```c
#include <asm/cp0regdef.h>
#include <elf.h>
#include <env.h>
#include <mmu.h>
#include <pmap.h>
#include <printk.h>
#include <sched.h>

// The maximum number of available ASIDs.
// Our bitmap requires this to be a multiple of 32.
#define NASID 64

struct Env envs[NENV] __attribute__((aligned(BY2PG))); // All environments

struct Env *curenv = NULL;        // the current env
static struct Env_list env_free_list; // Free list

// Invariant: 'env' in 'env_sched_list' iff. 'env->env_status' is 'RUNNABLE'.
struct Env_sched_list env_sched_list; // Runnable list

static Pde *base_pgdir;

static uint32_t asid_bitmap[NASID / 32] = {0}; // 64

/* Overview:
 *  Allocate an unused ASID.
 *
 * Post-Condition:
 *   return 0 and set '*asid' to the allocated ASID on success.
 *   return -E_NO_FREE_ENV if no ASID is available.
 */
static int asid_alloc(u_int *asid) {
    for (u_int i = 0; i < NASID; ++i) {
        int index = i >> 5;
        int inner = i & 31;
        if ((asid_bitmap[index] & (1 << inner)) == 0) {
            asid_bitmap[index] |= 1 << inner;
            *asid = i;
            return 0;
        }
    }
    return -E_NO_FREE_ENV;
}

/* Overview:
 *  Free an ASID.
 *
 * Pre-Condition:
 *  The ASID is allocated by 'asid_alloc'.
 *
 * Post-Condition:
 *  The ASID is freed and may be allocated again later.
 */
static void asid_free(u_int i) {
    int index = i >> 5;
    int inner = i & 31;
    asid_bitmap[index] &= ~(1 << inner);
}

/* Overview:
 *   Map [va, va+size) of virtual address space to physical [pa, pa+size) in the
'pgdir'. Use
 *   permission bits 'perm | PTE_V' for the entries.
 *
 * Pre-Condition:
 *   'pa', 'va' and 'size' are aligned to 'BY2PG'.
 */
static void map_segment(Pde *pgdir, u_int asid, u_long pa, u_long va, u_int size,
u_int perm) {

    assert(pa % BY2PG == 0);
    assert(va % BY2PG == 0);
    assert(size % BY2PG == 0);
    struct Page *p;
    /* Step 1: Map virtual address space to physical address space. */
    for (int i = 0; i < size; i += BY2PG) {
        /*
         * Hint:
         *  Map the virtual page 'va + i' to the physical page 'pa + i' using
'page_insert'.
         *  Use 'pa2page' to get the 'struct Page *' of the physical address.
         */
        /* Exercise 3.2: Your code here. */
        p = pa2page(pa + i);
        p -> pp_ref ++;
        page_insert(pgdir, asid, p, va + i, perm | PTE_V);
    }
}

/* Overview:
 *  This function is to make a unique ID for every env
 *
 * Pre-Condition:
 *  e should be valid
 *
 * Post-Condition:
 *  return e's envid on success
 */
u_int mkenvid(struct Env *e) {
    static u_int i = 0;
    return ((++i) << (1 + LOG2NENV)) | (e - envs);
}

/* Overview:
 *   Convert an existing 'envid' to an 'struct Env *'.
 *   If 'envid' is 0, set '*penv = curenv', otherwise set '*penv =
&envs[ENVX(envid)]'.
 *   In addition, if 'checkperm' is non-zero, the requested env must be either
'curenv' or its
 *   immediate child.
 *
 * Pre-Condition:
 *   'penv' points to a valid 'struct Env *'.
 *
 * Post-Condition:
 *   return 0 on success, and set '*penv' to the env.
 *   return -E_BAD_ENV on error (invalid 'envid' or 'checkperm' violated).
 */
int envid2env(u_int envid, struct Env **penv, int checkperm) {
    struct Env *e;

    /* Step 1: Assign value to 'e' using 'envid'. */
    /* Hint:
     *   If envid is zero, set 'penv' to 'curenv'.
     *   You may want to use 'ENVX'.
     */
    /* Exercise 4.3: Your code here. (1/2) */

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

    /* Step 2: Check when 'checkperm' is non-zero. */
    /* Hints:
     *   Check whether the calling env has sufficient permissions to manipulate
the
     *   specified env, i.e. 'e' is either 'curenv' or its immediate child.
