

טבלת התהליכים

טבלת התהליכים, ה-proctab, הינה מערך של רשומות המכיל אינפורמציה של כל תהליך שנחוץ להיות נגיש למערכת ההפעלה בכל רגע נתון ממספר שיקולים. האינפורמציה היא מצב התהליך, העדיפות שלו, מספר הסמפור שהוא ממתין לו (במידה וזה המצב), מבנה הנתונים של מנגנון ההודעות, מצביעים למחסנית הפרטית של התהליך ועוד שדות עזר: שם התהליך, מספר הפרמטרים וקוד התחלתי.

ההכרזה ב-initiali.c המהווה הקצאה של שטח ל-proctab הוא:

```
struct pentry proctab[NPROC].
```

מצבי התהליכים

בכל רגע נתון כל תהליך נמצא באחד מסידרה מוגדרת של מצבים. רשימת המצבים ב-XINU הסטנדרטי הם:

PRFREE

התהליך אינו בשימוש, חופשי להקצאה בעתיד.

PRCURR

התהליך הוא זה שמקבל שירותי CPU.

PRREADY

התהליך מעוניין בשירותי CPU, אך הוא משרת תהליך אחר והתהליך הזה מחכה לתורו.

PRSUSP

התהליך מושהה לזמן בלתי מוגבל.

PRSLLEEP

התהליך רדום לזמן קצוב.

PRWAIT

התהליך ממתין לסמפור.

PRRCEV

התהליך ממתין להודעה.

מנגנון החלפת התהליכים

כל הנושא של יצירת תהליכים, הפעלתם, והרצתם כביכול במקביל אפשרי אך ורק הודות למנגנון המאפשר עצירת ריצתו של התהליך המורץ כרגע ע"י ה-CPU והחלפתו בתהליך אחר. זהו מהלך לא פשוט בכלל, כרוך במספר מחויבויות, אבל אפשר לדבר על שורה תחתונה, או שתיים כאילו:

העובדה שה-CPU משרת תהליך מסוים נובע בעיקר מהעובדה שאוגרי ה-CPU (IP, AX, BX, ..., SP), מכילים את הערכים "שלו" – הערכים שהקוד שלו מעונין בהם. זה קובע את הפקודות הבאות לבצוע, המחסנית הנוכחית, משמעות ערכי האוגרים הכלליים וכו'.

בפועל החלפת תהליכים היא, בין השאר אבל בעיקר, מהלך של שימור אוגרים של התהליך היוצא, ושיחזור אוגרים של התהליך הנכנס. באנגלית העברת שליטה נקראית context switch, בתרגום חופשי "החלפת ההקשר" (של ה-CPU) וזה בעצם מה שהחלפת ערכי האוגרים עושה. המימוש ב-XINU ו-UNIX מבוסס על המרכזיות של מחסנית המערכת, זו שמוצבעת על ידי האוגרים BP, SP, SS: לכל תהליך יש שטח זכרון המשמשת מחסנית פרטית משלו, והחלפת המחסנית הפעילה (תוכן האוגרים BP, SP, SS) מזו של התהליך היוצא לזה של הנכנס עושה את עיקר המשימה.

השאלה העיקרית היא מהי הסיבה שהחלפת המחסניות די בכך לממש את העברת השליטה מתהליך אחד לשני. הסיבה שדי בכך היא שיחד עם מנגנון הקריאות לפונקציה של C, פקודת המכונה CALL ומנגנון הפסיקות, קביעת המחסנית קובעת בסופו של דבר את הפקודות הבאות לביצוע (האוגרים IP ו-CS), את המשתנים האוטומטיים הפעילים (דרך BP) ואת משתני אוגר SI ו-DI. על האוגרים הכלליים האחרים (AX, BX, CX, DX), C ממילא אינו מסתמך עליהם. קביעת אוגר הדגלים נעשה ע"י קוד מיוחד של XINU וגם הוא נשמר במחסנית. אשר לאוגרי הסגמנטים, הם אינם משתנים בקוד במודל של Turbo C שאנחנו עובדים איתו, הם עשויים להשתנות רק בפסיקות, וגם שם הם נשמרים ומשתחזרים דרך המחסנית.

