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06月22日下午2:30~4:30

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23页。
5. 考试完毕，必须将试题纸和答卷本一起交回。

一、（20分）下面是与read()系统调用实现相关源代码。请补全其中所缺的代码，以正确完成从用户态函数read()到内核态函数sysfile_read()的参数传递和返回过程。提示：每处需要补全的代码只需要一行，一共有10个空要填。

user/libs/file.c

```
---------------------------------
...
int
read(int fd, void *base, size_t len) {
        ...(1)...
}
...
---------------------------------
```

user/libs/syscall.c

```
---------------------------------
...
#define MAX_ARGS                            5

static inline int
syscall(int num, ...) {
        int ret;
        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);

        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;
}
...
int
sys_read(int fd, void *base, size_t len) {
         ...(2)
}
...
---------------------------------
libs/stdarg.h
---------------------------------
...
typedef char * va_list;

#define
__va_size(type)                                                             \
         ((sizeof(type) + (sizeof(long) - 1)) / sizeof(long) * sizeof(long))

#define va_start(ap, last)                                                   \
         ((ap) = (va_list)&(last) + __va_size(last))

#define va_arg(ap, type)                                                    \
         (*(type *)((ap) += __va_size(type), (ap) - __va_size(type)))

#define va_end(ap)                ((void)0)
...
---------------------------------

libs/unistd.h
---------------------------------
...
#define T_SYSCALL                            0x80

/* syscall number */
...
#define SYS_read                        102
#define SYS_write                       103
...
```

```
---------------------------------
kern/syscall/syscall.c
---------------------------------
...
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;
};
...
---------------------------------
kern/trap/trap.c
---------------------------------
...
static void
trap_dispatch(struct trapframe *tf) {
        char c;

        int ret;

        switch (...(3)...) {
        case T_DEBUG:
        case T_BRKPT:
                debug_monitor(tf);
                break;
        case T_PGFLT:
                if ((ret = pgfault_handler(tf)) != 0) {
                        print_trapframe(tf);
                        if (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);
                                }
                                cprintf("killed by kernel.\n");
                                do_exit(-E_KILLED);
                        }
                }
                break;
        case T_SYSCALL:
                ...(4)...
                break;
        case IRQ_OFFSET + IRQ_TIMER:
                ticks ++;
                assert(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 (current != NULL) {
                        cprintf("unhandled trap.\n");
                        do_exit(-E_KILLED);
                }
                panic("unexpected trap in kernel.\n");
        }
}
void
trap(struct trapframe *tf) {
        // used for previous projects
        if (current == NULL) {
                trap_dispatch(tf);
        }
        else {
```

```
                        // keep a trapframe chain in stack
                        struct trapframe *otf = current->tf;
                        current->tf = tf;

                        bool in_kernel = trap_in_kernel(tf);

                        ...(5)...

                        current->tf = otf;
                        if (!in_kernel) {
                                if (current->flags & PF_EXITING) {
                                        do_exit(-E_KILLED);
                                }
                                if (current->need_resched) {
                                        schedule();
                                }
                        }
                }
}
...
---------------------------------
kern/syscall/syscall.c
---------------------------------
...
static int
sys_read(uint32_t arg[]) {
        int fd = (int)arg[0];
        size_t len = (size_t)...(6)...;
        void *base = (void *)...(7)...;
        ...(8a)...
}
...
static int (*syscalls[])(uint32_t arg[]) = {
...
        [SYS_read]                              sys_read,
        [SYS_write]                             sys_write,
...
        [SYS_mkfifo]                           sys_mkfifo,
};

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


void
syscall(void) {
        struct trapframe *tf = current->tf;
        uint32_t arg[5];
        int num = tf->...(8b)..
        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 = ...(9)..
                                return ;
                        }
        }
        print_trapframe(tf);
        panic("undefined syscall %d, pid = %d, name = %s.\n",
                                num, current->pid, current->name);
}
...
---------------------------------
kern/fs/sysfile.c
---------------------------------
...
int
sysfile_read(int fd, void *base, size_t len) {
        struct mm_struct *mm = current->mm;
        if (len == 0) {
                        return 0;
        }
        if (!file_testfd(fd, 1, 0)) {
                        return -E_INVAL;
        }
        void *buffer;
        if ((buffer = kmalloc(IOBUF_SIZE)) == NULL) {
                        return -E_NO_MEM;
        }

        int ret = 0;
        size_t copied = 0, alen;
        while (len != 0) {
                        if ((alen = IOBUF_SIZE) > len) {
                                alen = len;
                        }
                        ret = ...(10).
                        if (alen != 0) {
                                lock_mm(mm);
                                {
                                        if (copy_to_user(mm, base, buffer, alen)) {
                                                assert(len >= alen);
                                                base += alen, len -= alen, copied
+= alen;
                                        }
                                        else if (ret == 0) {
                                                ret = -E_INVAL;
                                        }
                                }
                                unlock_mm(mm);
                        }
                        if (ret != 0 || alen == 0) {
                                goto out;
                        }
        }

out:
        kfree(buffer);
        if (copied != 0) {
                        return copied;
        }
        return ret;
}
...
---------------------------------
kern/fs/file.c
---------------------------------
...
int
file_read(int fd, void *base, size_t len, size_t *copied_store) {
        int ret;
        struct file *file;
        *copied_store = 0;
        if ((ret = fd2file(fd, &file)) != 0) {
                        return ret;
        }
        if (!file->readable) {
                        return -E_INVAL;
        }
        filemap_acquire(file);

        struct iobuf __iob, *iob = iobuf_init(&__iob, base, len, file->pos);
        ret = vop_read(file->node, iob);

        size_t copied = iobuf_used(iob);
        if (file->status == FD_OPENED) {
```

```
                   file->pos += copied;
         }
         *copied_store = copied;
         filemap_release(file);
         return ret;
}
...
---------------------------------
```

