清华大学本科生考试试题专用纸

考试课程：**操作系统（A 卷）**　　时间：2012 年 04 月 10 日上午 9:50~11:50

任课教师：_____ 系别：_____ 班级：_____ 学号：_____ 姓名：_____

答卷注意事项:

1. 在开始答题前，请在试题纸和答卷本上写明系别、班级、学号和姓名。
2. 在答卷本上答题时，要写明题号，不必抄题。
3. 答题时，要书写清楚和整洁。
4. 请注意回答所有试题。本试卷有 7 个题目，共 14 页。
5. 考试完毕，必须将试题纸和答卷本一起交回。

一、（10 分）1）操作系统的微内核结构特征是什么？2）它有什么优点和缺点？3）在微内核结构中，内存管理、进程通信、文件系统、I/O 管理这几种操作系统功能中，哪些是放在内核中的？哪些是放在用户态的？

二、（14 分）1）试描述进程执行中利用堆栈实现函数调用和返回的过程。2）请补全下面 print_stackframe()函数所缺的代码，以利用函数调用时保存在堆栈中的信息输出嵌套调用的函数入口地址和参数信息。

==========kern-ucore/arch/i386/debug/kdebug.c==========

```
/* *
 * print_debuginfo - read and print the stat information for the address @eip,
 * and info.eip_fn_addr should be the first address of the related function.
 * */
void
print_debuginfo(uintptr_t eip) {
……

}

static uint32_t read_eip(void) __attribute__((noinline));

static uint32_t
read_eip(void) {
……

}

/* *
 * print_stackframe - print a list of the saved eip values from the nested 'call'
 * instructions that led to the current point of execution
 *
 * The x86 stack pointer, namely esp, points to the lowest location on the stack
```

```
 * that is currently in use. Everything below that location in stack is free. Pushing
 * a value onto the stack will invole decreasing the stack pointer and then writing
 * the value to the place that stack pointer pointes to. And popping a value do the
 * opposite.
 *
 * The ebp (base pointer) register, in contrast, is associated with the stack
 * primarily by software convention. On entry to a C function, the function's
 * prologue code normally saves the previous function's base pointer by pushing
 * it onto the stack, and then copies the current esp value into ebp for the duration
 * of the function. If all the functions in a program obey this convention,
 * then at any given point during the program's execution, it is possible to trace
 * back through the stack by following the chain of saved ebp pointers and determining
 * exactly what nested sequence of function calls caused this particular point in the
 * program to be reached. This capability can be particularly useful, for example,
 * when a particular function causes an assert failure or panic because bad arguments
 * were passed to it, but you aren't sure who passed the bad arguments. A stack
 * backtrace lets you find the offending function.
 *
 * The inline function read_ebp() can tell us the value of current ebp. And the
 * non-inline function read_eip() is useful, it can read the value of current eip,
 * since while calling this function, read_eip() can read the caller's eip from
 * stack easily.
 *
 * In print_debuginfo(), the function debuginfo_eip() can get enough information about
 * calling-chain. Finally print_stackframe() will trace and print them for debugging.
 *
 * Note that, the length of ebp-chain is limited. In boot/bootasm.S, before jumping
 * to the kernel entry, the value of ebp has been set to zero, that's the boundary.
 * */
void
print_stackframe(void) {
    uint32_t ebp = read_ebp(), eip = read_eip();

    int i, j;
    for (i = 0; ebp != 0 && i < 10; i ++) {
        kprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        uint32_t *args = (uint32_t *) --YOUR CODE 1--;
        for (j = 0; j < 4; j ++) {
            kprintf("0x%08x ", args[j]);
        }
        kprintf("\n");
        print_debuginfo(eip - 1);
        eip = ((uint32_t *) --YOUR CODE 2--;
        ebp = ((uint32_t *) --YOUR CODE 3--;
    }
}
```

三、（18 分）1）系统调用接口是操作系统内核向用户进程提供操作系统服务的接口。试描述用户进程通过系统调用使用操作系统服务的过程。2）gettime_msec 是一个获取当前系统时间的系统调用。请补全该系统调用的实现代码。

==========libs-user-ucore/ulib.c==========

```
unsigned int
gettime_msec(void) {
    return (unsigned int)sys_gettime();
}
```

==========libs-user-ucore/syscall.c==========

```
size_t
sys_gettime(void) {
    return (size_t) --YOUR CODE 4--;
}
```