בנוסף להחלפת ערכי האוגרים (בעיקר החלפת המחסנית הפעילה) למנגנון החלפת התהליכים שתי מחויבויות נוספות: מימוש מדיניות החלפת התהליכים, ועדכון מבנה הנתונים הגלובלי של XINU.

בכדי להמשיך אנחנו צריכים להכיר סידרה של מושגים ורקע נוסף.

מדיניות החלפת התהליכים של XINU

XINU דואג שבכל רגע התהליך בעל העדיפות הגבוהה ביותר במערכת שמעונין ב-CPU הוא זה שמקבל אותו. מידה ויש יותר מתהליך אחד בעל העדיפות הגבוהה ביותר, התהליכים הללו מקבלים שירותי CPU, לפי מכסת זמן, לפי סדר של ותק בהמתנה. ב-XINU כפי שאנחנו מקבלים אותו, אם יש תהליך יחיד בעל עדיפות גבוהה ביותר במערכת, הוא יקבל שירותי CPU כל עוד הוא מעונין בהם והוא יוחלף רק אם יותר על ה-CPU ויעבור למצב אחר. רק אם יש יותר מתהליך אחד בעדיפות הגבוהה יתבצע החלפת תהליך הנוכחי גם אם הוא מעונין להמשיך.

מבחינת resched משמעות הדבר שאם לתהליך הנוכחי יש עדיפות גבוהה ממש מכל תהליך ממתין בתור ה-READY הוא ממשיך לרוץ, גם אם המכסה שלו פגה. במקרה של שוויון, (בגרסה שלנו) התהליך ממשיך לרוץ אם המכסה שלו לא פגה.

מצב נוסף שבו לא יתבצע החלפת תהליכים היא אם המערכת לא יכולה להרשות זאת - יש דגל מערכת מיוחד לסמן מצב כזה. הסיבות למצב כזה הם די טכניות, לא נכנס לזה כאן.

מימוש מנגנון החלפת התהליכים

קבוצת התהליכים שמעונינים בשירותי ה-CPU הם התהליך הנוכחי היחיד שבפועל מקבל אותו וקבוצת התהליכים במצב PRREADY המסודרים בתור עדיפויות (הממש FIFO בין תהליכים שווי עדיפות).

מערכת XINU כתובה בצורה כזו שבכל מקרה שבו יתכן שיש צורך בהחלפת תהליכים, מתבצע קריאה לרוטינת C בשם **resched**. במידה ויש החלפת תהליכים, **העברת השליטה בפועל** נעשה בתוך **resched** ע"י קריאה לרוטינת אסמבלי פנימית בשם **ctxsw**, התהליך הנכנס נשלף מתור ה-READY ע"י הרוטינה **getlast**. במידה והתהליך היוצא נעשה PRREADY

(אנחנו נראה מתי זה קורה) הוא מוכנס לתור ה-ready ע"י הרוטינה insert. כל יתר המחויבויות של מנגנון החלפת התהליכים (יתר מימוש המדיניות ושינוי מבנה הנתונים הגלובלי במקרה של החלפה) נעשה בקוד של resched.

היתרון בכך שיש רוטינה אחת לכל המקרים של החלפת התהליכים הוא שמימוש מנגנון החלפת התהליכים מרוכז במקום אחד, ובכדי להכניס שינוי כלשהוא צריך להתייחס רק אליו. החסרון הוא ש-resched נקרא במצבים שונים של המערכת והקוד צריך להביא את כל המקרים בחשבון.

כאמור, קריאה ל-recshed היא רק בדיקה, האם יש צורך לבצע החלפת תהליכים. למעשה, הקריאות ל-resched נופלות לתוך שתי קטגוריות עיקריות:

1. קריאות ל-resched ביוזמת התהליך.
2. קריאות ל-resched ביוזמת המערכת.

