מספר זהות:

סמסטר א, מועד א.
תאריך: 5/2/2016
שעה: 0900
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 16 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') בשאלה זו סביבת העבודה הינה קרנל לא ידוע שרץ על מעבד פנטיום־32 (המעבד הרגיל מההרצאות). ה־mmu עובד וניתן להתעלם מיחידת הסגמנטציה. במשתנה הגלובלי pgdir ישנה הכתובת הוירטואלית של טבלת תרגום ש־ cr3 מצביע אליה. בטבלה החיצונית ישנן כניסות לא ווליديות ואפשר להשתמש בהן באופן זמני כרצונכם. נתונות שתי רוטינות. palloc() מחזירה את הכתובת הפיזית של דף שאינו בשימוש. pfree(void *paddr) מחזירה למאגר הדפים שאינם בשימוש את הדף שכתובתו הפיזית paddr.

כיתבו רוטינה שרצה בקרנל וחתימתה:

```
int memcopy(int *ftbl, uint fva, int *ttbl, uint tva, uint len)
```

(א) ftbl הינה כתובת וירטואלית של טבלת תרגום סטנדרטית.

(ב) fva הינה כתובת ויטואלית במובן הטבלה הנתונה על ידי ftbl.

(ג) ttbl הינה כתובת וירטואלית של טבלת תרגום סטנדרטית.

(ד) tva הינה כתובת ויטואלית במובן הטבלה הנתונה על ידי ttbl.

(ה) הרוטינה תעתיק len בתים רציפים וירטואלית החל המכתובת הויטואלית fva במרחב הכתובות הנתון על־ידי ftbl אל רצף הבתים הוירטואלי המתחיל בכתובת tva במרחב כתובות הנתון על־ידי ttbl.

(ו) אם ההעתקה אינה אפשרית בנקודה מסויימת יש להחזיר -1 כערך הפונקציה .

(ז) אם הכל תקין יש להחזיר 0 כערך הפונקציה.

תזכורת:

(א) רוחב השדות בכתובת וירטואלית הם 10 (אינדקס חיצוני), 10 (אינדקס פנימי), 12 (היסט).

(ב) בטבלת תרגום בכל כניסה, 20 הביטים השמאליים הם מספר דף פיזי, ו־12 הימניים הם דגלים. הביט הימני ביותר הוא ביט הווליד.

2. (15 נק') סביבת שאלה זו היא xv6 ב־kernel-mode. נתון הווקטור של מבנים הבא.

```
struct abstract {
   ⋮
};

#define ABSTRACT 100
struct abstract abstract[ABSTRACT];
```

עליכם לכתוב את הפונקציות

```
void abstract_lock(struct abstract *p);
void abstract_unlock(struct abstract *p);
```

עליכם לנעול ולשחרר את המבנה המסופק בארגומנט. הנעילה חייבת להיות עדינה. אפשר להוסיף שדות למבנה. יש להוסיף בדיקות שפיות)אבל הגיוניות, לא לא־שפויות!(כליכולות xv6 לרשותכם.

3. (35 נק') סביבת שאלה זו היא linux ב־user-mode. נתונים בתכנית ההגדרות הבאות:

```
#define LEVEL 3
#define SILVER_NUM 13
```

נתונות וממומשות כבר, הפונקציות הבאות:

מחזירה מספר אקראי בתחום הדרוש `int get_random();`

התהליך הולך "לישון" למשך זמן אקראי `int sleep_random();`

כיתבו תכנית המבצעת את ההתנהגות הבאה: בתחילה ההורה יוצר שני צאצאים (רמה 1). כל אחד מהם יוצר שני צאצאים (רמה 2), וכך הלאה. הצאצאים ברמה LEVEL, שנכנה אותם "צאצאים־עלים" על שום הדמיון לעץ בינרי) לא יוצרים יותר צאצאים.

התהליכים צאצאים־עלים, מבצעים את הפעילות הבאה. כל צאצא, מגריל מספר ע"י קריאה לפונקציה get_random() ובודק אם מספר זה הוא SILVER_NUM. אם המספר הוא SILVER_NUM אזי הצאצא מסתיים. אחרת, הצאצא מבצע קריאה לפונקציה sleep_random, וחוזר חלילה. כל צאצא סופר כמה מספרים הוא הגריל.