二、　（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；在进程标识为1000的命令解释程序shell中启动该程序的执行。

```
#include <sys/types.h>
#include <unistd.h>

/* getpid() and fork() are system calls declared in unistd.h.  They return */
/* values of type pid_t.  This pid_t is a special type for process ids. */
/* It's equivalent to int. */

int main(void)
{
    pid_t childpid;

    int x = 5;
       int i;
    childpid = fork();
    for ( i = 0;  i < 2;  i++)  {
       printf("This is process %d; childpid = %d; The parent of this process has
id %d; i = %d; x = %d\n", getpid(), childpid, getppid(), i, x);
             sleep(1);
       x++;
    }

    return 0;
}
```

三、　(18分)调度器是操作系统内核中依据调度算法进行进程切换选择的模块。1）试描述时间片轮转算法（Round Robin）的基本原理。2）下面代码是ucore中调度器和时间片轮转算法的实现代码。请补全其中所缺代码，以实现调度器和调度算法的功能。提示：每处需要补全的代码只需要一行，一共有7个空要填。

```
sched.h
---------------------------------------------
…
struct proc_struct;

typedef struct {
    unsigned int expires;
    struct proc_struct *proc;
    list_entry_t timer_link;
} timer_t;

#define le2timer(le, member)            \
    to_struct((le), timer_t, member)

static inline timer_t *
timer_init(timer_t *timer, struct proc_struct *proc, int expires) {
    timer->expires = expires;
    timer->proc = proc;
    list_init(&(timer->timer_link));
    return timer;
}

struct run_queue;

struct sched_class {
    const char *name;
    void (*init)(struct run_queue *rq);
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);
    struct proc_struct *(*pick_next)(struct run_queue *rq);
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};

struct run_queue {
    list_entry_t run_list;
    unsigned int proc_num;
    int max_time_slice;
    list_entry_t rq_link;
};

#define le2rq(le, member)            \
    to_struct((le), struct run_queue, member)

void sched_init(void);
void wakeup_proc(struct proc_struct *proc);
void schedule(void);
void add_timer(timer_t *timer);
void del_timer(timer_t *timer);
void run_timer_list(void);

extern struct proc_struct *idleproc, *initproc, *current;
extern struct proc_struct *kswapd;
…
```

```
---------------------------------------------
sched.c
---------------------------------------------
…
static list_entry_t timer_list;

static struct sched_class *sched_class;

static struct run_queue *rq;

static inline void
sched_class_enqueue(struct proc_struct *proc) {
    if (proc != idleproc) {
        sched_class->enqueue(rq, proc);
    }
}

static inline void
sched_class_dequeue(struct proc_struct *proc) {
    sched_class->dequeue(rq, proc);
}

static inline struct proc_struct *
sched_class_pick_next(void) {
    return sched_class->pick_next(rq);
}

static void
sched_class_proc_tick(struct proc_struct *proc) {
    if (proc != idleproc) {
        sched_class->proc_tick(rq, proc);
    }
    else {
        proc->need_resched = 1;
    }
}

static struct run_queue __rq[4];

void
sched_init(void) {
    list_init(&timer_list);

    rq = __rq;
    list_init(&(rq->rq_link));
    rq->max_time_slice = 8;
