

גרעין מערכת ההפעלה

גרעין מערכת ההפעלה היא רכיב תוכנה (מבנה נתונים + פקודות) הממש את אותו רובד השולט בחומרה של המחשב בצורה כזו, שהיא מגדירה את המכונה הוירטואלית המסתיר את החומרה מהמשתמש והאפליקציה. בדרך כלל הגרעין הוא קוד בינארי הנטען לזכרון ומקבל שליטה עם עלית מערכת ונשאר כאן, צריך קצת להתאמץ בשביל להבחין בו.

באופן ציורי נהוג לחלק את הגרעין לסידרה של קטגוריות למשל:

- מנהל התהליכים.
- מנהל הפסקות.
- מנהל הזכרון.
- מנהל הדיסקים.
- מנהל הקלט / פלט.

הרכיב הראשון והעיקרי שנראה ב-XINU יהיה מנהל התהליכים.

מנהל התהליכים

תפקידו העיקרי של מנהל התהליכים הם ליצג את התהליכים במערכת, לתמוך ביצירה ושחרור של המשאב שקרוי תהליך, לתמוך בהפעלה ועצירה של תהליכים, לממש את מנגנון החלפת התהליכים, ורכיב אופציונלי של תמיכה בשירותי תאום בין תהליכים.

כמו כל רכיב תוכנה, מנהל התהליכים הוא שילוב של מבנה נתונים ופעולות עליהם.

החלק הראשון שנראה מנגנון תורי התהליכים.

תורי תהליכים

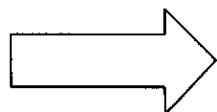
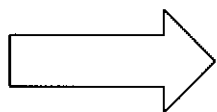
ב-xinu ישנם מצבים שבהם תהליכים ממתינים יחסית לתהליכים אחרים. לפיכך יש צורך במבנה נתונים למימוש תורים שבו מיוצגים התהליכים. תהליך שהיצוג שלו נמצא בתור נמצא באיזה תור בכל מקרה נמצא בהמתנה מסוג כלשהוא. סוגי ההמתנה הם:

תרשים מיקומו של הגרעין במערכת ההפעלה

תהליכי ישומים

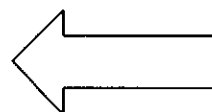
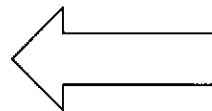
תהליכי מערכת

קריאות מערכת



פסיקות

קריאות מערכת



פסיקות



- המתנה להתפנות ה-CPU.
- המתנה לסמפור.
- הרדמה לזמן קצוב.

לצורך תמיכה בהמתנות המערכת צריכה לתמוך בשלושה סוגי תורים:

- לצורך המתנה ל-CPU - תור עדפיות.
- לצורך המתנה לסמפור - תור FIFO.
- לצורך הרדמה - תור הפרשים.

מימוש

ניתן לממש את כל התורים במבנה נתונים אחד. זה לא לחלוטין הכרחי אבל במובן זה או אחר זה נוח שיש מבנה נתונים אחד לכולם.

כל התורים ימומשו ברשימה מקושרת דו כיוונית. יהיה לנו מערך אחד, $q[]$, אשר מבנה הצמתים שלו מוגדר ב- $q.h$ וההקצאה שלו, כמו כמעט כל המשתנים הגלובליים, ב-`initiali.c` שיממש את הצמתים של הרשימות המקושרות. כל צומת מכיל שדות ליצוג העוקב, קודם, ומפתח. האינדקס של כל רשומה במערך ישמש לזהוי הצומת. המפתח ישמש למיון הרשומות בתור העדיפות, וגם יאכסן את גודל ההרדמה בתור ההפרשים של תור הרדומים. גודל המערך $q[]$ הוא מספר התהליכים במערכת (NPROC) + פעמיים מספר הסמפורים $(NSEM) + 4$: NPROC הצמתים הראשונים מיצגים את התהליכים, והיתר הם 2 צמתים לכל תור אפשרי: צומת התחלה (HEAD) וצומת סיום (TAIL). הם תמיד יהיו צמתים עוקבים במערך. בסה"כ מספר התורים $NSEM + 2$: תור לכל סמפור וכן שני התורים: תור ה-`ready`, התהליכים הממתינים לתורם לקבל שירותי CPU, ותור הרדומים. בתור לא ריק יהיו רשומות של נציגי תהליכים ברשימה המקושרת בין ה-HEAD ל-TAIL. בתור ריק ה-HEAD וה-TAIL יצביעו אחד לשני. ערכי המפתחות של צומתי ה-HEAD תמיד 32768-, של צומתי ה-TAIL תמיד 32767+, הערכים הקיצוניים של מספר שלם 16 ביט, ואלה אינם משתנים.