==========libs-user-ucore/arch/i386/syscall.c==========

```
#define MAX_ARGS            5

uint32_t
syscall(int num, ...) {
    va_list ap;
    va_start(ap, num);
    uint32_t a[MAX_ARGS];
    int i;
    for (i = 0; i < MAX_ARGS; i ++) {
        a[i] = va_arg(ap, uint32_t);
    }
    va_end(ap);

    uint32_t ret;
    asm volatile (
        "int %1;"
        : "=a" (ret)
        : "i" (T_SYSCALL),
          "a" (num),
          "d" (a[0]),
          "c" (a[1]),
          "b" (a[2]),
          "D" (a[3]),
          "S" (a[4])
        : "cc", "memory");
    return ret;
}
```

```
=============libs-user-ucore/common/unistd.h=============

/* syscall number */
#define SYS_exit            1
#define SYS_fork            2
#define SYS_wait            3
#define SYS_exec            4
#define SYS_clone           5
#define SYS_exit_thread     9
#define SYS_yield           10
#define SYS_sleep           11
#define SYS_kill            12
#define SYS_gettime         17
#define SYS_getpid          18
#define SYS_brk             19

……

=============kern-ucore/arch/i386/glue-ucore/trap.c=============

static void
trap_dispatch(struct trapframe *tf) {
    char c;

    int ret;
    switch (tf->tf_trapno) {
    case T_DEBUG:
    case T_BRKPT:
        debug_monitor(tf);
        break;
    case T_PGFLT:
        if ((ret = pgfault_handler(tf)) != 0) {
            print_trapframe(tf);
            if (pls_read(current) == NULL) {
                panic("handle pgfault failed. %e\n", ret);
            }
            else {
                if (trap_in_kernel(tf)) {
                    panic("handle pgfault failed in kernel mode. %e\n", ret);
                }
                kprintf("killed by kernel.\n");
                do_exit(-E_KILLED);
            }
        }
        break;
    case T_SYSCALL:
```

```
        --YOUR CODE 5--;
        break;
    case IRQ_OFFSET + IRQ_TIMER:
        ticks ++;
        assert(pls_read(current) != NULL);
        run_timer_list();
        break;
    case IRQ_OFFSET + IRQ_COM1:
    case IRQ_OFFSET + IRQ_KBD:
        if ((c = cons_getc()) == 13) {
            debug_monitor(tf);
        }
        else {
            extern void dev_stdin_write(char c);
            dev_stdin_write(c);
        }
        break;
    case IRQ_OFFSET + IRQ_IDE1:
    case IRQ_OFFSET + IRQ_IDE2:
        /* do nothing */
        break;
    default:
        print_trapframe(tf);
        if (pls_read(current) != NULL) {
            kprintf("unhandled trap.\n");
            do_exit(-E_KILLED);
        }
        panic("unexpected trap in kernel.\n");
    }
}

void
trap(struct trapframe *tf) {
    // used for previous projects
    if (pls_read(current) == NULL) {
        trap_dispatch(tf);
    }
    else {
        // keep a trapframe chain in stack
        struct trapframe *otf = pls_read(current)->tf;
        pls_read(current)->tf = tf;

        bool in_kernel = trap_in_kernel(tf);

        --YOUR CODE 6--;
```

```
        pls_read(current)->tf = otf;
        if (!in_kernel) {
            may_killed();
            if (pls_read(current)->need_resched) {
                schedule();
            }
        }
    }
}

=============kern-ucore/arch/i386/syscall/syscall.c=============

……

static uint32_t
sys_gettime(uint32_t arg[]) {
    return (int)ticks;
}

……

static uint32_t (*syscalls[])(uint32_t arg[]) = {
    [SYS_exit]              sys_exit,
    [SYS_fork]              sys_fork,
    [SYS_wait]              sys_wait,
    [SYS_exec]              sys_exec,
    [SYS_clone]             sys_clone,
    [SYS_exit_thread]       sys_exit_thread,
    [SYS_yield]             sys_yield,
    [SYS_kill]              sys_kill,
    [SYS_sleep]             sys_sleep,
    [SYS_gettime]           --YOUR CODE 7--,
    [SYS_getpid]            sys_getpid,
……
};

#define NUM_SYSCALLS        ((sizeof(syscalls)) / (sizeof(syscalls[0])))

void
syscall(void) {
    struct trapframe *tf = pls_read(current)->tf;
    uint32_t arg[5];
    int num = tf->tf_regs.reg_eax;
    if (num >= 0 && num < NUM_SYSCALLS) {
        if (syscalls[num] != NULL) {
            arg[0] = tf->tf_regs.reg_edx;
```

```
            arg[1] = tf->tf_regs.reg_ecx;
            arg[2] = tf->tf_regs.reg_ebx;
            arg[3] = tf->tf_regs.reg_edi;
            arg[4] = tf->tf_regs.reg_esi;
            tf->tf_regs.reg_eax = --YOUR CODE 8--;
            return ;
        }
    }
    print_trapframe(tf);
    panic("undefined syscall %d, pid = %d, name = %s.\n",
            num, pls_read(current)->pid, pls_read(current)->name);
}
```