    int i;
    for (i = 1; i < sizeof(__rq) / sizeof(__rq[0]); i ++) {
        list_add_before(&(rq->rq_link), &(__rq[i].rq_link));
        __rq[i].max_time_slice = rq->max_time_slice * (1 << i);
    }

    sched_class = &MLFQ_sched_class;
    sched_class->init(rq);

    cprintf("sched class: %s\n", sched_class->name);
}

void
wakeup_proc(struct proc_struct *proc) {
    assert(proc->state != PROC_ZOMBIE);
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        if (proc->state != PROC_RUNNABLE) {
            proc->state = PROC_RUNNABLE;
            proc->wait_state = 0;
            sched_class_enqueue(proc);
        }
        else {
            warn("wakeup runnable process.\n");
        }
    }
    local_intr_restore(intr_flag);
}

void
schedule(void) {
    bool intr_flag;
    struct proc_struct *next;
    local_intr_save(intr_flag);
    {
        current->need_resched = 0;
        if (current->state == PROC_RUNNABLE) {
            ...(1)
        }
        if ((next = sched_class_pick_next()) != NULL) {
            ...(2)...
        }
    }
    local_intr_restore(intr_flag);
    if (next == NULL) {
        next = ...(3)...;
    }
    next->runs ++;
    if (next != current) {
        ...(4)...
    }
}
…
-----------------------------------------------
sched_RR.c
-----------------------------------------------
…
static void
RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));
    rq->proc_num = 0;
}

static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        ...(5)...
    }
    proc->rq = rq;
    rq->proc_num ++;
}

static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}

static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
```

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        ...(6)...
    }
    if (proc->time_slice == 0) {
        ...(7)...
    }
}

struct sched_class RR_sched_class = {
    .name = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};
-----------------------------------------------
proc.c
-----------------------------------------------
…
// proc_run - make process "proc" running on cpu
// NOTE: before call switch_to, should load  base addr of "proc"'s new PDT
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            load_esp0(next->kstack + KSTACKSIZE);
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
…
-----------------------------------------------
```

四、 （22分）管程是操作系统提供的一种进程同步机制，利用管程可解决进程间通信时遇到的同步互斥问题。读者-写者问题（Reader-writer problem）是一个经典的同步问题。写者优先的读者-写者问题是指，假定有多个并发的读进程和写进程都要访问一个共享的数据结构，要求：(1)读写互斥；(2)写写互斥；(3)允许多个读进程同时访问；(4)只要有写进程提出申

请，其后提出申请的读进程就必须等待该写进程完成访问。下面是ucore中管程机制和写者优先的读者-写者问题的实现代码。请尝试补全其中所缺的代码，以正确实现管程机制和读者-写者间的读写操作协调。提示：文件“cdt_wf.c”中的补全代码可能需要在一处加多行代码，其他需要补全的代码只需要一行，一共有11个空要填。

```
condition.h
------------------------------------------------
…
typedef struct {
    int numWaiting;
    int valid;
    wait_queue_t wait_queue;
} condition_t;

#define cdtid2cdt(cdt_id)                            \
    ((condition_t *)((uintptr_t)(cdt_id) + KERNBASE))

