מספר זהות:

סמסטר א, מועד ב.
תאריך: 9/3/2015
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. (20 נק') השנוי הבא בוצע בקרנל. בזמן יצירת תהליך נשמר ב־int האחרון של מחסנית הקרנל (זה שמתחיל בכתובת proc->kstack + 4092) ערך חשוב (משמע אסור שהוא יהרס). גם בהטענת elf חדש קוד הקרנל עודכן בהתאם.

   איפה ומה צריך עוד לעדכן בקרנל.

2. (20 נק') כיתבו רוטינה בקרנל uint bmap(struct inode *ip, uint bn) כאשר:
ip מצביע ל־i-node המתאר קובץ, bn הוא מספר בלוק בקובץ.

הרוטינה מחזירה את מספרו של הבלוק הנ"ל בדיסק. אם הבלוק לא קיים בקובץ על הרוטינה להחזיר 0.

3. (20 נק') כיתבו רוטינה dirlink(struct inode *dp, char *name, int inum) בקרנל כאשר: dp מצביע ל־i-node המתאר directory, name מצביע למחרוזת המסתיימת ב־ NULL, inum מספר i-node.

על הרוטינה להכניס ל־directory הנתון רשומה שתאפשר גישה על ידי השם name לקובץ המתואר בידי ה־i-node שמספרו inum.

יש לבצע בדיקות תקינות רלוונטיות.

אם הפעולה מצליחה יש להחזיר 0. אחרת יש להחזיר -1.

4. (20 נק') כיתבו את הרוטינה static uint balloc(uint dev). הרוטינה מחזירה מספר של בלוק פנוי בדיסק מספר dev, ומסמנת את הבלוק הזה כתפוס.

5. (20 נק') כיתבו את הרוטינה

loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offset, uint sz).

מטרתה להטעין sz בתים ברציפות החל מבית offset בקובץ המתואר על־ידי ip אל הכתובות הוירטואליות הרציפות (במובן הטבלה pgdir) המתחילות בכתובת addr. הכתובות המתחילות ב־addr הן בשטח משתמש, והן תקינות.

הכתובת addr לא בהכרח על גבול דף!

```
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];
};

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];
};

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};

static struct inode *iget(uint dev, uint inum)
{
  struct inode *ip, *empty;

  acquire(&icache.lock);

  // Is the inode already cached?
  empty = 0;
```

```
  for(ip = &icache.inode[0]; ip < &icache.inode[NINODE]; ip++){
    if(ip->ref > 0 && ip->dev == dev && ip->inum == inum){
      ip->ref++;
      release(&icache.lock);
      return ip;
    }
    if(empty == 0 && ip->ref == 0)     // Remember empty slot.
      empty = ip;
  }

  // Recycle an inode cache entry.
  if(empty == 0)
    panic("iget: no inodes");

  ip = empty;
  ip->dev = dev;
  ip->inum = inum;
  ip->ref = 1;
  ip->flags = 0;
  release(&icache.lock);

  return ip;
}

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);
}

struct inode *dirlookup(struct inode *dp, char *name, uint *poff)
{
  uint off, inum;
  struct dirent de;

  if(dp->type != T_DIR)
    panic("dirlookup not DIR");

  for(off = 0; off < dp->size; off += sizeof(de)){
    if(readi(dp, (char*)&de, off, sizeof(de)) != sizeof(de))
      panic("dirlink read");
    if(de.inum == 0)
      continue;
    if(namecmp(name, de.name) == 0){
      // entry matches path element
      if(poff)
        *poff = off;
      inum = de.inum;
      return iget(dp->dev, inum);
    }
  }

  return 0;
}


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)];
}
```