מספר זהות:

סמסטר א, מועד א.
תאריך: 6/2/2016
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.

קוד לא קריא לא יבדק!

יש לענות בשטח המוקצה לכך.

בהצלחה!

1. (35 נק') בשאלה זו סביבת העבודה הינה xv6. ה־mmu שלנו תומך במבנה טבלאות תרגום נוסף על זה שלמדנו. במבנה זה כתובת וירטואלית מתפרשת באופן הבא על יד יחידת הדיפדוף:

31 30 29	21 20	12 11	0
i_0	i_1	i_2	offset

כלומר טבלת התרגום הינה בת שלוש רמות (ולא שתי רמות). כניסה בטבלת התירגום, בכל הרמות, היא בגודל 8 בתים (ולא 4 בתים). גודל דף הוא עדיין 4KB! מעשית, מבנה כל כניסה בטבלת התירגום הוא כמו שלמדנו, רק שנוספו עוד 32 ביטים מאופסים משמאל.

63 32	31 12	11 0
Zeros	Physical page #	flags

עליכם לכתוב את הפונקציה walkpgdir2. על פונקציה זו לעשות עבור המבנה החדש את מה שהפונקציה walkpgdir עושה עבור המבנה שנלמד בהרצאה.

```
#define PDX(va)          (((uint)(va) >> PDXSHIFT) & 0x3FF)
#define PTX(va)          (((uint)(va) >> PTXSHIFT) & 0x3FF)
#define PTXSHIFT         12
#define PDXSHIFT         22
#define PTE_P            0x001
#define PTE_W            0x002
#define PTE_U            0x004
#define PTE_ADDR(pte)    ((uint)(pte) & ~0xFFF)
static pte_t *walkpgdir(pde_t *pgdir, const void *va, int alloc) {
    pte_t *pgtab;
    pde_t *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;
            memset(pgtab, 0, PGSIZE);
            *pde = v2p(pgtab) | PTE_P | PTE_W | PTE_U;
        }
        return &pgtab[PTX(va)];
    }
```

2. (15 נק') סביבת שאלה זו היא xv6. כיתבו פונקציה המביאה את הערך של הארגומנט ה־n של קריאת מערכת, בהנחה שהוא מטיפוס int:

```
int argument(int n, int *val);
```

כלומר אם ב־user-mode מבצעים:

```
pushl $5
pushl $7
subl  $4,%esp
movl  $6,%eax
int   $64
```

אזי בקרנל, בזמן טיפול בקריאת מערכת, ביצוע argument(0,&v) או argument(1,&v) יציב ל־v את הערכים 7 או 5, בהתאמה. הפונקציה מחזירה 0 אם אין בעית אבטחה/חוקיות. אחרת הפונקציה תחזיר 1-.

אין אפשרות להסתמך על רוטינות xv6 אחרות.

```
struct proc {
        uint sz; // Size of process memory (bytes)
        pde_t* pgdir; // Page table
        char *kstack; // Bottom of kernel stack for this process
        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 trapframe {
uint edi;
uint esi;
uint ebp;
uint oesp; // useless & ignored
```

```
 uint ebx;
 uint edx;
 uint ecx;
 uint eax;

 ushort gs;
 ushort padding1;
 ushort fs;
 ushort padding2;
 ushort es;
 ushort padding3;
 ushort ds;
 ushort padding4;
 uint trapno;
 uint err;
 uint eip;
 ushort cs;
 ushort padding5;
 uint eflags;

 uint esp;
 ushort ss;
 ushort padding6;
};

void trap(struct trapframe *tf) {
 if (tf->trapno == 64) {
        if (proc->killed)
                exit();
        proc->tf = tf;
        syscall()
        if (proc->killed)
                exit();
        return;
 }

⋮
```

3. (35 נק') סביבת שאלה זו היא linux ב־user-mode.

כיתבו תכנית המממשת את ההתנהגות הבאה: עם הרצת התוכנית נאמר שאנו ברמה 0 והתוכנית תייצר שני צאצאים. נאמר שצאצאים אלו הם ברמה 1. כל צאצא יוצר שני צאצאים, שיהיו רמה 2, וכך הלאה. הצאצאים ברמה LEVEL, שנכנה אותם "צאצאים־עלים" (על שום הדמיון לעץ בינרי), לא יוצרים צאצאים. LEVEL מוגדר על־ידי

#define LEVEL 4

כל זוג צאצאים־עלים שיש להם הורה משותף ישיר, יכונו LL (עבור Left_Leaf) ו־RL (עבור Right_Leaf).