#define cdt2cdtid(cdt)                              \
    ((cdt_t)((uintptr_t)(cdt) - KERNBASE))


void
condition_value_init(condition_t *cdt) {
    ...(1)...
    cdt->valid=1;
        wait_queue_init(&(cdt->wait_queue));
}

int
condition_init(){
    condition_t *cdt;
    if ((cdt = kmalloc(sizeof(condition_t))) != NULL) {
        condition_value_init( cdt );
    }
    if (cdt != NULL) {
            return cdt2cdtid(cdt);
    }
    return -E_INVAL;
}

int
condition_free(cdt_t cdt_id) {
    condition_t *cdt = cdtid2cdt(cdt_id);
    int ret = -E_INVAL;
    if (cdt != NULL) {
        bool intr_flag;
        local_intr_save(intr_flag);
        {
                cdt->valid = 0, ret = 0;
                wakeup_queue(&(cdt->wait_queue), WT_INTERRUPTED, 1);
                kfree(cdt);
        }
        local_intr_restore(intr_flag);
    }
    return ret;
}

int
condition_wait(cdt_t cdt_id, klock_t kl_id){

    condition_t *cdt = cdtid2cdt(cdt_id);
    bool intr_flag;
    local_intr_save(intr_flag);
    ...(2)
    wait_t __wait, *wait = &__wait;
    ...(3)
    local_intr_restore(intr_flag);

    sys_unlock(kl_id);
    schedule();
    sys_lock(kl_id);

    //local_intr_save(intr_flag);
    //wait_current_del(&(cdt->wait_queue), wait);
    //local_intr_restore(intr_flag);

    if (wait->wakeup_flags != WT_UCONDITION) {
        return wait->wakeup_flags;
    }
    return 0;
}

int
condition_signal(cdt_t cdt_id){
    condition_t *cdt = cdtid2cdt(cdt_id);
    if (cdt == NULL) {
        return -E_INVAL;
    }

    bool intr_flag;
    local_intr_save(intr_flag);
    if (cdt->numWaiting > 0) {
        wait_t *wait;
        if ((wait = wait_queue_first(&(cdt->wait_queue))) != NULL) {
                assert(wait->proc->wait_state == WT_UCONDITION);
                ...(4)...
        }
        ...(5)...
    }
    local_intr_restore(intr_flag);
    return 0;
}
…
------------------------------------------------
ulib.c
------------------------------------------------
…
cdt_t
cdt_init(){
    return sys_cdt_init();
}

int
cdt_signal(cdt_t cdt_id){
    return sys_cdt_signal(cdt_id);
}

int
cdt_wait(cdt_t cdt_id ,klock_t klock_id){
    return sys_cdt_wait(cdt_id ,klock_id);
}

int
cdt_free(cdt_t cdt_id){
    return sys_cdt_free(cdt_id);
}

klock_t
klock_init(){
    return sys_klock_init();
}

int
klock_aquire(klock_t klock_id){
    return sys_klock_aquire(klock_id);
}

int
```

```
klock_release(klock_t klock_id){
    return sys_klock_release(klock_id);
}

int
klock_free(klock_t klock_id){
    return sys_klock_free(klock_id);
}
…
-----------------------------------------------
cdt_wf.c
-----------------------------------------------
…
int *active_reader ;     // # count of active readers
int *active_writer ; // # count of active writers
int *waiting_reader ;    // # count of waiting readers
int *waiting_writer ;    // # count of waiting writers
cdt_t cdt_okToRead;
cdt_t cdt_okToWrite;
klock_t lock;

void
failed(void) {
    cprintf("FAIL: T.T\n");
    exit(-1);
}

void
init(void) {
    if ((cdt_okToRead = cdt_init()) < 0 || (cdt_okToWrite = cdt_init()) < 0) {
        failed();
    }
    if ((lock = klock_init()) < 0) {
        failed();
    }
    if ((active_reader = shmem_malloc(sizeof(int))) == NULL || (active_writer =
shmem_malloc(sizeof(int))) == NULL
    || (waiting_reader = shmem_malloc(sizeof(int))) == NULL || (waiting_writer =
shmem_malloc(sizeof(int))) == NULL) {
        failed();
    }
    *active_reader = *active_writer = *waiting_reader = *waiting_writer = 0;
}

void
check_init_value(void) {
    if (cdt_okToRead < 0 || cdt_okToWrite < 0 ) {
    failed();
    }
    if (lock < 0 ) {
        failed();
    }
    if (*active_reader != 0 || *active_writer != 0 || *waiting_reader != 0 ||
*waiting_writer != 0) {
        failed();
    }
}

