מספר זהות:

סמסטר א, מועד ב.
תאריך: 5/3/2018
שעה: 1600
משך הבחינה: 3 שעות.
חומר עזר: אסור

בחינה בקורס: **מבוא למערכות הפעלה**

מרצה: ד"ר כרמי מרימוביץ
מתרגל: מר צבי מלמד

מדבקית
ברקוד

הנחיות:

טופס הבחינה כולל 15 עמודים (כולל עמוד זה).

קוד לא קריא לא יבדק!

בנוסף לנכונות, אלגנטיות תמיד נלקחת בחשבון!

יש לענות בשטח המוקצה לכך.

בהצלחה!

1. (30 נק') עליכם לממש בקרנל xv6 קריאות מערכת לניהול מנעולי קריאה/כתיבה. ב־user-mode הקריאות נראות כך:

```
int lock_read(int lid);
int lock_write(int lid);
int lock_none();
```

תהליך יכול לנעול מנעול אחד לכל היותר ברגע נתון. שיחרור מנעול מתבצע על־ידי קריאה ל־lock_none. כמובן שאם התהליך לא נעל מנעול אז זו שגיאה לשחרר.

מנעול יכול להימצא באחד משלושה מצבים: לא נעול, נעול לקריאה, נעול לכתיבה. מנעול יכול להיות נעול לכתיבה על־ידי תהליך אחד בלבד, ולא יכולה להיות נעילת קריאה עליו באותו זמן. מנעול יכול להיות נעול לקריאה על־ידי מספר לא מוגבל של תהליכים ולא יכולה להיות נעילת כתיבה עליו באותו זמן.

רוטינות הנעילה חוזרות לקורא רק לאחר שהמנעול ננעל, כלומר הן ממתינות עד שיתקיימו התנאים לביצוע הנעילה, ככל שצריך. אין צורך לדאוג מהרעבה.

(אין צורך לממש את הבאת הארגומנטים מ־user-mode.)

מזהה מנעול הוא מספר בן 32־ביטים ואסור באיסור חמור ליצור ווקטור של מנעולים! מותר להוסיף שדות למבנה proc.

```
struct spinlock tickslock;
int sys_sleep(void)
{
  int n;
  uint ticks0;

  if(argint(0, &n) < 0)
    return -1;
  acquire(&tickslock);
  ticks0 = ticks;
  while(ticks - ticks0 < n){
    if(proc->killed){
      release(&tickslock);
      return -1;
    }
    sleep(&ticks, &tickslock);
  }
  release(&tickslock);
  return 0;
}
```

מקום עבור תשובה 1

2. (20 נק') בשאלה זו סביבת העבודה הינה xv6 במצב קרנל. עליכם לכתוב רוטינה בקרנל שחתימתה
char *readpage(struct inode *ip, int vaddr);

רוטינה זו נקראת בקרנל בזמן ריצה של תהליך שקובץ ה־elf בו הוא משתמש נגיש על־ידי ip. הרוטינה תקצה דף בזיכרון ותקרא אליו מהקובץ את הדף שמכיל את הכתובת הוירטואלית vaddr של התוכנית. הרוטינה תחזיר את הכתובת הקרנלית אליה נקרא הדף. אם הכתובת vaddr לא קיימת הרוטינה תחזיר NULL. אם יש פחות מדף בקובץ, יש למלא באפסים כמובן. גם אם יש בעיית קריאה כלשהי יש להחזיר NULL.

בבקשה לא להתבלבל עם מרחב הכתובות הקיים של התהליך. הוא לא רלוונטי.

```
struct proghdr {
   uint type;
   uint off;
   uint vaddr;
   uint paddr;
   uint filesz;
   uint memsz;
   uint flags;
   uint align;
};
int exec(char *path, char **argv) {
   char *s, *last;
   int i, off;
   uint argc, sz, sp, ustack[3+MAXARG+1];
   struct elfhdr elf;
   struct inode *ip;
   struct proghdr ph;
   pde_t *pgdir, *oldpgdir;
   struct proc *curproc = myproc();

   begin_op();

   if ((ip = namei(path)) == 0){
      end_op();
      cprintf("exec: fail\n");
      return -1;
   }
   ilock(ip);
```

```
    pgdir = 0;

    // Check ELF header
    if (readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
        goto bad;
    if (elf.magic != ELF_MAGIC)
        goto bad;

    if ((pgdir = setupkvm()) == 0)
        goto bad;

// Load program into memory.
    sz = 0;
    for (i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
        if (readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
            goto bad;
        if (ph.type != ELF_PROG_LOAD)
            continue;
        if (ph.memsz < ph.filesz)
            goto bad;
        if (ph.vaddr + ph.memsz < ph.vaddr)
            goto bad;
        if ((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
            goto bad;
        if (ph.vaddr % PGSIZE != 0)
            goto bad;
        if (loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz)
            goto bad;
    }
    iunlockput(ip);
    end_op();
    ip = 0;
    ⋮
    ⋮
bad:
    if (pgdir)
        freevm(pgdir);
    if (ip){
```

```
        iunlockput(ip);
        end_op();
    }
    return -1;
}


int loaduvm(pde_t *pgdir, char *addr, struct inode *ip, uint offs
                    uint sz) {
    uint i, pa, n;
    pte_t *pte;

    if ((uint) addr % PGSIZE != 0)
        panic("loaduvm: addr must be page aligned");
    for (i = 0; i < sz; i += PGSIZE) {
        if ((pte = walkpgdir(pgdir, addr+i, 0)) == 0)
            panic("loaduvm: address should exist");
        pa = PTE_ADDR(*pte);
        if (sz - i < PGSIZE)
            n = sz - i;
        else
            n = PGSIZE;
        if (readi(ip, P2V(pa), offset+i, n) != n)
            return -1
    }
    return 0;
}
```