     *   If violated, return '-E_BAD_ENV'.
     */
    /* Exercise 4.3: Your code here. (2/2) */

    /* Step 3: Assign 'e' to '*penv'. */
    *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' returns
'envs[0]'.
 *
 * Hints:
 *   You may use these macro definitions below: 'LIST_INIT', 'TAILQ_INIT',
'LIST_INSERT_HEAD'
 */
void env_init(void) {
    printk("start env_init()...\n");
    int i;
    /* Step 1: Initialize 'env_free_list' with 'LIST_INIT' and 'env_sched_list'
```

```
with
     * 'TAILQ_INIT'. */
    /* Exercise 3.1: Your code here. (1/2) */
    LIST_INIT(&env_free_list);
    TAILQ_INIT(&env_sched_list);
    /* Step 2: Traverse the elements of 'envs' array, set their status to
'ENV_FREE' and insert
     * them into the 'env_free_list'. Make sure, after the insertion, the order of
envs in the
     * list should be the same as they are in the 'envs' array. */

    /* Exercise 3.1: Your code here. (2/2) */
    // int num = sizeof(envs);
    for (int i = NENV - 1; i >= 0; i--) {
        envs[i].env_status = ENV_FREE;
        LIST_INSERT_HEAD(&env_free_list, &envs[i], env_link);
    }
    /*
     * We want to map 'UPAGES' and 'UENVS' to *every* user space with PTE_G
permission (without
     * PTE_D), then user programs can read (but cannot write) kernel data
structures 'pages' and
     * 'envs'.
     *
     * Here we first map them into the *template* page directory 'base_pgdir'.
     * Later in 'env_setup_vm', we will copy them into each 'env_pgdir'.
     */
    struct Page *p;
    panic_on(page_alloc(&p));
    p->pp_ref++;

    base_pgdir = (Pde *)page2kva(p);
    map_segment(base_pgdir, 0, PADDR(pages), UPAGES, ROUND(npage * sizeof(struct
Page), BY2PG),
            PTE_G);
    map_segment(base_pgdir, 0, PADDR(envs), UENVS, ROUND(NENV * sizeof(struct
Env), BY2PG),
            PTE_G);
    printk("finish env_init()...\n");
}

/* Overview:
 *   Initialize the user address space for 'e'.
 */
static int env_setup_vm(struct Env *e) {
    /* Step 1:
     *   Allocate a page for the page directory with 'page_alloc'.
     *   Increase its 'pp_ref' and assign its kernel address to 'e->env_pgdir'.
     *
     * Hint:
     *   You can get the kernel address of a specified physical page using
'page2kva'.
     */
    struct Page *p;
    try(page_alloc(&p));
    /* Exercise 3.3: Your code here. */

    p -> pp_ref ++;
    e -> env_pgdir = (Pde *)page2kva(p);
    /* Step 2: Copy the template page directory 'base_pgdir' to 'e->env_pgdir'. */
    /* Hint:
     *   As a result, the address space of all envs is identical in [UTOP, UVPT).
     *   See include/mmu.h for layout.
     */
    memcpy(e->env_pgdir + PDX(UTOP), base_pgdir + PDX(UTOP),
           sizeof(Pde) * (PDX(UVPT) - PDX(UTOP)));

    /* Step 3: Map its own page table at 'UVPT' with readonly permission.
     * As a result, user programs can read its page table through 'UVPT' */
    e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V;
    return 0;
}

/* Overview:
 *   Allocate and initialize a new env.
 *   On success, the new env is stored at '*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 basic fields of the new Env are set up.
 *   return < 0 on error, if no free env, no free asid, or 'env_setup_vm' failed.
 *
 * Hints:
 *   You may need to use these functions or macros:
 *     'LIST_FIRST', 'LIST_REMOVE', 'mkenvid', 'asid_alloc', 'env_setup_vm'
 *   Following fields of Env should be set up:
 *     'env_id', 'env_asid', 'env_parent_id', 'env_tf.regs[29]',
'env_tf.cp0_status',
 *     'env_user_tlb_mod_entry', 'env_runs'
 */
int env_alloc(struct Env **new, u_int parent_id) {
    int r;
    struct Env *e;

    /* Step 1: Get a free Env from 'env_free_list' */
    /* Exercise 3.4: Your code here. (1/4) */
    if (LIST_EMPTY(&env_free_list)) {
        *new = NULL;
        return -E_NO_FREE_ENV;
    }
    e = LIST_FIRST(&env_free_list);
    /* Step 2: Call a 'env_setup_vm' to initialize the user address space for this
new Env. */
    /* Exercise 3.4: Your code here. (2/4) */
    env_setup_vm(e);
    /* Step 3: Initialize these fields for the new Env with appropriate values:
     *   'env_user_tlb_mod_entry' (lab4), 'env_runs' (lab6), 'env_id' (lab3),
'env_asid' (lab3),
     *   'env_parent_id' (lab3)
     *
     * Hint:
     *   Use 'asid_alloc' to allocate a free asid.