השימוש במבנה הזה אפשרי משום שלעולם לא יהיה צורך ליצג תהליך בשני תורים – כל תהליך נמצא רק בסטטוס אחד בכל רגע.

השימוש בסוגי התורים יהיה ע"י הרוטינות הבאות:

תור עדיפיות:

הכנסה ממוינת לפי מפתח ע"י הפונקציה insert, הוצאה ע"י הפונקציה getlast - הוצאת האחרון (על יד ה-TAIL) מהתור והחזרת המזהה שלו.

בתור עדיפיות התהליכים מסודרים בסדר עולה מה-HEAD ל-TAIL.

תור FIFO:

הכנסה לסוף התור (על יד ה-TAIL) ע"י הפונקציה enqueue הוצאת הראשון (ליד ה-HEAD) ע"י הפונקציה getfirst

תור הרדומים

לא נדון בתור הזה בשלב זה.

QCL P-1552 2/3 2/20

Qkey

Qnext

Qprev

0			
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			
28			
29			
30			
31			
32			
33			
34			
35			
36			
37			
38			
39			
40			
41			
42			
43			
44			
45			
46			
47			
48			
49			
50			
51			
52			
53			
54			
55			
56			
57			
58			
59			
60			
61			
62			
63			
64			
65			
66			
67			
68			
69			
70			
71			
72			
73			
74			
75			
76			
77			
78			
79			
80			
81			
82			
83			
84			
85			
86			
87			
88			
89			
90			
91			
92			
93			
94			
95			
96			
97			
98			
99			

NPROC

213

213/20

2(1/SEIM
+2)

213

213/5

Phase 2

head1

tail1

head2

tail2

	key	next	prev
0			
1			
2	80	tail	4 5
3			
4	70	2 5	head
5	75	2	4
6			
head	-32768	4	-
tail	+32767	-	2