מקום עבור תשובה 2

3. (50 נק') סביבת שאלה זו היא Linux ב־user-mode. הכלים הבאים נתונים לצורך מימוש תוכנת שרת.

```
struct msg_request {
 .....
};
struct msg_response {
 .....
};
int get_request(struct msg_request*);
int process_request(struct msg_request*, struct msg_response*);
int send_response(struct msg_response*);
```

בקשה מלקוח מתקבלת על־ידי הפונקציה get_request. טיפול בבקשה ובניית תשובה מתבצעים על ידי הפונקציה process_request. שליחת תשובה ללקוח מתבצעת על־ידי הפונקציה send_response.

במערכת השרת צריך להיות תהליך ראשי אשר מקבל בקשות בלולאה אינסופית ומעביר את הבקשות להמשך טיפול על־ידי פייפ.

הפונקציה process_request הינה "החלק הכבד" במערכת. יש לשאוף לכך שבכל רגע נתון מספר התהליכים שמבצעים (או פנויים לבצע) את החלק הכבד הוא כמספר המעבדים במחשב (אך לא יותר), הנתון על־ידי הקבוע הבא: #define NCPU 8

לתהליך הראשי מותר לשלוח מידע לתהליך המבצע עיבוד רק אם ידוע שהתהליך פנוי.

ידוע שקיימת שונות גדולה בזמן שלוקח לעבד בקשה (כלומר העיבוד של בקשות מסוימות יהיה מהיר ואילו של אחרות יהיה איטי).

הנחיות:

- מירב הנקודות יינתנו לפתרונות פשוטים ואלגנטיים (ונכונים).
- אין צורך לבדוק אם קריאות המערכת נכשלות.
- בפייפ יש מספיק מקום למבנים הנ"ל.

יש לתת שני מימושים של המערכת, לפי הסעיפים הבאים.

(א) (20 נק') ממשו מערכת כנ"ל כאשר מעבר מידע בין כל התהליכים יתבצע בעזרת פייפ יחיד.

(ב) (30 נק') ממשו מערכת כנ"ל כאשר פייפ מסויים יכול לשמש לתקשורת בין זוג תהליכים בלבד.

```
FLOCK(2)                    Linux Programmer's Manual                   FLOCK(2)

NAME
       flock - apply or remove an advisory lock on an open file

SYNOPSIS
       #include <sys/file.h>

       int flock(int fd, int operation);

DESCRIPTION
       Apply or remove an advisory lock on the open file specified by fd.  The
       argument operation is one of the following:

           LOCK_SH  Place a shared lock.  More than one  process  may  hold  a
                    shared lock for a given file at a given time.

           LOCK_EX  Place  an  exclusive  lock.   Only one process may hold an
                    exclusive lock for a given file at a given time.

           LOCK_UN  Remove an existing lock held by this process.

       A call to flock() may block if an incompatible lock is held by  another
       process.   To  make  a  nonblocking request, include LOCK_NB (by ORing)
       with any of the above operations.

       A single file may not simultaneously have  both  shared  and  exclusive
       locks.

       Locks  created by flock() are associated with an open file table entry.
       This means that duplicate file descriptors (created  by,  for  example,
       fork(2)  or  dup(2)) refer to the same lock, and this lock may be modi-
       fied or released using any of these descriptors.  Furthermore, the lock
       is  released  either  by  an explicit LOCK_UN operation on any of these
       duplicate descriptors, or when all such descriptors have been closed.

       If a process uses open(2) (or similar) to obtain more than one descrip-
       tor  for  the same file, these descriptors are treated independently by
       flock().  An attempt to lock the file using one of these file  descrip-
       tors  may  be  denied  by  a  lock that the calling process has already
       placed via another descriptor.
```

```
WAIT(2)                    Linux Programmer's Manual                    WAIT(2)

NAME
       wait, waitpid, waitid - wait for process to change state

SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *status);

       pid_t waitpid(pid_t pid, int *status, int options);

       int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

```
DESCRIPTION
       All of these system calls are used to wait for state changes in a child
       of  the  calling  process, and obtain information about the child whose
       state has changed.  A state change is considered to be: the child  ter-
       minated; the child was stopped by a signal; or the child was resumed by
       a signal.  In the case of a terminated child, performing a wait  allows
       the  system  to  release  the resources associated with the child; if a
       wait is not performed, then the terminated child remains in a  "zombie"
       state (see NOTES below).