     *   Use 'mkenvid' to allocate a free envid.
     */
    e->env_user_tlb_mod_entry = 0; // for lab4
    e->env_runs = 0;           // for lab6
    /* Exercise 3.4: Your code here. (3/4) */
    e->env_id = mkenvid(e);
    try(asid_alloc(&(e->env_asid)));
    e->env_parent_id = parent_id;
    /* Step 4: Initialize the sp and 'cp0_status' in 'e->env_tf'. */
    // Timer interrupt (STATUS_IM4) will be enabled.
    e->env_tf.cp0_status = STATUS_IM4 | STATUS_KUp | STATUS_IEp;
    // Keep space for 'argc' and 'argv'.
    e->env_tf.regs[29] = USTACKTOP - sizeof(int) - sizeof(char **);
    /* Step 5: Remove the new Env from env_free_list. */
    /* Exercise 3.4: Your code here. (4/4) */
    LIST_REMOVE(e, env_link);
    *new = e;
    return 0;
}

/* Overview:
 *   Load a page into the user address space of an env with permission 'perm'.
 *   If 'src' is not NULL, copy the 'len' bytes from 'src' into 'offset' at this
page.
 *
 * Pre-Condition:
 *   'offset + len' is not larger than 'BY2PG'.
 *
 * Hint:
 *   The address of env structure is passed through 'data' from 'elf_load_seg',
where this function
 *   works as a callback.
 *
 */
static int load_icode_mapper(void *data, u_long va, size_t offset, u_int perm,
const void *src,
                 size_t len) {
    struct Env *env = (struct Env *)data;
    struct Page *p;
    int r;

    /* Step 1: Allocate a page with 'page_alloc'. */
    /* Exercise 3.5: Your code here. (1/2) */
    page_alloc(&p);
    /* Step 2: If 'src' is not NULL, copy the 'len' bytes started at 'src' into
'offset' at this
     * page. */
    // Hint: You may want to use 'memcpy'.
    if (src != NULL) {
        /* Exercise 3.5: Your code here. (2/2) */
        memcpy((void *)page2kva(p) + offset, src, len);
    }

    /* Step 3: Insert 'p' into 'env->env_pgdir' at 'va' with 'perm'. */
    return page_insert(env->env_pgdir, env->env_asid, p, va, perm);
}

/* Overview:
 *   Load program segments from 'binary' into user space of the env 'e'.
 *   'binary' points to an ELF executable image of 'size' bytes, which contains
both text and data
 *   segments.
 */
static void load_icode(struct Env *e, const void *binary, size_t size) {
    printk("begin load_icode()...\n");
    /* Step 1: Use 'elf_from' to parse an ELF header from 'binary'. */
    const Elf32_Ehdr *ehdr = elf_from(binary, size); // 解析ELF头文件
    if (!ehdr) {
        panic("bad elf at %x", binary);
    }

    /* Step 2: Load the segments using 'ELF_FOREACH_PHDR_OFF' and 'elf_load_seg'.
     * As a loader, we just care about loadable segments, so parse only program
headers here.
     */
    size_t ph_off;
    ELF_FOREACH_PHDR_OFF (ph_off, ehdr) {
        Elf32_Phdr *ph = (Elf32_Phdr *)(binary + ph_off);
        if (ph->p_type == PT_LOAD) {
            // 'elf_load_seg' is defined in lib/elfloader.c
            // 'load_icode_mapper' defines the way in which a page in this segment
            // should be mapped.
            panic_on(elf_load_seg(ph, binary + ph->p_offset, load_icode_mapper,
e));
        }
    }

    /* Step 3: Set 'e->env_tf.cp0_epc' to 'ehdr->e_entry'. */
    /* Exercise 3.6: Your code here. */
    // uint32_t ehdr -> e_entry    unsigned long e -> env_tf.cp0_epc
    e -> env_tf.cp0_epc = ehdr -> e_entry;
    printk("finish load_icode()\n");
}

/* Overview:
 *   Create a new env with specified 'binary' and 'priority'.
