מספר זהות:

סמסטר ב, מועד ב.
תאריך: 25/7/2017
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 20 עמודים (כולל עמוד זה).

תשובות צריכות לכלול הסבר.

קוד לא קריא לא יבדק!

יש לענות בשטח המוקצה לכך.

בהצלחה!

1. (30 נק') סביבת שאלה זו היא xv6 ב־kernel mode. עליכם לממש את שלוש קריאות המערכת הבאות:

```
int event_set(int num);
int event_clear(int num);
int event_wait(int num);
```

קריאות אלו נועדו לממש מערכת event־ים. המערכת צריכה לתפקד באופן הבא.

מספר ה־event־ים במערכת הוא MAX_EVENTS. כל event מזוהה על ידי מספר בתחום 0 עד MAX_EVENT. event נמצא באחד משני מצבים: set או clear. הקריאות event_set ו־event_clear קובעות את מצב האירוע המזוהה על־ידי הארגומנט למצב set או clear, בהתאמה. שתי הקריאות מחזירות, בהנחה שאין שגיאה, את המצב בו היה ה־event לפני שנקבע מצבו החדש. (קריאות עוקבות ל־event_set או ל־event_clear הינן חוקיות.)

תהליך מחכה ש־event מסוים יהיה במצב set על ידי קריאה ל־event_wait כשהארגומנט מזהה את ה־event המדובר. אם ה־event כבר במצב set הקריאה חוזרת מיד. אם ה־event במצב clear הקריאה תמתין עד שה־event יעבור למצב set ואז תחזור. בכל מקרה, עם החזרה, על ה־event לחזור למצב clear.

אם כמה תהליכים מחכים ביחד לכך ש־event יעבור למצב set אזי רק אחד מהם יתעורר כתוצאה ממעבר ה־event למצב set.

כל שנויי המצב חייבים להיות אטומיים.

אין צורך לדאוג להגינות, או לדאוג מהרעבה.

אין צורך לכתוב את קוד שליפת הארגומנטים מ־user miode.

```
acquire(struct *spinlock lk);
release(struct *spinlock lk);
sleep(void *chan, struct *spinlock lk);
```

2. (20 נק') סביבת שאלה זו היא xv6 במצב קרנל. כיתבו פונקציה שחתימתה

int loadMem(int reg, int val);

משתמשים בפונקציה זו בקוד המטפל בקריאות מערכת. רוטינה זו תציב את הערך val לכתובת בזיכרון אליה מצביע האוגר reg ב־user-mode. את reg יש לפרש לפי הטבלה הבאה:

reg	means	reg	means
0	eax	4	esi
1	ecx	5	edi
2	edx	6	ebx
3	ebx	7	esp

לדוגמא, אם ב־user-mode מבצעים:

```
arg:   .long
   ⋮
   ⋮
   movl   $arg,%ecx
   subl   $4,%esp
   movl   $6,%eax
   int    $64
```

אזי בקרנל, בזמן טיפול בקריאת מערכת, ביצוע store(1,17) יציב ל־arg את הערך 17.

אין אפשרות להסתמך על רוטינות xv6 אחרות.

אלגנטיות הפתרון תשפיע על הציון.

```
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. (25 נק') סביבת שאלה זו היא Linux ב־user-mode. בקובץ common.h נתונות ההגדרות הבאות:

```
#define GOOD      1
#define BAD       2
#define ERROR     3
#define TERMINATE 4

int read_item_quick(int fd);
int read_item_slow(int fd, int* num_of_bytes_read);
```

נתונות הפונקציות read_item_quick ו־read_item_slow. שתי הפונקציות הן כמעט זהות. ההבדלים ביניהן נתונים בסעיפים (א) ו־(ב) בהמשך. הפונקציות קוראות נתונים באורך לא ידוע דרך ה־file descritpr שנתון בארגומנט הראשון fd. לאחר הקריאה, הפונקציות מבצעות על הנתונים עיבוד כלשהו, ומחזירות את אחד מהערכים GOOD, BAD, ERROR או TERMINATE. שני ההבדלים בין הפונקציות הם:

(א) הפונקציה read_item_slow מחזירה, באמצעות הארגומנט השני שלה, כמה בתים נקראו מהקובץ.

(ב) הפונקציה read_item_slow היא אטית מאוד ביחס לפונקציה read_item_quick, ולכן, ככל שניתן נעדיף לקרוא לפונקציה read_item_quick.

עליכם לכתוב את התכניות prog1 ו־prog2. התוכנית prog1 מקבלת ארגומנט יחיד, שם של קובץ שממנו יש לקבל קלט. ידוע שקובץ הקלט גדול מאוד. התנהגות התוכנית היא כלהלן:

התוכנית מבצעת בלולאה אינסופית את הפעולות הבאות. אחת מהפונקציות הנ"ל נקראת. אם הערך המוחזר מהפונקציה הוא GOOD או BAD אזי התוכנית מדפיסה לפלט הסטנדרטי את השורה "GOOD" או "BAD", בהתאמה. אם הערך המוחזר הוא TERMINATE אזי התכנית מסתיימת. אם הערך המוחזר הוא ERROR משמע נוצר מצב שגיאה כלשהו. במקרה כזה, צריך להפעיל את התוכנית prog2, שתמשיך לבצע את אותה המשימה **בדיוק(!)** – כלומר־ לולאת while(1) קריאות לפונקציות הנ"ל וטיפול זהה בערכי החזרה כמו prog1 בשנוי אחד. התכנית המורצת תחל לקרוא מהקובץ מאותו מקום בדיוק בו סיימה לקרוא התכנית בה אירעה השגיאה!

אין צורך לכתוב הוראות #include וכמו כן אין צורך לבדוק מצבי שגיאה לאחר קריאות כמו fork() וכו'.

4. (25 נק') סביבת שאלה זו היא Linux ב־user-mode. במבחן ניתנו ההגדרות הבאות:

```
01 #define N 5
02 struct message {
03     int pids[N];
04     int temp;
05 };
```

הנבחנים התבקשו לכתוב תוכנית אשר תענה על הדרישות הבאות.

תהליך הורה מייצר צינור (pipe) ולאחר מכן מייצר N צאצאים. המטרה היא שצאצא אחד ידפיס את ה־pid־ים של כל הצאצאים ובאופן הבא.

כל צאצא, פרט לראשון, קורא מהצינור את המבנה message, ומוסיף למקום פנוי במערך pids את ה־pid שלו.

כל צאצא, פרט לאחרון, כותב לצינור את המבנה (לאחר שה־pid הוסף) ומסיים.

הצאצא הראשון אינו קורא מהצינור אלא רק כותב אליו את המבנה, לאחר שהציב את ה־pid שלו למקום פנוי בווקטור, ומסיים.

הצאצא האחרון אינו כותב לצינור אלא קורא לרוטינה print_message שתבצע את ההדפסה המבוקשת, ומסיים.

אין לקבוע מראש באיזה סדר הצאצאים יקראו מהצינור או מי יהיה הראשון.

אין לקבוע מראש באיזה מקום במערך כל צאצא משתמש.

אחת הנבחנות כתבה את התכנית הבאה:

```
09 int pip[2];
10 int main() {
11     pipe(pip);
12     for (int i = 0; i < N ; i++ )
13         if (fork() == 0)
14             do_child();
15         return 0;
16 }
17
18 void do_child() {
19     struct message msg;
20     read(pip[0], &msg, sizeof(struct message));
21     msg.pids[msg.temp++] = getpid();
```

```
22      write(pip[1], &msg, sizeof(msg));
23      if (msg.temp == N)
24          print_msg(&msg);
25      exit(0);
26  }

27  void print_msg(struct message *msg) {
28      for (int i = 0; i < N; i++)
29          printf("(%d) pid=%d\n", i, msg->pids[i]);
30  }
```

בסעיפים הבאים:

אין לשנות את החתימה של הפונקציה do_child().

אין להוסיף פונקציות לקוד הנתון.

(א) בסעיפים הבאים סמנו בעיגול נכון או לא־נכון, או השלימו את התשובה.

i. נכון / לא־נכון הצאצא האחרון כותב לצינור ואין מי שיקרא את המידע. זה עלול ליצור בעיה.

ii. נכון / לא נכון. ההורה אינו סוגר את הצינור, ולכן עלולה להיווצר בעיה. נמקו בקצרה:

_____

_____

iii. נכון / לא־נכון. בתכנית זו מן הראוי להשתמש ב־flock בתור מנגנון סנכרון.

(ב) בסעיף זה המטרה היא סינכרון פעולות התכנית, תוך שינויים מינימליים. בפרט, יש רק להוסיף קוד, ולא לשנות קוד קיים. מותר להכניס קוד קיים לתוך ה־then או ה־else של תנאי חדש.

אם יש תוספת של משתנים גלובליים, כיתבו כאן את ההכרזות:

_____

_____

_____

_____

כיתבו כאן את הפונקציה main() כולל התוספות:

_____

_____

כיתבן כאן את הפונקציה $\text{do\_child}()$ כולל התוספות:

```
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.
```

```
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.
```

NAME top

_llseek - reposition read/write file offset

SYNOPSIS top

```
#include <sys/types.h>
#include <unistd.h>

int _llseek(unsigned int fd, unsigned long offset_high,
            unsigned long offset_low, loff_t *result,
            unsigned int whence);
```

Note: There is no glibc wrapper for this system call; see NOTES.

DESCRIPTION top

The **_llseek**() function repositions the offset of the open file description associated with the file descriptor *fd* to *(offset_high<<32) | offset_low* bytes relative to the beginning of the file, the current file offset, or the end of the file, depending on whether *whence* is **SEEK_SET**, **SEEK_CUR**, or **SEEK_END**, respectively. It returns the resulting file position in the argument *result*.

This system call exists on various 32-bit platforms to support seeking to large file offsets.

RETURN VALUE top

Upon successful completion, **_llseek**() returns 0. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.