四、（18 分）1）试利用图示描述伙伴系统（Buddy System）中对物理内存的分配和回收过程。2）请补全下面伙伴系统实现中所缺的代码。

==========kern-ucore/arch/i386/mm/buddy_pmm.c==========

```
// {1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}
// from 2^0 ~ 2^10
#define MAX_ORDER 10
static free_area_t free_area[MAX_ORDER + 1];

//x from 0 ~ MAX_ORDER
#define free_list(x) (free_area[x].free_list)
#define nr_free(x) (free_area[x].nr_free)

#define MAX_ZONE_NUM 10
struct Zone {
    struct Page *mem_base;
} zones[MAX_ZONE_NUM] = {{NULL}};

//buddy_init - init the free_list(0 ~ MAX_ORDER) & reset nr_free(0 ~ MAX_ORDER)
static void
buddy_init(void) {
    int i;
    for (i = 0; i <= MAX_ORDER; i ++) {
        list_init(&free_list(i));
        nr_free(i) = 0;
    }
}

//buddy_init_memmap - build free_list for Page base follow  n continuing pages.
static void
buddy_init_memmap(struct Page *base, size_t n) {
……
```

```
}

//getorder - return order, the minmal 2^order >= n
static inline size_t
getorder(size_t n) {
    size_t order, order_size;
    for (order = 0, order_size = 1; order <= MAX_ORDER; order ++, order_size <<= 1) {
        if (n <= order_size) {
            return order;
        }
    }
    panic("getorder failed. %d\n", n);
}

//buddy_alloc_pages_sub - the actual allocation implimentation, return a page whose siz
e >=n,
//                      - the remaining free parts insert to other free list
static inline struct Page *
buddy_alloc_pages_sub(size_t order) {
    assert(order <= MAX_ORDER);
    size_t cur_order;
    for (cur_order = order; cur_order <= MAX_ORDER; cur_order ++) {
        if (!list_empty(&free_list(cur_order))) {
            list_entry_t *le = list_next(&free_list(cur_order));
            struct Page *page = le2page(le, page_link);
            nr_free(cur_order) --;
            --YOUR CODE 9--(le);
            size_t size = 1 << cur_order;
            while (cur_order > order) {
                cur_order --;
                size >>= 1;
                struct Page *buddy = page + size;
                buddy->property = cur_order;
                SetPageProperty(buddy);
                nr_free(cur_order) ++;
                --YOUR CODE 10--(&free_list(cur_order), &(buddy->page_link));
            }
            ClearPageProperty(page);
            return page;
        }
    }
    return NULL;
}

//buddy_alloc_pages - call buddy_alloc_pages_sub to alloc 2^order>=n pages
static struct Page *
```

```
buddy_alloc_pages(size_t n) {
    assert(n > 0);
    size_t order = getorder(n), order_size = (1 << order);
    struct Page *page = buddy_alloc_pages_sub(order);
    if (page != NULL && n != order_size) {
        free_pages(page + n, order_size - n);
    }
    return page;
}

//page_is_buddy - Does this page belong to the No. zone_num Zone & this page
//              -  be in the continuing page block whose size is 2^order pages?
static inline bool
page_is_buddy(struct Page *page, size_t order, int zone_num) {
    if (page2ppn(page) < npage) {
        if (page->zone_num == zone_num) {
            return !PageReserved(page) && PageProperty(page) && page->property == order
;
        }
    }
    return 0;
}

//page2idx - get the related index number idx of continuing page block which this page
belongs to
static inline ppn_t
page2idx(struct Page *page) {
    return page - zones[page->zone_num].mem_base;
}

//idx2page - get the related page according to the index number idx of continuing page
block
static inline struct Page *
idx2page(int zone_num, ppn_t idx) {
    return zones[zone_num].mem_base + idx;
}

//buddy_free_pages_sub - the actual free implimentation, should consider how to
//                     - merge the adjacent buddy block
static void
buddy_free_pages_sub(struct Page *base, size_t order) {
    ppn_t buddy_idx, page_idx = page2idx(base);
    assert((page_idx & ((1 << order) - 1)) == 0);
    struct Page *p = base;
    for (; p != base + (1 << order); p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
```

```
        p->flags = 0;
        set_page_ref(p, 0);
    }
    int zone_num = base->zone_num;
    while (order < MAX_ORDER) {
        buddy_idx = page_idx ^ (1 << order);
        struct Page *buddy = idx2page(zone_num, buddy_idx);
        if (!page_is_buddy(buddy, order, zone_num)) {
            break;
        }
        nr_free(order) --;
        --YOUR CODE 11--(&(buddy->page_link));
        ClearPageProperty(buddy);
        page_idx &= buddy_idx;
        order ++;
    }
    struct Page *page = idx2page(zone_num, page_idx);
    page->property = order;
    SetPageProperty(page);
    nr_free(order) ++;
    --YOUR CODE 12--(&free_list(order), &(page->page_link));
}

//buddy_free_pages - call buddy_free_pages_sub to free n continuing page block
static void
buddy_free_pages(struct Page *base, size_t n) {
    assert(n > 0);
    if (n == 1) {
        buddy_free_pages_sub(base, 0);
    }
    else {
        size_t order = 0, order_size = 1;
        while (n >= order_size) {
            assert(order <= MAX_ORDER);
            if ((page2idx(base) & order_size) != 0) {
                buddy_free_pages_sub(base, order);
                base += order_size;
                n -= order_size;
            }
            order ++;
            order_size <<= 1;
        }
        while (n != 0) {
            while (n < order_size) {
                order --;
                order_size >>= 1;
```

```
            }
            buddy_free_pages_sub(base, order);
            base += order_size;
            n -= order_size;
        }
    }
}

//buddy_nr_free_pages - get the nr: the number of free pages
static size_t
buddy_nr_free_pages(void) {
    size_t ret = 0, order = 0;
    for (; order <= MAX_ORDER; order ++) {
        ret += nr_free(order) * (1 << order);
    }
    return ret;
}

//buddy_check - check the correctness of buddy system
static void
buddy_check(void) {
……

}

//the buddy system pmm
const struct pmm_manager buddy_pmm_manager = {
    .name = "buddy_pmm_manager",
    .init = buddy_init,
    .init_memmap = buddy_init_memmap,
    .alloc_pages = buddy_alloc_pages,
    .free_pages = buddy_free_pages,
    .nr_free_pages = buddy_nr_free_pages,
    .check = buddy_check,
};
```