void
free_wf(void){
    if (  cdt_free(cdt_okToRead) < 0 ||  cdt_free(cdt_okToWrite) < 0 ){
        scprintf(" conditon free failed! \n");
        exit(-1);
    }
    if (  klock_free(lock) < 0 ){
        scprintf(" kernal lock free failed! \n");
        exit(-1);
    }
}

void
start_read(void) {
    klock_aquire(lock);
        ...(6)...
        klock_release(lock);
}

void
done_read(void) {
    klock_aquire(lock);
        ...(7)...
    klock_release(lock);
}

void
start_write(void) {
    klock_aquire(lock);
        ...(8)...
    klock_release(lock);
}

void
```

```
done_write(void) {
    klock_aquire(lock);
        ...(9)...
        if ((*waiting_writer) > 0) {
            ...(10)...
        }
        else if ((*waiting_reader) > 0) {
        int wakecount=0;
        while(...(11)...){
            cdt_signal(cdt_okToRead);
            wakecount++;
        }
        }
    klock_release(lock);
}

void
writer(int id, int time) {
    scprintf("writer %d: (pid:%d) arrive \n", id, getpid());
        start_write();
        scprintf("    writer_wf %d: (pid:%d) start %d\n", id, getpid(), time);
    sleep(time);
    scprintf("    writer_wf %d: (pid:%d) end %d\n", id, getpid(), time);
        done_write();
}

void
reader(int id, int time) {
    scprintf("reader %d: (pid:%d) arrive\n", id, getpid());
    start_read();
    scprintf("    reader_wf %d: (pid:%d) start %d\n", id, getpid(), time);
    sleep(time);
    scprintf("    reader_wf %d: (pid:%d) end %d\n", id, getpid(), time);
    done_read();
}


void
read_test_wf(void) {
…
}

void
write_test_wf(void) {
…
}
```

```
void
read_write_test_wf(void) {
…
}

int
main(void) {
   init();
   read_test_wf();
   write_test_wf();
   read_write_test_wf();
   free_wf();
   cprintf("condition reader_writer_wf_test pass..\n");
   return 0;
}
-----------------------------------------------
wait.c
-----------------------------------------------
…
void
wait_init(wait_t *wait, struct proc_struct *proc) {
   wait->proc = proc;
   wait->wakeup_flags = WT_INTERRUPTED;
   list_init(&(wait->wait_link));
}

void
wait_queue_init(wait_queue_t *queue) {
   list_init(&(queue->wait_head));
}

void
wait_queue_add(wait_queue_t *queue, wait_t *wait) {
   assert(list_empty(&(wait->wait_link)) && wait->proc != NULL);
   wait->wait_queue = queue;
   list_add_before(&(queue->wait_head), &(wait->wait_link));
}

void
wait_queue_del(wait_queue_t *queue, wait_t *wait) {
   assert(!list_empty(&(wait->wait_link)) && wait->wait_queue == queue);
   list_del_init(&(wait->wait_link));
}

wait_t *
```

```
wait_queue_next(wait_queue_t *queue, wait_t *wait) {
    assert(!list_empty(&(wait->wait_link)) && wait->wait_queue == queue);
    list_entry_t *le = list_next(&(wait->wait_link));
    if (le != &(queue->wait_head)) {
        return le2wait(le, wait_link);
    }
    return NULL;
}

wait_t *
wait_queue_prev(wait_queue_t *queue, wait_t *wait) {
    assert(!list_empty(&(wait->wait_link)) && wait->wait_queue == queue);
    list_entry_t *le = list_prev(&(wait->wait_link));
    if (le != &(queue->wait_head)) {
        return le2wait(le, wait_link);
    }
    return NULL;
}

wait_t *
wait_queue_first(wait_queue_t *queue) {
    list_entry_t *le = list_next(&(queue->wait_head));
    if (le != &(queue->wait_head)) {
        return le2wait(le, wait_link);
    }
    return NULL;
}

wait_t *
wait_queue_last(wait_queue_t *queue) {
    list_entry_t *le = list_prev(&(queue->wait_head));
    if (le != &(queue->wait_head)) {
        return le2wait(le, wait_link);
    }
    return NULL;
}

bool
wait_queue_empty(wait_queue_t *queue) {
    return list_empty(&(queue->wait_head));
}

bool
wait_in_queue(wait_t *wait) {
    return !list_empty(&(wait->wait_link));
}
```