כאשר צאצא־עלה מסוים הגריל SILVER_NUM זהו סימן שהוא צריך להסתיים, אבל יחד אתו גם שאר הצאצאים־עלים צריכים להסתיים. כלומר, אם הם בדיוק ישנים, הם יסיימו את שנתם, אבל יפסיקו להגריל ויסתיימו.

שימו לב, שכל תהליך צאצא "יודע" (זוכר) כמה מספרים הוא הגריל. בדרך כלשהי שתחליטו עליה, המידע הזה מגיע לתהליך ההורה, והוא, בסיום מדפיס הודעה בסגנון:

`Total of Random Numbers created is: 154`.

לגבי התהליכים ברמות השונות שאינם בצאצאים־עלים – אין הוראות מיוחדות. כמובן, שבמוקדם או במאוחר גם הם צריכים להסתיים.

הנחיות / הבהרות: כמובן שכל צאצא־עלה יודע כמה מספרים הוא הגריל. עליכם לבנות את המנגנון שבאמצעותו גם ההורה ידע את מה שהוא צריך לדעת. מירב הנקודות יינתנו למנגנון מאוד פשוט וקל למימוש.

(א) הסבירו בקצרה – 2־3 משפטים לכל היותר, איך המידע עובר)כמות המספרים שהוגרלו(.

(ב) הסבירו איך תגרמו לצאצאים־עלים להסתיים.

(ג) ממשו את התכנית.

הערה: אין צורך לכתוב משפטי `#include`. כמו כן, לא צריך לבדוק הצלחה של פונקציות מערכת, כמו `fork` או `open`.

4. (15 נק') סביבת שאלה זו היא linux ב־user-mode. נתונה התוכנית הבאה.

```c
#include <stdlib.h>
#include <stdio.h>

void do_son(){
    puts("hello from son");
}
void do_parent(){
    puts("hello from parent");
}
int main () {
  int pid;
  pid = fork();
  pid = fork();
  if (pid == 0) {
     do_son();
     exit(0);
  }
  else {
     // wait(NULL);
     do_parent();
     exit(0);
  }
  return 0;
}
```

הערות:

(א) הפונקציה puts מוסיפה new-line בסוף המחרוזת.

(ב) הניחו שאין ערבול בין הפלטים של הפונקציה puts. כלומר, הפלט של כל קריאה לפונקציה מודפס בשלמותו (כולל ה־new-line) מבלי שתהליך אחר "מפריע לו".

ענו על השאלות הבאות.

(א) כמה תהליכים נוצרים?

(ב) מהו הפלט של התכנית? (אם אפשרי מספר פלטים, תארו רק אחד מהם):

(ג) הוסיפו לקוד התכנית את שורה 19 (כלומר, בטלו את ההערה). האם תהיה לזה השפעה על הפלט האפשרי? אם כן, מה תהיה ההשפעה? נמקו בקצרה בכל מקרה (כלומר אם כן או אם לאו)

(ד) שינו את שורה 13 כפי שמתואר להלן (ניתנת רק הפונקציה main, הפונקציות האחרות לא השתנו). האם התכנית מתנהגת באופן שונה? הסבירו בקצרה.

```
10 int main () {
11     int pid;
12     pid = fork();
13     pid += fork();
14     if (pid == 0) {
15         do_son();
16         exit(0);
17     }
18     else {
19         wait(NULL);
20         do_parent();
21         exit(0);
22     }
23     return 0;
24 }
```

(ה) תארו פלט אפשרי של התכנית:

```
#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)

struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
  char *name;        // Name of lock.
  struct cpu *cpu;   // The cpu holding the lock.
  uint pcs[10];      // The call stack (an array of program counters)
                     // that locked the lock.
};

void acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  while(xchg(&lk->locked, 1) != 0)
    ;


  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}

// Release the lock.
void release(struct spinlock *lk)
{
  if(!holding(lk))
```

```
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;
  xchg(&lk->locked, 0);

  popcli();
}

void sleep(void *chan, struct spinlock *lk)
{
  if(proc == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }

  proc->chan = chan;
  proc->state = SLEEPING;
  sched();

  proc->chan = 0;

  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}

static void wakeup1(void *chan)
{
  struct proc *p;
```

```
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

void wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

```
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.
```

```
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.
```

```
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_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.
```

```
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.
```