19

האקדמית
תל אביב-יפו

סמסטר ב, מועד א.
תאריך: 21/6/2016
שעה: 0900
משך הבחינה: 3 שעות.
חומר עזר: אסור

בחינה בקורס: **מבוא למערכות הפעלה**

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

55	1
20	2
15	3
90	

הנחיות:
טופס הבחינה כולל 14 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_send(int pid, int len, char *msg);
int sys_receive(int len, char *buf);
```

תהליך משתמש בקריאה sys_send כדי לשלוח הודעה לתהליך שמספרו המזהה pid. ההודעה היא רצף בתים מכתובת msg באורך len בתים. שגרת המערכת חוזרת אל הקורא רק לאחר שההודעה הגיעה ליעדה, כלומר בתהליך היעד sys_receive הביא את ההודעה.

תהליך משתמש בקריאת המערכת sys_receive כדי לקבל הודעה שנשלחה אליו. במידה ולא התקבלה הודעה, הקריאה ממתינה עד שתתקבל הודעה.

אם אורך המאגר המקבל קצר מאורך הודעה שנשלחה, תתקבל הודעה קצוצה, וההודעה תחשב כאילו התקבלה.

ערך החזרה הוא אורך ההודעה שנשלחה (האורך המקורי, במידה והיה קיצוץ.)

הודעה היא "אטומית". כלומר, אם יש כמה שליחות של הודעות במקביל לאותו תהליך לא יווצר עירבוב בין תכני ההודעות, וכל הודעה תגיע בנפרד ל־sys_receive.

תהליך אינו יכול לשלוח הודעות לעצמו!

אסור בהחלט לשמור את ההודעות בקרנל. (כיון שיתכן ביזבוז זיכרון עצום ומיותר!)

ניתן להוסיף שדות למבנה proc.

אלגנטיות תילקח בחשבון בניקוד.

רעיון כללי: נוסיף שדות ל Proc

int sendtoId;
int recived;
uint [illegible];
~~[illegible]~~;
int len;

sys_send ונעשה תחילה בדיקות תקינות כמו אי שליחה עצמית או קיום ה pid שאליו אנו רוצים לשלוח במידה (ומצא) תהליך זה נעדכן את השדות הרלוונטים שהוספנו ל Proc, ~~[illegible]~~ ~~[illegible]~~ ונעיר את התהליך שאמור לקבל, ואז נישן עד את ההודעה תקבל ב Recive.

sys_recive : ימצא תהליך אחר וזה לשלוח אליו הודעה (ואם אין כזה) ונכנס ל sleep מאחר שכן יתנגש שוב, זה יורד עד שימצא הודעה.
ברגע שמצא הוא ~~[illegible]~~ ומעתיק את תוכן ההודעה ע"י copyout, ויעדכן את ה Recive של ה proc ויעיר אותו.

```
int SYS_Send(int Pid, int len, char *msg)
{
    int found=0;

    if (Pid==Proc->Pid)
        return -1;

    acquire(&Ptable.lock);

    for(p=Ptable.Proc; p<Ptable.Proc[NProc]; p++)
    {
        if (Pid == p->Pid)
        {
            found=1;
            break;
        }
    }

    if (!found)
    {
        Realease(&Ptable.lock);
        return -1;
    }

    Proc->sendtoId = Pid;
    Proc->recived = 0;
    Proc->len = len;
    Proc->Mva = msg;
    ~~Pwrite(Proc->sspte, &msg, len);~~
    ~~wakeup(p);~~

    while (Proc->recived ==0)
    {
        wakeUP(p);

        Sleep(Proc, &Ptable.lock);

        if (Proc->killed)
        {
            Realeased &Ptable.lock);
            return -1;
        }
    }