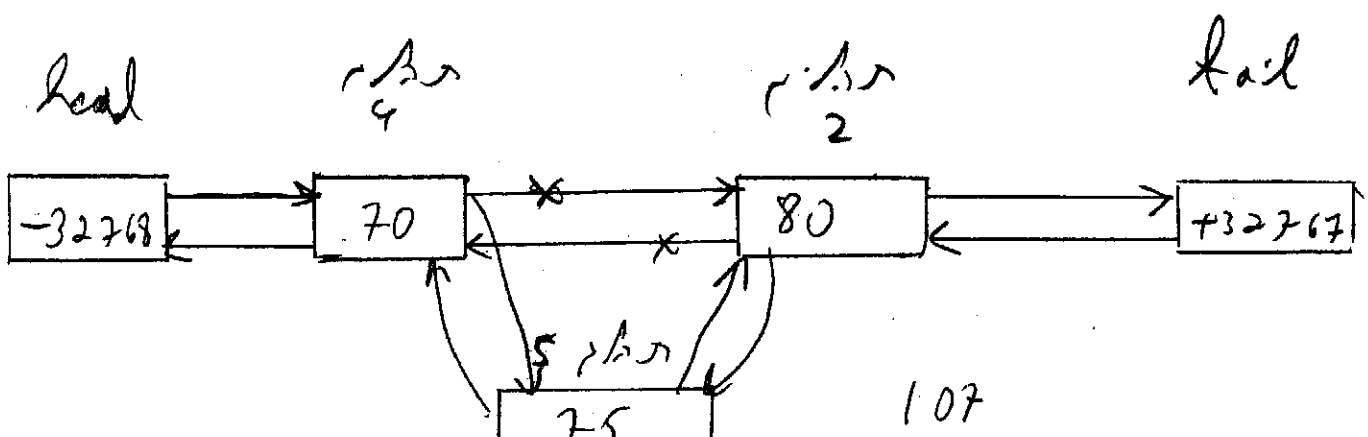
10013

7158

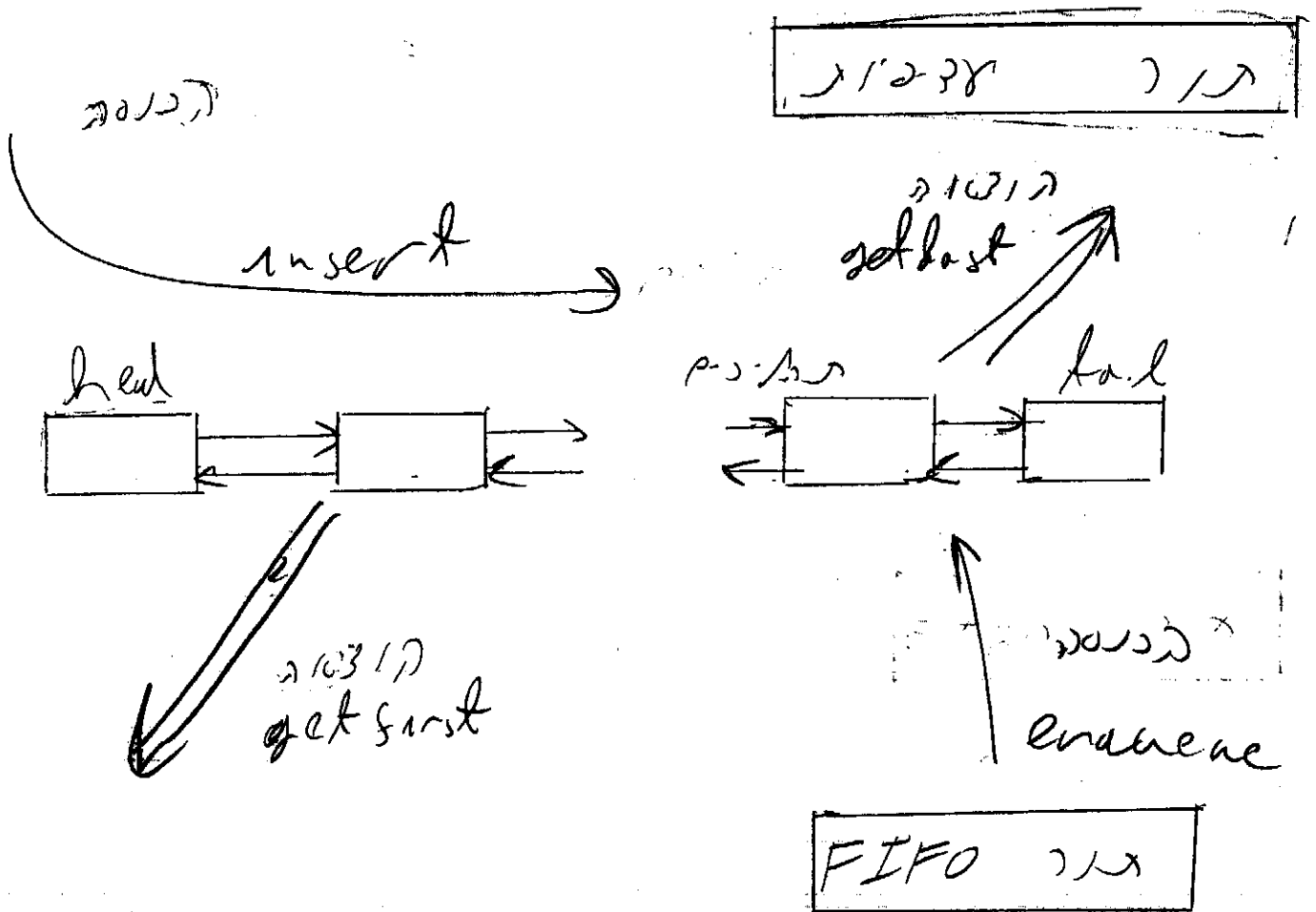
51232

head

tail



107



```

/* q.h - firstid, firstkey, isempty, lastkey, nonempty */

/* q structure declarations, constants, and inline procedures */

#define NQENT      NPROC + NSEM + NSEM + 4 /* for ready & sleep */

struct gent {
    /* one for each process plus two for */
    /* each list */
    int    qkey;      /* key on which the queue is ordered */
    int    qnext;     /* pointer to next process or tail */
    int    qprev;     /* pointer to previous process or head */
};

extern struct gent q[];
extern int    nextqueue;

/* inline list manipulation procedures */

#define isempty(list)    (q[(list)].qnext >= NPROC)
#define nonempty(list)  (q[(list)].qnext < NPROC)
#define firstkey(list)  (q[q[(list)].qnext].qkey)
#define lastkey(tail)   (q[q[(tail)].qprev].qkey)
#define firstid(list)   (q[(list)].qnext)

#define EMPTY    -1          /* equivalent of null pointer */

```



```

/* insert.c - insert */

#include <conf.h>
#include <kernel.h>
#include <q.h>

/*-----
 * insert -- insert a process into a q list in key order
 *-----
 */

int insert(proc, head, key)
int proc;          /* process to insert */
int head;          /* q index of head of list */
int key;           /* key to use for this process */
{
    int next;      /* runs through list */
    int prev;

    next = q[head].qnext;
    while (q[next].qkey < key) /* tail has MAXINT as key */
        next = q[next].qnext;
    q[proc].qnext = next;
    q[proc].qprev = prev = q[next].qprev;
    q[proc].qkey = key;
    q[prev].qnext = proc;
    q[next].qprev = proc;
    return(OK);
}

```

```
/* getitem.c - getfirst, getlast */
```

```
#include <conf.h>
#include <kernel.h>
#include <q.h>
```

```
/*-----
 * getfirst -- remove and return the first process on a list
 *-----
```

```
*/
```

```
int getfirst(head)
    int head; /* q index of head of list */
{
    int proc; /* first process on the list */

    if ((proc=q[head].qnext) < NPROC)
