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：2011 年 04 月 12 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 7 页。
5. 考试完毕, 必须将试题纸和答卷本一起交回。

一、 (15分)下面是ucore中用于按需分页处理过程的内核代码。请补全其中所缺的代码，以正确完成按需分页过程。

kern/trap/trap.h
```
---------------------------------
...
struct trapframe {
    struct pushregs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    // below here only when crossing rings, such as from user to kernel
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
...
---------------------------------
```

kern/trap/trap.c
```
---------------------------------
...
static int
```

```
pgfault_handler(struct trapframe *tf) {
    extern struct mm_struct *check_mm_struct;
    print_pgfault(tf);
    if (check_mm_struct != NULL) {
        return do_pgfault(check_mm_struct, tf->tf_err, rcr2());
    }
    panic("unhandled page fault.\n");
}

static void
trap_dispatch(struct trapframe *tf) {
     char c;
     int ret;
     switch ( --YOUR CODE 1-- ) {
     ...
     case T_PGFLT:
          if ( --YOUR CODE 2-- ) != 0) {
               print_trapframe(tf);
               if (current == NULL) {
                    panic("handle pgfault failed. %e\n", ret);
               }
               else { ... }
          }
          break;
  ...
}

void
trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
}
...
// do_pgfault - interrupt handler to process the page fault execption
int
do_pgfault(struct mm_struct *mm, uint32_t error_code, uintptr_t addr) {
    int ret = -E_INVAL;
    struct vma_struct *vma = find_vma(mm, addr);
    if (vma == NULL || vma->vm_start > addr) {
        goto failed;
    }

    switch (error_code & 3) {
    default:
           /* default is 3: write, present */
    case 2: /* write, not present */
```

```
        if (!(vma->vm_flags & VM_WRITE)) {
            goto failed;
        }
        break;
    case 1: /* read, present */
        goto failed;
    case 0: /* read, not present */
        if (!(vma->vm_flags & (VM_READ | VM_EXEC))) {
            goto failed;
        }
    }

    uint32_t perm = PTE_U;
    if (vma->vm_flags & VM_WRITE) {
        perm |= PTE_W;
    }
    addr = ROUNDDOWN(addr, PGSIZE);

    ret = -E_NO_MEM;

    if (pgdir_alloc_page(mm->pgdir, addr, perm) == 0) {
        goto failed;
    }
    ret = 0;

failed:
    return ret;
}
...
---------------------------------

Pmm.h
---------------------------------
...
//ppn is physical page number
static inline ppn_t
page2ppn(struct Page *page) {
    return  --YOUR CODE 3--;
}
//pa is physical address
static inline uintptr_t
page2pa(struct Page *page) {
    return --YOUR CODE 4--;
}
...
---------------------------------
```

```
pmm.c
---------------------------------
...
// virtual address of physicall page array
struct Page *pages;
// amount of physical memory (in pages)
size_t npage = 0;
// virtual address of boot-time page directory
pde_t *boot_pgdir = NULL;
……
// pgdir_alloc_page - call alloc_page & page_insert functions to
//                - allocate a page size memory & setup an addr map
//                - pa<->la with linear address la and the PDT pgdir
struct Page *
pgdir_alloc_page(pde_t *pgdir, uintptr_t la, uint32_t perm) {
    struct Page *page = alloc_page();
    if (page != NULL) {
        if (page_insert(pgdir, page, la, perm) != 0) {
            free_page(page);
            return NULL;
        }
    }
    return page;
}
...

//page_insert - build the map of phy addr of an Page with the linear addr la
// paramemters:
//  pgdir: the kernel virtual base address of PDT
//  page:  the Page which need to map
//  la:    the linear address need to map
//  perm:  the permission of this Page which is setted in related pte
// return value: always 0
//note: PT is changed, so the TLB need to be invalidate
int
page_insert(pde_t *pgdir, struct Page *page, uintptr_t la, uint32_t perm) {
    pte_t *ptep = get_pte(pgdir, la, 1);
    if (ptep == NULL) {
        return -E_NO_MEM;
    }
    page_ref_inc(page);
    if (*ptep & PTE_P) {
        struct Page *p = pte2page(*ptep);
        if (p == page) {
            page_ref_dec(page);
        }
```

```
        else {
            page_remove_pte(pgdir, la, ptep);
        }
    }
    *ptep = --YOUR CODE 5--
    tlb_invalidate(pgdir, la);
    return 0;
}
--------------------------------
```

