מספר זהות:

סמסטר ב, מועד ב.
תאריך: 13/7/2016
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 15 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. (55 נק') סביבת שאלה זו היא xv6 ב־kernel-mode. ממשו את קריאות המערכת הבאות:

```
int sys_peek(int pid, int len, char *buf, int addr);
int sys_peekwho(int len, char *buf);
int sys_peekack(int flag);
```

מטרת קריאות אלו היא לאפשר לתהליך אחד לעיין בתכני הזיכרון של תהליך אחר תחת בקרה.

תהליך שקורא ל־sys_peek מבקש לקרוא len בתים מכתובת addr במרחב הזיכרון של תהליך שמספרו המזהה pid. (התהליך pid נקרא תהליך היעד.) התהליך שמעוניין לקרוא ניכנס להמתנה עד שתהליך היעד מאשר או דוחה את הפעולה. באם תהליך היעד דוחה את הפעולה, אזי לא תתבצעה המשימה, וערך החזרה מ־sys_peek יהיה שלילי. באם תהליך היעד מאשר את הפעולה, תתבצע ההעתקה המבוקשת וערך החזרה מ־sys_peek יהיה אפס.

תהליך יעד קורא ל־sys_peekwho כדי לדעת מי מעוניין לקרוא מהמרחב שלו. ערך החזרה הוא ה־pid של תהליך המעוניין לקרוא. אם יש כמה מעוניינים מחזירים את המזהה של אחד מהם. אם אף אחד לא מעוניין, sys_peekwho נכנס להשהיה עד שמישהוא כן מעוניין.

כאשר ידוע על תהליך שמעוניין לקרוא ניתן לאשר את הפעולה על ידי קריאה ל־sys_peekack(1) או לדחות את הפעולה על ידי קריאה ל־sys_peekack(0).

תהליך אינו יכול לבקש לקרוא מעצמו!

אסור בהחלט לשמור העתקים של מרחבי כתובות בקרנל. (כיון שיתכן ביזבוז זיכרון עצום ומיותר!)

ניתן להוסיף שדות למבנה proc.

אלגנטיות תילקח בחשבון בניקוד.

2. (30 נק') נתונים בתכנית ההגדרות הבאות:

```
#define N_CHILDREN     4
#define GOOD_MSG 0
#define BAD_MSG  1
```

מוגדר גם המבנה struct msg. (הקוד שעליכם לכתוב איננו מתעניין במבנה הפנימי של ה־struct הזה.)

```
struct msg {
   .......
   .......
};
```

נתונות וממומשות כבר, הפונקציות הבאות:

```
struct *msg create_msg();
int write_msg(struct *msg);
int read_msg(struct *msg);
int check_msg(struct *msg);
```

כיתבו תכנית המבצעת את ההתנהגות הבאה: התהליך הראשי יוצר N_CHILDREN תהליכים – צאצאים ישירים שלו, שנכנה אותם CREATORS כיוון שתפקידם לייצר הודעות. כל אחד מה־CREATORS האלו, יוצר תהליך בן שנכנה אותו CHECKER כי תפקידו לבדוק הודעות.

ה־CREATOR יוצר הודעות על ידי קריאה לפונקציה create_msg. את ההודעות שהוא יוצר הוא מעביר לתהליך ה־CHECKER שלו, ע"י קריאה לפונקציה write_msg המקבלת את ההודעה כארגומנט. הפונקציה הזאת מבצעת עיבוד או תוספת כלשהיא להודעה, ולאחר מכן, היא כותבת את ההודעה לפלט הסטנדרטי.

התהליך CHECKER מקבל הודעות ע"י קריאה לפונקציה read_msg. הפונקציה הזאת קוראת הודעה מהקלט הסטנדרטי, ומחזירה את ההודעה באמצעות הארגומנט msg. בדיקת תקינות ההודעה מתבצעת על ידי תהליך ה־CHECKER באמצעות קריאה לפונקציה check_msg שמחזירה את הערך GOOD_MSG או BAD_MSG. בנוסף, אם זאת הודעה BAD_MSG אזי הפונקציה check_msg מדפיסה ל־standard-error את המחרוזת "ERROR!". אנו רוצים למנוע מהמחרוזות "ERROR!" להופיע על המסך (או למה שהוא ה־standard-error של התכנית).