    Release(&Ptable.lock);

    return Proc->recived;
}
```

```
int sys_recive(int len, char * buf)
{
 int found = 0, res = 0;
 acquire(&ptable.lock);

 again:

   for( p = ptable.proc; p < ptable.proc[NProc]; p++)
   {
     if (p->sendtoId == proc->id )
     {
       found = 1;
       break;
     }
   }

   if(!found)
   {
     sleep(proc , &ptable.lock);
     if (proc->killed )
     {
       Release(&ptable.lock);
       return -1;
     }
     goto  again;
   }

  if ( p->received != 0 )
    panic("How can it be?");
        my_mem_copy (p->pgdir, p->va, len, buf);
  p->received = ~~[illegible]~~;
  res = p->len;
  wakeUP(p);
  Release(&ptable.lock);

  return res;
}
```

→ [illegible]

```
int my_mem_copy(int *tbl, uint fva, uint tva, int len)
{   uint *cva;
    char *c;
    ~~uint tva~~

    for(i=0; i<len; i++)
    {
      if(tva2kva(tbl, fva+i, &cva)<0)
        return 0;
        (char*)
      *c = cva;

      if(tva2kva(Proc->pgdr, tva+i, &cva)<0)
        return 0;

      *cva = c;
    }

    return len;
}

int tva2kva(int *tbl, uint va, uint* cva)
{   int i0 = (va<<22) & 1023;
    int i1 = (va<<12) & 1023;
    if((tbl[i0]&1)==0)
      return -1;
    v1 = P2V(tbl[i0]&-4096);
    ~~i1 = (va<<12) & 1023;~~
    if((v1[i1]&1)==0)
      return -1
    *cva = P2V(v1[i1]&-4096) | (va&4095);

    return 0;
}
```

2. (30 נק') סביבת שאלה זו היא linux ב־user-mode. נתונות ההגדרות הבאות:

```
#define LEVEL     4
#define GOOD_MSG  0
#define BAD_MSG   1
#define MAX_MSGS  5

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

נתונות וממומשות הפונקציות הבאות:

```
struct *msg create_msg();
void pass_msg(struct *msg, int fd);
int check_msg(struct *msg);
```

יש לכתוב תכנית המבצעת את ההתנהגות הבאה: תהליך האב (נכנה אותו P0) יוצר תהליך צאצא (אותו נכנה P1). התהליך P1 יוצר צאצא P2. זה יוצר צאצא P3 וכך הלאה LEVEL פעמים עד התהליך P-Level (במקרה שלנו: P4). את התהליך האחרון שנוצר בשרשרת זאת נכנה "צאצא־עלה".

התהליך P1 יוצר בתוך לולאת while הודעות ע"י קריאה ל־create_msg() כל הודעה צריכה לעבור לתהליך P2, וממנו לתהליך P3 עד לצאצא־עלה. העברת הודעות מתבצעת ע"י קריאה לפונקציה pass_msg, שמקבלת את ההודעה כארגומנט. פונקציה זאת מבצעת עיבוד או תוספת כלשהי להודעה, ולאחר מכן כותבת את ההודעה ל־file descriptor שהועבר כארגומנט ב־fd. כשהצאצא־עלה מקבל את ההודעה הוא בודק אותה ע"י קריאה ל־check_msg. אם ההודעה תקינה (ערך מוחזר GOOD_MSG) אזי הצאצא מדפיס לפלט הסטנדרטי "Got Good Message", ובכך מסתיים הטיפול בהודעה זו. אם מוחזר הערך BAD_MSG אזי מודפס לפלט הסטנדרטי "Got BAD Message", וכל התהליכים צריכים להסתיים. (במקרה כזה מתעלמים מההודעות האחרות שנמצאות בשלב כלשהו של טיפול.) קיימת מגבלה על מספר ההודעות שיכולות להיות "בטיפול" בו זמנית (ע"י כלל התהליכים) – אסור שמספרן יעלה על ערך הקבוע MAX_MSGS.

(א) תארו בקצרה (2-3 משפטים לכל היותר) כיצד ההודעות עוברות בין התהליכים (במימוש הנבחר). ההודעות עוברות בין התהליכים ע"י PIPES

לא תהליך יקבל מהבנים ואת ומתיב שב כבה עד שיגיע לתהליך

הראשון שהוא רק יקבל מבנים

(ב) תארו בקצרה (2-3 משפטים לכל היותר) כיצד התהליכים מסתיימים כאשר התברר שהודעה מסוימת היא BAD_MSG.