 *   This is only used to create early envs from kernel during initialization,
before the
 *   first created env is scheduled.
 *
 * Hint:
```

```
 *   'binary' is an ELF executable image in memory.
 */
struct Env *env_create(const void *binary, size_t size, int priority) {
    printk("begin env_create()...\n");
    struct Env *e;
    /* Step 1: Use 'env_alloc' to alloc a new env, with 0 as 'parent_id'. */
    /* Exercise 3.7: Your code here. (1/3) */
    env_alloc(&e, 0);
    e -> env_parent_id = 0;
    /* Step 2: Assign the 'priority' to 'e' and mark its 'env_status' as runnable.
*/
    /* Exercise 3.7: Your code here. (2/3) */
    printk("priority is :%d, status is :%d.\n", priority, ENV_RUNNABLE);
    e -> env_pri = priority;
    e -> env_status = ENV_RUNNABLE;
    /* Step 3: Use 'load_icode' to load the image from 'binary', and insert 'e'
into
     * 'env_sched_list' using 'TAILQ_INSERT_HEAD'. */
    /* Exercise 3.7: Your code here. (3/3) */
    load_icode(e, binary, size);
    TAILQ_INSERT_HEAD(&env_sched_list, (e), env_sched_link);
    printk("finish env_create()\n");
    return e;
}

/* Overview:
 *  Free 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.*/
    printk("[%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, e->env_asid,
                        (pdeno << PDSHIFT) | (pteno << PGSHIFT));
            }
        }
        /* Hint: free the page table itself. */
        e->env_pgdir[pdeno] = 0;
        page_decref(pa2page(pa));
        /* Hint: invalidate page table in TLB */
        tlb_invalidate(e->env_asid, UVPT + (pdeno << PGSHIFT));
    }
    /* Hint: free the page directory. */
    page_decref(pa2page(PADDR(e->env_pgdir)));
    /* Hint: free the ASID */
    asid_free(e->env_asid);
    /* Hint: invalidate page directory in TLB */
    tlb_invalidate(e->env_asid, UVPT + (PDX(UVPT) << PGSHIFT));
    /* Hint: return the environment to the free list. */
    e->env_status = ENV_FREE;
    LIST_INSERT_HEAD((&env_free_list), (e), env_link);
    TAILQ_REMOVE(&env_sched_list, (e), env_sched_link);
}

/* Overview:
 *  Free env e, and schedule 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;
        printk("i am killed ... \n");
        schedule(1);
    }
}

/* Overview:
 *   This function is depended by our judge framework. Please do not modify it.
 */
static inline void pre_env_run(struct Env *e) {
#ifdef MOS_SCHED_MAX_TICKS
    static int count = 0;
    if (count > MOS_SCHED_MAX_TICKS) {
        printk("%4d: ticks exceeded the limit %d\n", count, MOS_SCHED_MAX_TICKS);
        halt();
    }
    printk("%4d: %08x\n", count, e->env_id);
    count++;
#endif
#ifdef MOS_SCHED_END_PC
    struct Trapframe *tf = (struct Trapframe *)KSTACKTOP - 1;
    u_int epc = tf->cp0_epc;
    if (epc == MOS_SCHED_END_PC) {
        printk("env %08x reached end pc: 0x%08x, $v0=0x%08x\n", e->env_id, epc,
               tf->regs[2]);
        env_destroy(e);
        schedule(0);
    }
#endif
}
```

```c
extern void env_pop_tf(struct Trapframe *tf, u_int asid)
__attribute__((noreturn));

/* Overview:
 *   Switch CPU context to the specified env 'e'.
 *
 * Post-Condition:
 *   Set 'e' as the current running env 'curenv'.
 *
 * Hints:
 *   You may use these functions: 'env_pop_tf'.
 */
void env_run(struct Env *e) {
    printk("start env_run()\n");
    assert(e->env_status == ENV_RUNNABLE);
    pre_env_run(e); // WARNING: DO NOT MODIFY THIS LINE!
    /* Step 1:
     *   If 'curenv' is NULL, this is the first time through.
     *   If not, we may be switching from a previous env, so save its context into
     *   'curenv->env_tf' first.