תפקידו של הצאצא RL הוא להגריל מספרים ולשלוח אותם לצאצא LL כל עוד הצאצא LL קיים. לאחר כל שליחה על הצאצא RL "לישון" פרק זמן אקראי. הגרלת המספרים תתבצע על־ידי קריאה לפונקציה שחתימתה int get_random(). (פונקציה זו נתונה ואין צורך לכתוב אותה). שינה לפרק זמן אקראי מתבצעת על־ידי הפונקציה sleep_random(), שגם היא נתונה.

תפקידו של הצאצא LL הוא לקבל את המספרים מ־RL ולהמשיך לרוץ כל עוד המספר שהתקבל אינו מיוחד.. למזלנו קיימת **תוכנית** (כלומר קובץ elf) במעטפת העבודה שלנו בשם check_numbers. רק תוכנית זו יודעת לבדוק אם מספר הוא מיוחד.

התכנית check_numbers מתנהגת באופן הבא. היא קוראת קוראת מספר בינרי באורך 4 בתים מהקלט הסטנדרטי, ובודקת אותו. כל עוד המספר אינו מיוחד התכנית ממשיכה לקרוא מספר מהקלט ולבדוק אותו. במידה והמספר מיוחד, התוכנית מדפיסה לפלט הסטנדרטי הודעה כגון: found special number 12345.... Exiting ולאחר מכן היא מסתיימת.

קיימת מגבלה נוספת במערכת. התכנית check_number היא תכנית עתירת CPU, ואם כל התהליכים עלים יעבדו במקביל אזי המחשב "יחנק" (כלומר, יהיה עמוס מעבר למה שאנחנו רוצים). לפיכך לכל היותר NCPU תהליכים LL רשאים לרוץ בו זמנית. (כאשר NCPU מוגדר על ידי

#define NCPU 4

הנחיות:

- מירב הנקודות בשאלה הזאת ינתנו לפתרונות פשוטים ואלגנטיים.
- אין ל התייחס בתשובה לאפשרויות שקריאות מערכת כגון open, pipe, fork וכו' נכשלות.
- אין צורך להוסיף #include's.
- תשובה בכתב לא קריא – לא תיבדק!
- לגבי הקריאה exec התייחסו לחתימה של xv6:

int exec(char* prog_name, char **argv);

(א) תארו בקצרה (2 שורות לכל היותר) איזה מידע עובר בין כל זוג תהליכים LL ו־RL, וכיצד הוא עובר.

(ב) תארו בקצרה (2 שורות לכל היותר) כיצד צאצא RL (הצאצא מגריל המספרים) מסתיים.

(ג) תארו בקצרה (3־2 משפטים לכל היותר) איך מבטיחים שלא ירוצו יותר מ־NCPU תוכניות check_numbers בו זמנית.

(ד) ממשו את התכנית.

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

```
10  int glb = 0;
11  int main()
12  {
13      static int stat = 0;
14      int pid;
15
16      if (fork() && fork() && (pid = fork()) > 0)
17      {
18          stat++;
19          glb++;
20          printf("msg #1: pid=%d stat=%d glb=%d\n", pid, stat, glb);
21      }
22      stat++;
23      glb++;
24      printf("msg #2: pid=%d stat= %d glb=%d\n", getpid(), stat, glb);
25    return 0;
26  }
```

לצורך התשובה ניתן להניח שההודעות שמודפסות אינן מתערבבות (כלומר כל הודעה מודפסת בשלמותה). הניחו שה־pid של התהליך הראשון (תהליך ההורה) הוא 100, וכל התהליכים שנוצרים בהמשך מקבלים מספרים עוקבים.

(א) כמה תהליכים נוצרים בסה"כ?

(ב) תארו פלט אפשרי של התכנית:

(ג) החליפו את שורה 16 בשורה הבאה:

$$\text{if } (\text{fork}() == \text{fork}() \ \&\& \ (\text{pid} = \text{fork}()) > 0)$$

כמה תהליכים ייווצרו הפעם?

```
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.
```

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

NAME
       kill - send signal to a process

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

       int kill(pid_t pid, int sig);

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

       kill(): _POSIX_C_SOURCE >= 1 || _XOPEN_SOURCE || _POSIX_SOURCE

DESCRIPTION
       The  kill()  system  call can be used to send any signal to any process
       group or process.

       If pid is positive, then signal sig is sent to the process with the  ID
       specified by pid.
```

to kill a process with pid==100 you need to do:

```
kill(100, Term);     // Term is the signal to terminate the process
```