二、 (17分)（1）试描述内存分配算法-伴侣系统(Buddy System)的工作原理；
（2）假定某计算机系统中有1MB内存采用伴侣系统进行内存分配，请用图示描述下列存储分配和释放请求序列所对应的存储分配情况。
1)进程A请求分配34KB内存；
2)进程B请求分配66KB内存；
3)进程C请求分配35KB内存；
4)进程D请求分配67KB内存；
5)进程C释放它所占用的内存区域；
6)进程A释放它所占用的内存区域；
7)进程B释放它所占用的内存区域；
8)进程D释放它所占用的内存区域；

三、 (16分) 请说明五状态进程模型中的状态和状态含义，并说明哪些状态会发生转换以及转换的原因。

四、 (10分) （1）试描述虚拟存储管理系统中的时钟置换算法（Clock Page Replacement）的工作原理；
（2）假定在一个采用时钟置换算法的虚拟存储系统中某进程分配了4个物理页面，当进程按c, a, d, b, e, c, b, a, d, b, c, a, d的序列进行页面访问时，会出现多少次缺页？要求说明过程。如果需要，你可以补充假定条件。

五、 (20分)（1）请使用图示来简要描述采用多级页面的虚拟存储系统中的地址变换过程。
（2）假定在一个32位计算机系统中，采用多级页表结构来实现虚拟存储管理，页面大小为4KB，每级页表大小为一个页面的大小，每个页表项占8字节。请问该计算机系统中要使用几级页表？各级各占多少位？对于虚拟地址0X87654321对应的各级页表号分别是多少？

六、 (10分)给出程序fork.c的输出结果。注：（1）getpid()和getppid()是两个系统调用，分别返回本进程标识和父进程标识。（2）你可以假定每次新进程创建时生成的进程标识是顺序加1得到的，该程序执行时创建的第一个进程的标识为1000。

fork.c

```
----------------------

/* Includes */
#include <unistd.h>     /* Symbolic Constants */
#include <sys/types.h>  /* Primitive System Data Types */
#include <errno.h>      /* Errors */
#include <stdio.h>      /* Input/Output */
```

```
#include <sys/wait.h>   /* Wait for Process Termination */
#include <stdlib.h>     /* General Utilities */

int main()
{
   pid_t childpid; /* variable to store the child's pid */
   int retval;     /* child process: user-provided return code */
   int status;     /* parent process: child's exit status */

   /* only 1 int variable is needed because each process would have its
      own instance of the variable
      here, 2 int variables are used for clarity */

   /* now create new process */
   childpid = fork();

   if (childpid >= 0) /* fork succeeded */
   {
      if (childpid == 0) /* fork() returns 0 to the child process */
      {
         printf("CHILD: I am the child process!\n");
         printf("CHILD: Here's my PID: %d\n", getpid());
         printf("CHILD: My parent's PID is: %d\n", getppid());
         printf("CHILD: The value of my copy of childpid is: %d\n", childpid);
         printf("CHILD: Sleeping for 1 second...\n");
         sleep(1); /* sleep for 1 second */
         printf("CHILD: Enter an exit value (0 to 255): ");
         scanf(" %d", &retval);
         printf("CHILD: Goodbye!\n");
         exit(retval); /* child exits with user-provided return code */
      }
      else /* fork() returns new pid to the parent process */
      {
         printf("PARENT: I am the parent process!\n");
         printf("PARENT: Here's my PID: %d\n", getpid());
         printf("PARENT: The value of my copy of childpid is %d\n", childpid);
         printf("PARENT: I will now wait for my child to exit.\n");
         wait(&status); /* wait for child to exit, and store its status */
         printf("PARENT: Child's exit code is: %d\n", WEXITSTATUS(status));
         printf("PARENT: Goodbye!\n");
         exit(0);  /* parent exits */
      }
   }
   else /* fork returns -1 on failure */
   {
      perror("fork"); /* display error message */
```

```
        exit(0);
    }
}
----------------------
```

七、 (12分) 请求分页管理系统中，假定页表内容如下表所示：

页号	页框（Page Frame）号	有效位（存在位）
0	101H	1
1	N/A	0
2	254H	1

页面大小为4KB，一次内存的访问时间是100ns，一次快表（TLB）的访问时间是10ns，处理一次缺页的平均时间是10^8ns（已包含更新TLB和页表的时间），进程的驻留集大小固定为2，采用最近最少使用置换算法（LRU）和局部淘汰策略。假设：TLB初始为空；地址转换时先访问TLB，若TLB未命中，再访问页表（忽略访问页表之后的TLB更新时间）；有效位为0表示页面不在内存，会产生缺页中断，缺页中断处理后，返回到产生缺页中断的指令处重新执行。设有虚地址访问序列2362H、1565H、25A5H，请问：

（1） 依次访问上述三个虚地址，各需多少时间？给出计算过程。

（2） 基于上述访问序列，虚地址1565H的物理地址是多少？请说明理由。