```
void
wakeup_wait(wait_queue_t *queue, wait_t *wait, uint32_t wakeup_flags, bool del)
{
   if (del) {
      wait_queue_del(queue, wait);
   }
   wait->wakeup_flags = wakeup_flags;
   wakeup_proc(wait->proc);
}

void
wakeup_first(wait_queue_t *queue, uint32_t wakeup_flags, bool del) {
   wait_t *wait;
   if ((wait = wait_queue_first(queue)) != NULL) {
      wakeup_wait(queue, wait, wakeup_flags, del);
   }
}

void
wakeup_queue(wait_queue_t *queue, uint32_t wakeup_flags, bool del) {
   wait_t *wait;
   if ((wait = wait_queue_first(queue)) != NULL) {
      if (del) {
         do {
            wakeup_wait(queue, wait, wakeup_flags, 1);
         } while ((wait = wait_queue_first(queue)) != NULL);
      }
      else {
         do {
            wakeup_wait(queue, wait, wakeup_flags, 0);
         } while ((wait = wait_queue_next(queue, wait)) != NULL);
      }
   }
}
void
wait_current_set(wait_queue_t *queue, wait_t *wait, uint32_t wait_state) {
   assert(current != NULL);
   wait_init(wait, current);
   current->state = PROC_SLEEPING;
   current->wait_state = wait_state;
   wait_queue_add(queue, wait);
}
-----------------------------------------------
proc.h
-----------------------------------------------
```

```
…
//the wait state
#define WT_CHILD       (0x00000001 | WT_INTERRUPTED)  // wait child process
#define WT_TIMER       (0x00000002 | WT_INTERRUPTED)  // wait timer
#define WT_KSWAPD      0x00000003                   // wait kswapd to free page
#define WT_KSEM        0x00000100                   // wait kernel semaphore
#define WT_USEM        (0x00000101 | WT_INTERRUPTED)  // wait user semaphore
#define WT_EVENT_SEND  (0x00000110 | WT_INTERRUPTED)  // wait the sending event
#define WT_EVENT_RECV  (0x00000111 | WT_INTERRUPTED)  // wait the recving event
#define WT_MBOX_SEND   (0x00000120 | WT_INTERRUPTED)  // wait the sending mbox
#define WT_MBOX_RECV   (0x00000121 | WT_INTERRUPTED)  // wait the recving mbox
#define WT_UCONDITION  (0x00000130 | WT_INTERRUPTED)  // wait user condition
--liuruilin
#define WT_INTERRUPTED 0x80000000// the wait state could be interrupted


…
-----------------------------------------------
```

五、（10分）银行家算法(Banker's Algorithm)是一种在资源分配过程中避免出现死锁的算法，资源管理者可以有进程申请资源时，使用银行家算法来判断分配相应资源后是否可能出现死锁。试回答下列问题。

1）形成死锁的条件是什么？

2）试用伪代码描述银行家算法。

3）假设系统中有A、B、C和D这四类资源，有P1、P2和P3这三个进程正在使用这些资源。下面某次资源申请后的资源占用情况。请问这个状态是否安全？如果是安全的，请给出一个可能的资源分配和回收序列。

当前的可用资源情况：

A	B	C	D
3	1	1	2

当前各进程的已分配资源情况：

	A	B	C	D
P1	1	0	3	3
P2	1	2	2	1
P3	1	2	1	0

各进程声称的最大资源申请情况：

	A	B	C	D
P1	1	2	3	4
P2	3	3	2	2
P3	1	3	5	0

六、（10分）基本的文件组织方式有哪几种？请用图示方式描述UNIX文件系统UFS的文件组织方式。

七、（10分）磁盘缓存置换算法的作用是什么？试简要描述访问频率置换算法(Frequency-based Replacement)的基本原理。磁盘调度算法的作用是什么？试简要描述扫描算法（SCAN）的基本原理。