        return( dequeue(proc) );
    else
        return(EMPTY);
}
```

```
/*-----
 * getlast -- remove and return the last process from a list
 *-----
```

```
*/
```

```
int getlast(tail)
    int tail; /* q index of tail of list */
{
    int proc; /* last process on the list */

    if ((proc=q[tail].qprev) < NPROC)
        return( dequeue(proc) );
    else
        return(EMPTY);
}
```

```
/* queue.c - dequeue, enqueue */
```

```
#include <conf.h>
#include <kernel.h>
#include <q.h>
```

```
/*-----
 * enqueue -- insert an item at the tail of a list
 *-----
 */
```

```
int enqueue(item, tail)
int item;                /* item to enqueue on a list */
int tail;                /* index in q of list tail */
{
    struct qent    *tptr;    /* points to tail entry */
    struct qent    *mptr;    /* points to item entry */

    tptr = &q[tail];
    mptr = &q[item];
    mptr->qnext = tail;
    mptr->qprev = tptr->qprev;
    q[tptr->qprev].qnext = item;
    tptr->qprev = item;
    return(item);
}
```

```
/*-----
 * dequeue -- remove an item from a list and return it
 *-----
 */
```

```
int dequeue(item)
int item;
{
    struct qent    *mptr;    /* pointer to q entry for item */

    mptr = &q[item];
    q[mptr->qprev].qnext = mptr->qnext;
    q[mptr->qnext].qprev = mptr->qprev;
    return(item);
}
```

```

/* newqueue.c - newqueue */

#include <conf.h>
#include <kernel.h>
#include <q.h>

/*-----
 * newqueue -- initialize a new list in the q structure
 *-----
 */
int newqueue()
{
    struct qent *hptr;          /* address of new list head */
    struct qent *tptr;          /* address of new list tail */
    int hindex, tindex;         /* head and tail indexes */

    hptr = &q[ hindex=nextqueue++ ]; /* nextqueue is global variable */
    tptr = &q[ tindex=nextqueue++ ]; /* giving next used q pos. */
    hptr->qnext = tindex;
    hptr->qprev = EMPTY;
    hptr->qkey = MININT;
    tptr->qnext = EMPTY;
    tptr->qprev = hindex;
    tptr->qkey = MAXINT;
    return(hindex);
}

```