**מספר זהות:**

סמסטר א, מועד א.
תאריך: 10/2/2015
שעה: 0900
משך הבחינה: 3 שעות.
חומר עזר: אסור

בחינה בקורס: **מבוא למערכות הפעלה**

מרצה: ד"ר כרמי מרימוביץ

# מדבקית
# ברקוד

**הנחיות:**

טופס הבחינה כולל 12 עמודים (כולל עמוד זה).
**תשובות צריכות לכלול הסבר.**
**כתיבת תשובות עמומות תוריד נקודות.**
**כתיבת תשובות (או חלקן) שלא קשורות לשאלות תוריד נקודות.**
**יש לענות בשטח המוקצה לכך.**

# בהצלחה!

1. (35 נק') כיתבו רוטינה שרצה בקרנל ומעתיקה זיכרון ממרחב כתובות של תהליך אחד למרחב כתובות של תהליך שני:

```
int copyrange(uint fromVa, uint toVa, uint len, ___...);
```

(א) הרוטינה תעתיק `len` בתים מכתובת `fromVa` במרחב כתובות תהליך אחד לכתובת `toVa` במרחב כתובות תהליך אחר.

(ב) בהגדרה למעלה חסרים ארגומנטים, השלימו אותם.

(ג) על הרוטינה לבצע את כל בדיקות התקינות הרלוונטיות, כלומר כל כתובת שניגשים אליה חייבת להיות חוקית וקיימת במרחב הרלוונטי, ונמצאת בשטח הנמוך (לא באזור הקרנל).

(ד) כרגיל, אם משהוא אינו תקין יש להחזיר $-1$ כערך הפונקציה ולא לבצע דבר.

(ה) אם הכל תקין יש לבצע את ההעתקה ולהחזיר 0 כערך הפונקציה.

2. (30 נק') כיתבו רוטינה itrunc המקבלת מצביע ל־inode המתאר קובץ ומשחררת את כל בלוקים המתוארים בו, כלומר הקובץ יתפוס 0 בתים:

```
void itrunc(struct inode *ip);
```

3. (20 נק') ידוע שבדיסק ככל שמספרי הבלוקים קרובים זה לזה, הבלוקים קרובים זה לזה. שנו את דרייבר ה־ide כך שבסיום פעולה על בלוק מסוים נבחר לבצע פעולה על הבלוק הכי קרוב אליו (מתוך אלו שעליהם יש בקשת פעולה כמובן).

4. (15 נק') בקרנל יש קריאת מערכת שמיספרה 123, היא מקבלת שלושה ארגומנטים מטיפוס int, ושמה moshe. כיתבו רוטינה (ב־user-mode) שתאפשר לשיגרת C להפעיל קריאת מערכת זו. יש חשיבות ליעילות בקוד זה.

```
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this proces
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)
};

struct buf {
  int flags;
  uint dev;
  uint sector;
  struct buf *prev; // LRU cache list
  struct buf *next;
  struct buf *qnext; // disk queue
  uchar data[512];
};

static struct spinlock idelock;
static struct buf *idequeue;

void ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
```

```
    // cprintf("spurious IDE interrupt\n");
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, 512/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}

//PAGEBREAK!
// Sync buf with disk.
// If B_DIRTY is set, write buf to disk, clear B_DIRTY, set B_VALID.
// Else if B_VALID is not set, read buf from disk, set B_VALID.
void iderw(struct buf *b)
{
  struct buf **pp;

  if(!(b->flags & B_BUSY))
    panic("iderw: buf not busy");
  if((b->flags & (B_VALID|B_DIRTY)) == B_VALID)
    panic("iderw: nothing to do");
  if(b->dev != 0 && !havedisk1)
    panic("iderw: ide disk 1 not present");

  acquire(&idelock);  //DOC:acquire-lock
```

```
  // Append b to idequeue.
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext)  //DOC:insert-queue
    ;
  *pp = b;

  // Start disk if necessary.
  if(idequeue == b)
    idestart(b);

  // Wait for request to finish.
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    sleep(b, &idelock);
  }

  release(&idelock);
}

struct inode {
  uint dev;           // Device number
  uint inum;          // Inode number
  int ref;            // Reference count
  int flags;          // I_BUSY, I_VALID

  short type;         // copy of disk inode
  short major;
  short minor;
  short nlink;
  uint size;
  uint addrs[NDIRECT+1];
};

void ilock(struct inode *ip)
{
  struct buf *bp;
  struct dinode *dip;

  if(ip == 0 || ip->ref < 1)
```

```
    panic("ilock");

  acquire(&icache.lock);
  while(ip->flags & I_BUSY)
    sleep(ip, &icache.lock);
  ip->flags |= I_BUSY;
  release(&icache.lock);

  if(!(ip->flags & I_VALID)){
    bp = bread(ip->dev, IBLOCK(ip->inum));
    dip = (struct dinode*)bp->data + ip->inum%IPB;
    ip->type = dip->type;
    ip->major = dip->major;
    ip->minor = dip->minor;
    ip->nlink = dip->nlink;
    ip->size = dip->size;
    memmove(ip->addrs, dip->addrs, sizeof(ip->addrs));
    brelse(bp);
    ip->flags |= I_VALID;
    if(ip->type == 0)
      panic("ilock: no type");
  }
}

void iunlock(struct inode *ip)
{
  if(ip == 0 || !(ip->flags & I_BUSY) || ip->ref < 1)
    panic("iunlock");

  acquire(&icache.lock);
  ip->flags &= ~I_BUSY;
  wakeup(ip);
  release(&icache.lock);
}


void iput(struct inode *ip)
{
```

```
  acquire(&icache.lock);
  if(ip->ref == 1 && (ip->flags & I_VALID) && ip->nlink == 0){
    // inode has no links: truncate and free inode.
    if(ip->flags & I_BUSY)
      panic("iput busy");
    ip->flags |= I_BUSY;
    release(&icache.lock);
    itrunc(ip);
    ip->type = 0;
    iupdate(ip);
    acquire(&icache.lock);
    ip->flags = 0;
    wakeup(ip);
  }
  ip->ref--;
  release(&icache.lock);
}

static void bfree(int dev, uint b)
{
  struct buf *bp;
  struct superblock sb;
  int bi, m;

  readsb(dev, &sb);
  bp = bread(dev, BBLOCK(b, sb.ninodes));
  bi = b % BPB;
  m = 1 << (bi % 8);
  if((bp->data[bi/8] & m) == 0)
    panic("freeing free block");
  bp->data[bi/8] &= ~m;
  log_write(bp);
  brelse(bp);
}


#define KERNBASE 0x80000000         // First kernel virtual address
#define KERNLINK (KERNBASE+EXTMEM)  // Address where kernel is linked
```

```
static inline uint v2p(void *a) { return ((uint) (a))  - KERNBASE; }
static inline void *p2v(uint a) { return (void *) ((a) + KERNBASE); }

#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)
#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
typedef uint pde_t;
typedef uint pte_t;

static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc)
{
  pde_t *pde;
  pte_t *pgtab;

  pde = &pgdir[PDX(va)];
  if(*pde & PTE_P){
    pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
  } else {
    if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)
      return 0;
    // Make sure all those PTE_P bits are zero.
    memset(pgtab, 0, PGSIZE);
    // The permissions here are overly generous, but they can
    // be further restricted by the permissions in the page table
    // entries, if necessary.
    *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
  }
  return &pgtab[PTX(va)];
}
```