תגדיר Sculpture מאותחל ל-0 ונשתר בהודעה מסוימת

(כי אז) BAD_MSG נבלע sem_signal לא תהליך ובלע schedule

ויגלה שהשרת בלע מ-0 ויבלע exit.

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

```
int main()
{
  int i;
  sem_t semMsg = sem_open("semNum", O_CREATE, MAX_MSGS);
  sem_t semf = sem_open("semf", O_CREATE, 0);
  int myPipe[2];
  pipe(myPipe);

  for(i=0; i<LEVEL; i++)
  {
    if (fork() && i==0)
    {
      ~~wait(NULL);~~ while (sem_get_value(semf)==0)
      ~~exit(0);~~ exit(0);
    }
    if (i==0)
      do_L1_child(semf, semMsg, myPipe);
    else
      do_Lchild(semf, myPipe);
  }
  do_Leaf_chd(semf, semMsg, myPipe);
}

void do_L1_child(sem_t semf, sem_t semMsg, int* myPipe)
{
  struct msg * m;

  while(1)
  {
    sem_wait(semMsg);
    m= create_msg();
    pass_msg(m, myPipe[1]);
    if(sem_get_value(semf) > 1
      exit(1);
  }
}
```

sem_unlink(semMsg);
sem_unlink(semf);
close(myPipe[0]);
close(myPipe[1]);

אבל הוא
בדר לא
ומשה בנים...

— לוגיקה לא נכונה של יצירת תהליכים

— כולם משתמשים ב - שקוף
ומשולב בדרכים נפרד
לכל צאצא תהליכים
אבא-בן

```
void do_Lchild(sem_t semf, int *mypipe)
{   struct msg *m;
    while(1)
    {
      if(sem_get_value(semf)>0)
        exit(1);
      read(mypipe[0], &m, sizeof(struct msg));
      pass_msg(m, mypipe[1]);
    }
}

void do_Leaf_child(sem_t semf, sem_t semMsg, int *mypipe)
{   struct msg *m;
    while(1)
    {
      read(mypipe[0], &m, sizeof(struct msg));
      if( check_msg(m) == Good_MSG )
      {
        sem_signal(semMsg);
        Printf("Got Good message");
      }
      else if ( check_msg(m) == BAd_MSG )
      {
        Printf("Got Bad message");
      sem_signal(semf);
      exit(1);
      }
}
```

sem_post ✓

(20)

3. (15 נק') שאלה זו מתייחסת למערכת דמוית לינוקס ב־user-mode, בהבדל אחד – הפונקציה fork() מחזירה 0 במקרה של הצלחה – גם לתהליך האב וגם לתהליך הבן. (היא מחזירה 1– במקרה של אי־הצלחה, אבל אין צורך לבדוק זאת.) פרט לכך, עומדות לרשותנו כל הפונקציות של לינוקס או C כפי שאנחנו מכירים.

מוגדר הקבוע N, למשל:

```
#define N 5
```

כיתבו תכנית בה התהליך הראשי (נכנה אותו P0) יוצר תהליך צאצא P1, תהליך זה יוצר צאצא P2 וכן הלאה, שרשרת של N תהליכים, כאשר P5 (במקרה שלנו) הוא התהליך האחרון. הצאצא האחרון (P5) מדפיס P5 done ומסתיים. לאחר שהוא הסתיים, ההורה שלו, מדפיס P4 done ומסתיים, וכן הלאה. בסה"כ הפלט של התכנית נראה כך

```
:
p5 done
P4 done
P3 done
P1 done
P0 done
```

```
int main()
{
  int Pid, i=0 ;
  Pid= getPid();
  for(i=0;i<N;i++)
  {
    fork();
    if(Pid==getPid())
    {
      wait(NULL);
      printf("P%d done",i);
      exit(1);
    }
    else{
      Pid = getPid();
    }
  }
  printf("P%d Done",i);
  exit(1);
}
```

15

```
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;
}
```

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 terminated; 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 immediately. 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.

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 values 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.

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.