       If  a  child has already changed state, then these calls return immedi-
       ately.  Otherwise they block until either a child changes  state  or  a
       signal  handler interrupts the call (assuming that system calls are not
       automatically restarted using the SA_RESTART flag of sigaction(2)).  In
       the  remainder  of this page, a child whose state has changed and which
       has not yet been waited upon by one of these  system  calls  is  termed
       waitable.
```

```
     WEXITSTATUS(status)
            returns  the  exit  status  of  the child.  This consists of the
            least significant 8 bits of the status argument that  the  child
            specified  in  a  call to exit(3) or _exit(2) or as the argument
            for a return statement in main().  This  macro  should  only  be
            employed if WIFEXITED returned true.
```

```
PIPE(2)                    Linux Programmer's Manual                    PIPE(2)

NAME
       pipe, pipe2 - create pipe

SYNOPSIS
       #include <unistd.h>

       int pipe(int pipefd[2]);

       #define _GNU_SOURCE             /* See feature_test_macros(7) */
       #include <unistd.h>

       int pipe2(int pipefd[2], int flags);

DESCRIPTION
       pipe()  creates  a pipe, a unidirectional data channel that can be used
       for interprocess communication.  The array pipefd is used to return two
       file  descriptors  referring to the ends of the pipe.  pipefd[0] refers
       to the read end of the pipe.  pipefd[1] refers to the write end of  the
       pipe.   Data  written  to  the write end of the pipe is buffered by the
       kernel until it is read from the read end of  the  pipe.   For  further
       details, see pipe(7).

       If  flags is 0, then pipe2() is the same as pipe().  The following val-
       ues can be bitwise ORed in flags to obtain different behavior:

       O_NONBLOCK  Set the O_NONBLOCK file status flag on  the  two  new  open
                   file  descriptions.   Using  this flag saves extra calls to
                   fcntl(2) to achieve the same result.

       O_CLOEXEC   Set the close-on-exec (FD_CLOEXEC) flag on the two new file
                   descriptors.   See  the  description  of  the  same flag in
                   open(2) for reasons why this may be useful.

RETURN VALUE
       On success, zero is returned.  On error, -1 is returned, and  errno  is
       set appropriately.
```

```
SEM_POST(3)                Linux Programmer's Manual                SEM_POST(3)

NAME
       sem_post - unlock a semaphore

SYNOPSIS
       #include <semaphore.h>

       int sem_post(sem_t *sem);

       Link with -lrt or -pthread.

DESCRIPTION
       sem_post()  increments  (unlocks)  the semaphore pointed to by sem.  If
       the semaphore's value consequently  becomes  greater  than  zero,  then
       another  process  or thread blocked in a sem_wait(3) call will be woken
       up and proceed to lock the semaphore.
```

```
NAME
       sem_wait, sem_timedwait, sem_trywait - lock a semaphore

SYNOPSIS
       #include <semaphore.h>

       int sem_wait(sem_t *sem);

       int sem_trywait(sem_t *sem);

       int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);

       Link with -lrt or -pthread.

   Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

       sem_timedwait(): _POSIX_C_SOURCE >= 200112L || _XOPEN_SOURCE >= 600

DESCRIPTION
       sem_wait()  decrements (locks) the semaphore pointed to by sem.  If the
       semaphore's value is greater than zero, then  the  decrement  proceeds,
       and  the function returns, immediately.  If the semaphore currently has
       the value zero, then the call blocks until either it  becomes  possible
       to  perform the decrement (i.e., the semaphore value rises above zero),
       or a signal handler interrupts the call.

       sem_trywait() is the same as sem_wait(), except that if  the  decrement
       cannot  be immediately performed, then call returns an error (errno set
       to EAGAIN) instead of blocking.
```

```
NAME
       sem_getvalue - get the value of a semaphore

SYNOPSIS
       #include <semaphore.h>

       int sem_getvalue(sem_t *sem, int *sval);

       Link with -lrt or -pthread.

DESCRIPTION
       sem_getvalue() places the current value of the semaphore pointed to sem
       into the integer pointed to by sval.
```