שימו לב, אין ביכולתנו לשנות את הפונקציה check_msg. תהליך CHECKER מסתיים לאחר שהתקבלו שלוש הודעות BAD_MSG, ויחד איתו מסתיים גם תהליך האב (ה־CREATOR) שלו. ה־CREATOR מעביר בדרך כלשהיא לתהליך הראשי כמה הודעות הוא יצר בסה"כ. אפשר להניח שמספר זה קטן מ־100. לאחר שכל התהליכים צאצאים הסתיימו, התהליך הראשי מדפיס לפלט הסטנדרטי הודעה בסגנון:

```
Total Number of Messages is:  <number>
```

ואז הוא מסתיים.

מירב הנקודות בשאלה זאת ינתנו לקוד נכון לוגית ושימוש במנגנונים שמותאמים לבעיה הנתונה. גם אלגנטיות וקריאות הקוד נלקחים בחשבון במתן הציון.

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

3. (15 נק') נתונה התכנית הבאה:

```
8   #define N 3
9   int main()
10  {
11    int i, my_pipe[2];
12    char str[20] = {0};
13    for (i=0; i< N; i++)
14    {
15      pipe(my_pipe);
16      if (fork()) {
17        break;
18      }
19    }
20
21    if (i)
22    {
23      close(my_pipe[0]);
24      sprintf(str, "Hello from %d\n", i);
25      write(my_pipe[1], str, strlen(str));
26      exit(0);
27    }
28    else
29    {
30      printf("this is %d pid=%d\n", i, getpid());
31      while(1)
32      {
33        int n = read(my_pipe[0],str, 20);
34        if (n==0) break;
35        write(1, str, strlen(str));
36      }
37    }
38  }
```

הניחו שמספר ה־pid של התהליך הראשי שנוצר הוא 100 , וכל תהליך שנוצר לאחר מכן מקבל מספר עוקב, כלומר 101, 102,....

ענו על השאלות המופיעות בעמוד הבא.

(א) כמה תהליכים נוצרים בסה"כ (כולל התהליך הראשי)?

(ב) התכנית אינה מסתיימת (כלומר, כשמריצים אותה מתוך ה־Shell היא לא מסתיימת).
מה הסיבה לכך? הצע תיקון מינימלי שיגרום לתכנית להסתיים.

(ג) מה הפלט שנוצר מהתכנית לאחר התיקון שהצעת בסעיף ב'?

```
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
  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)
};
```

```
#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)
```

```
int pipewrite(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  for(i = 0; i < n; i++){
    while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
      if(p->readopen == 0 || proc->killed){
        release(&p->lock);
        return -1;
```

```
      }
      wakeup(&p->nread);
      sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
    }
    p->data[p->nwrite++ % PIPESIZE] = addr[i];
  }
  wakeup(&p->nread);  //DOC: pipewrite-wakeup1
  release(&p->lock);
  return n;
}

int piperead(struct pipe *p, char *addr, int n)
{
  int i;

  acquire(&p->lock);
  while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
    if(proc->killed){
      release(&p->lock);
      return -1;
    }
    sleep(&p->nread, &p->lock); //DOC: piperead-sleep
  }
  for(i = 0; i < n; i++){  //DOC: piperead-copy
    if(p->nread == p->nwrite)
      break;
    addr[i] = p->data[p->nread++ % PIPESIZE];
  }
  wakeup(&p->nwrite);  //DOC: piperead-wakeup
  release(&p->lock);
  return i;
}
```

```
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.
```

```
   The value of options is an OR of zero or more of the following constants:

   WNOHANG     return immediately if no child has exited.
```

```
       If WNOHANG was specified in options and there were no children in a waitable
       state, then waitid() returns 0 immediately and the state  of  the  siginfo_t
       structure pointed to by infop is unspecified.  To distinguish this case from
       that where a child was in a waitable state, zero out the si_pid field before
       the call and check for a nonzero value in this field after the call returns.

RETURN VALUE
       wait():  on  success,  returns  the  process  ID of the terminated child; on
       error, -1 is returned.
```

```
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.
```