000
    li  t0, 0
    sw  t0 , 0x93000010
    li  t0, 0
    sb  t0 , 0x93000020
    lw      v0, 0x93000030
//1: j 1b
nop
    jr  ra
    nop
END(read_sector)

LEAF(write_sector)
    sw  a0, 0x93000000
    li  t0, 0
    sw  t0 , 0x93000010
    li  t0, 1
    sb  t0 , 0x93000020
    lw      v0, 0x93000030
    jr  ra
    nop
END(write_sector)
```

user/pipe.c

```
#include "lib.h"
#include <mmu.h>
#include <env.h>
#define debug 0

static int pipeclose(struct Fd*);
static int piperead(struct Fd *fd, void *buf, u_int n, u_int offset);
static int pipestat(struct Fd*, struct Stat*);
static int pipewrite(struct Fd *fd, const void *buf, u_int n, u_int offset);

struct Dev devpipe =
{
.dev_id=    'p',
.dev_name=  "pipe",
.dev_read=  piperead,
.dev_write= pipewrite,
.dev_close= pipeclose,
.dev_stat=  pipestat,
};

#define BY2PIPE 32      // small to provoke races

struct Pipe {
    u_int p_rpos;       // read position
    u_int p_wpos;       // write position
    u_char p_buf[BY2PIPE];  // data buffer
};

int
pipe(int pfd[2])
{
    int r, va;
    struct Fd *fd0, *fd1;

    // allocate the file descriptor table entries
    if ((r = fd_alloc(&fd0)) < 0
    ||  (r = syscall_mem_alloc(0, (u_int)fd0, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        goto err;

    if ((r = fd_alloc(&fd1)) < 0
    ||  (r = syscall_mem_alloc(0, (u_int)fd1, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        goto err1;

    // allocate the pipe structure as first data page in both
    va = fd2data(fd0);
    if ((r = syscall_mem_alloc(0, va, PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        goto err2;
    if ((r = syscall_mem_map(0, va, 0, fd2data(fd1),
PTE_V|PTE_R|PTE_LIBRARY)) < 0)
        goto err3;

    // set up fd structures
    fd0->fd_dev_id = devpipe.dev_id;
    fd0->fd_omode = O_RDONLY;
```

```
    fd1->fd_dev_id = devpipe.dev_id;
    fd1->fd_omode = O_WRONLY;

    writef("[%08x] pipecreate \n", env->env_id, (* vpt)[VPN(va)]);

    pfd[0] = fd2num(fd0);
    pfd[1] = fd2num(fd1);
    return 0;

err3:   syscall_mem_unmap(0, va);
err2:   syscall_mem_unmap(0, (u_int)fd1);
err1:   syscall_mem_unmap(0, (u_int)fd0);
err:    return r;
}

static int
_pipeisclosed(struct Fd *fd, struct Pipe *p)
{
    // Your code here.
    //
    // Check pageref(fd) and pageref(p),
    // returning 1 if they're the same, 0 otherwise.
    //
    // The logic here is that pageref(p) is the total
    // number of readers *and* writers, whereas pageref(fd)
    // is the number of file descriptors like fd (readers if fd is
    // a reader, writers if fd is a writer).
    //
    // If the number of file descriptors like fd is equal
    // to the total number of readers and writers, then
    // everybody left is what fd is.  So the other end of
    // the pipe is closed.
    int pfd,pfp,runs;
    do {
        runs = env->env_runs;
        pfd = pageref(fd);
        pfp = pageref(p);
    } while (runs!=(env->env_runs));
    return (pfd==pfp? 1:0);
//  panic("_pipeisclosed not implemented");
//  return 0;
}

int
pipeisclosed(int fdnum)
{
    struct Fd *fd;
    struct Pipe *p;
    int r;

    if ((r = fd_lookup(fdnum, &fd)) < 0)
        return r;
    p = (struct Pipe*)fd2data(fd);
    return _pipeisclosed(fd, p);
}
```

```
static int
piperead(struct Fd *fd, void *vbuf, u_int n, u_int offset)
{
    // Your code here.  See the lab text for a description of
    // what piperead needs to do.  Write a loop that
    // transfers one byte at a time.  If you decide you need
    // to yield (because the pipe is empty), only yield if
    // you have not yet copied any bytes.  (If you have copied
    // some bytes, return what you have instead of yielding.)
    // If the pipe is empty and closed and you didn't copy any data out, return 0.
    // Use _pipeisclosed to check whether the pipe is closed.
    int i;
    struct Pipe *p;
    char *rbuf = vbuf;

    p = (struct Pipe*)fd2data(fd);
    if (_pipeisclosed(fd,p)) return 0;
    for (i=0;i<n;i++) {
        while ( (p->p_rpos) >= (p->p_wpos) )  {
            if (_pipeisclosed(fd,p))
                return i;
            syscall_yield();
        }
        *rbuf = p->p_buf[(p->p_rpos)%BY2PIPE];
        rbuf++;
        (p->p_rpos)++;
    }
    return i;
//  panic("piperead not implemented");
//  return -E_INVAL;
}



static int
pipewrite(struct Fd *fd, const void *vbuf, u_int n, u_int offset)
{
    // Your code here.  See the lab text for a description of what
    // pipewrite needs to do.  Write a loop that transfers one byte
    // at a time.  Unlike in read, it is not okay to write only some
    // of the data.  If the pipe fills and you've only copied some of
    // the data, wait for the pipe to empty and then keep copying.
    // If the pipe is full and closed, return 0.
    // Use _pipeisclosed to check whether the pipe is closed.
    int i;
    struct Pipe *p;
    char *wbuf = vbuf;
    p = (struct Pipe*)fd2data(fd);
    if (_pipeisclosed(fd,p)) return 0;
    for (i=0;i<n;i++) {
        while ( (p->p_wpos - p->p_rpos) >= BY2PIPE )  {
            if (_pipeisclosed(fd,p))
                return i;
            syscall_yield();
        }

        p->p_buf[(p->p_wpos)%BY2PIPE] = (*wbuf);
```

```
        wbuf++;
        (p->p_wpos)++;
    }
    return n;
}



static int
pipestat(struct Fd *fd, struct Stat *stat)
{
    struct Pipe *p;
    p = (struct Pipe*)fd2data(fd);
    p = (struct Pipe *)fd2data(fd);
    strcpy(stat->st_name, "<pipe>");
    stat->st_size = p->p_wpos - p->p_rpos;
    stat->st_isdir = 0;
    stat->st_dev = &devpipe;
    return 0;
}

static int
pipeclose(struct Fd *fd)
{
    syscall_mem_unmap(0, fd2data(fd));
    return 0;
}
```