     */
    if (curenv) {
        curenv->env_tf = *((struct Trapframe *)KSTACKTOP - 1);
    }
    /* Step 2: Change 'curenv' to 'e'. */
    curenv = e;
    curenv->env_runs++; // lab6
    /* Step 3: Change 'cur_pgdir' to 'curenv->env_pgdir', switching to its address
space. */
    /* Exercise 3.8: Your code here. (1/2) */
    cur_pgdir = curenv -> env_pgdir;
    /* Step 4: Use 'env_pop_tf' to restore the curenv's saved context (registers)
and return/go
     * to user mode.
     *
     * Hint:
     *  - You should use 'curenv->env_asid' here.
     *  - 'env_pop_tf' is a 'noreturn' function: it restores PC from 'cp0_epc'
thus not
     *    returning to the kernel caller, making 'env_run' a 'noreturn' function
as well.
     */
    /* Exercise 3.8: Your code here. (2/2) */
    env_pop_tf(&curenv->env_tf, curenv->env_asid);
    printk("finish env_run()\n");
}

void env_check() {
    struct Env *pe, *pe0, *pe1, *pe2;
    struct Env_list fl;
    u_long page_addr;
    /* should be able to allocate three envs */
    pe0 = 0;
    pe1 = 0;
    pe2 = 0;

    assert(env_alloc(&pe0, 0) == 0);
    assert(env_alloc(&pe1, 0) == 0);
    assert(env_alloc(&pe2, 0) == 0);

    assert(pe0);
    assert(pe1 && pe1 != pe0);
    assert(pe2 && pe2 != pe1 && pe2 != pe0);

    /* temporarily steal the rest of the free envs */
    fl = env_free_list;
    /* now this env_free list must be empty! */
    LIST_INIT(&env_free_list);
    /* should be no free memory */
    assert(env_alloc(&pe, 0) == -E_NO_FREE_ENV);
    /* recover env_free_list */
    env_free_list = fl;
    printk("pe0->env_id %d\n", pe0->env_id);
    printk("pe1->env_id %d\n", pe1->env_id);
    printk("pe2->env_id %d\n", pe2->env_id);

    assert(pe0->env_id == 2048);
    assert(pe1->env_id == 4097);
    assert(pe2->env_id == 6146);
    printk("env_init() work well!\n");

    /* 'UENVS' and 'UPAGES' should have been correctly mapped in *template* page
directory
     * 'base_pgdir'. */
    for (page_addr = 0; page_addr < npage * sizeof(struct Page); page_addr +=
BY2PG) {
        assert(va2pa(base_pgdir, UPAGES + page_addr) == PADDR(pages) + page_addr);
    }
    for (page_addr = 0; page_addr < NENV * sizeof(struct Env); page_addr += BY2PG)
{
        assert(va2pa(base_pgdir, UENVS + page_addr) == PADDR(envs) + page_addr);
    }
    /* check env_setup_vm() work well */
    printk("pe1->env_pgdir %x\n", pe1->env_pgdir);

    assert(pe2->env_pgdir[PDX(UTOP)] == base_pgdir[PDX(UTOP)]);
    assert(pe2->env_pgdir[PDX(UTOP) - 1] == 0);
    printk("env_setup_vm passed!\n");

    printk("pe2`s sp register %x\n", pe2->env_tf.regs[29]);

    /* free all env allocated in this function */
    TAILQ_INSERT_TAIL(&env_sched_list, pe0, env_sched_link);
    TAILQ_INSERT_TAIL(&env_sched_list, pe1, env_sched_link);
    TAILQ_INSERT_TAIL(&env_sched_list, pe2, env_sched_link);

    env_free(pe2);
```

```c
    env_free(pe1);
    env_free(pe0);

    printk("env_check() succeeded!\n");
}

void envid2env_check() {
    struct Env *pe, *pe0, *pe2;
    assert(env_alloc(&pe0, 0) == 0);
    assert(env_alloc(&pe2, 0) == 0);
    int re;
    pe2->env_status = ENV_FREE;
    re = envid2env(pe2->env_id, &pe, 0);

    assert(re == -E_BAD_ENV);

    pe2->env_status = ENV_RUNNABLE;
    re = envid2env(pe2->env_id, &pe, 0);

    assert(pe->env_id == pe2->env_id && re == 0);

    curenv = pe0;
    re = envid2env(pe2->env_id, &pe, 1);
    assert(re == -E_BAD_ENV);
    printk("envid2env() work well!\n");
}
```