五、（16 分）1）试用图示描述 32 位 X86 系统在采用 4KB 页面大小时的页表结构。2）在采用 4KB 页面大小的 32 位 X86 的 ucore 虚拟存储系统中，进程页面的起始地址由宏 VPT 确定。

```
#define VPT                 0xFAC00000
```

请计算：2a)页目录（PDE）在虚拟地址空间中的起始地址；2b)虚拟地址 0X87654321 对应的页目录项和页表项的虚拟地址。

六、（14 分）1）试描述 ucore 的进程创建系统调用 fork（）的基本过程。2）请补全 fork 系统调用的实现代码。

==========kern-ucore/process/proc.c==========

```
// get_pid - alloc a unique pid for process
static int
get_pid(void) {
……
}

……

int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }

    ret = -E_NO_MEM;

    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = --YOUR CODE 13--;
    list_init(&(proc->thread_group));
    assert(current->wait_state == 0);

    assert(current->time_slice >= 0);
    proc->time_slice = current->time_slice / 2;
    current->time_slice -= proc->time_slice;

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    if (copy_sem(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    if (copy_fs(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_sem;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_fs;
    }
```

```
    if (copy_thread(clone_flags, proc, stack, tf) != 0) {
        goto bad_fork_cleanup_sem;
    }

    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = --YOUR CODE 14--;
        hash_proc(proc);
        set_links(proc);
        if (clone_flags & CLONE_THREAD) {
            list_add_before(&(current->thread_group), &(proc->thread_group));
        }
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

    ret = --YOUR CODE 15--;
fork_out:
    return ret;

bad_fork_cleanup_fs:
    put_fs(proc);
bad_fork_cleanup_sem:
    put_sem_queue(proc);
bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
    kfree(proc);
    goto fork_out;
}

=============kern-ucore/arch/i386/process/proc.c=============

// forkret -- the first kernel entry point of a new thread/process
// NOTE: the addr of forkret is setted in copy_thread function
//       after switch_to, the current proc will execute here.
static void
forkret(void) {
    forkrets(pls_read(current)->tf);
}

……

// copy_thread - setup the trapframe on the  process's kernel stack top and
```

```
//             - setup the kernel entry point and stack of process
int
copy_thread(uint32_t clone_flags, struct proc_struct *proc,
            uintptr_t esp, struct trapframe *tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE) - 1;
    *(proc->tf) = *tf;
    proc->tf->tf_regs.reg_eax = 0;
    proc->tf->tf_esp = esp;
    proc->tf->tf_eflags |= FL_IF;

    proc->context.eip = (uintptr_t) --YOUR CODE 16--;
    proc->context.esp = (uintptr_t)(proc->tf);

    return 0;
}
```

七、（10 分）试用图示描述五状态进程模型，要求给出状态描述和各状态间的变迁。