קריאות ל-resched ביוזמת התהליך הם למעשה חלק מויתור של התהליך על ה-CPU. זה קורה במקרים שתהליך מגלה שהוא צריך להמתין לסמפור, ממתין להודעה, משהה את עצמו, מרדים את עצמו לזמן קצוב או פשוט חדל להתקיים. במצבים הללו התהליך הנוכחי דואג לשנות את מצבו לפני הקריאה ל-resched. זה גם הדרך ש-resched חש שמדובר בסוג הראשון של קריאה ל-resched. בסוג הזה של קריאות ל-resched, החלפת התהליכים מתבצעת בכל מקרה, גורל התהליך היוצא הוא להישאר במצב שבחר לעצמו.

קריאות ל-resched ביוזמת המערכת הם מצבים שבהם המערכת יוזמת קריאה ל-resched משיקולים שלה, למרות שהתהליך הנוכחי מעוניין להמשיך לרוץ. הדבר קורה או משום שמכסת הזמן של התהליך נגמרה, או שחל שינוי בסטטוס של תהליך שעד עכשיו לא היה מעוניין ב-CPU, אבל עכשיו כן מעוניין בו. הדבר יכול לנבוע מכך שתהליך יצא מהשהיה, נעשה signal לאחד הסמפורים, שתהליך ממתין להודעה קיבל אחת כזו, או שלתהליך רדום נגמרה זמן ההרדמה.

הדרך ש-resched חש שמדובר בסוג כזה של קריאה ל-resched הוא שהתהליך הנוכחי מגיע ל-resched במצב PRCURR. כאן לא בהכרח שיתבצע החלפת תהליכים, בהתאם למדיניות החלפת התהליכים. במידה ויש בכל זאת החלפת תהליכים, התהליך היוצא נעשה PRREADY.

סוג הקריאה ל-resched	הסיבה לקריאה	כיצד resched מבחין בין 2 סוגי הקריאות?	האם מתבצעת החלפה?	במקרה של החלפה, גורל התהליך הנוכחי
ביוזמת התהליך	ויתור על זכאות על המעבד של התהליך הנוכחי	שדה ה-pstate של התהליך הנוכחי \neq PRCURR	כן, בכל מקרה	נשאר במצב שבחר לעצמו
ביוזמת המערכת	שינוי במצב המערכת, הדורש הערכה מחדש על הזכאות על המעבד	שדה ה-pstate של התהליך הנוכחי = PRCURR	יכול להיות שלא, resched עשוי להחליט שלא לבצע החלפה, לפי מדיניות החלפת התהליכים	resched אחראי להעביר את התהליך ל-PRREADY

הרוטינה ctxsw

בכדי להבין את הקוד של ctxsw צריך לקרוא אותו מתוך ההנחה, שכל תהליך שאינו רץ כרגע נמצא באמצע קריאה ל-ctxsw. ההנחה הזו היא בפירוש נכונה לכל תהליך שרץ פעם, ומדומה לתהליכים חדשים שטרם רצו.

תפקידו העיקרי של ctxsw הוא לשמר את ערך אוגר ה-SP של התהליך היוצא בשדה ה-pregs של הרשומה שלו (התהליך) ב-proctab ולשלוף את ערך ה-pregs של התהליך הנכנס לתוך האוגר SP. משמעות הדבר הפעלת המחסנית של התהליך הנכנס על פני המחסנית של התהליך היוצא.

מלבד האמור לעיל, ctxsw משמר את SI ו-DI ומאתחל את אוגר הדגלים עבור תהליכים חדשים.

החלפת ערכי אוגרים בהחלפת הקשר – סיכום

כפי שנאמר קודם, העברת השליטה בין התהליכים, כלומר החלפת ההקשר, הוא תהליך של שימור של ערכי האוגרים של התהליך היוצא והחלפתם בערכי התהליך הנכנס. זהו מהלך הנפרש על פני כמה רוטינות שבמרכזו החלפת ההצבעה של המחסנית ה-CPU.

ב-XINU קבוצת האוגרים הרלוונטית הינה
AX, BX, CX, DX, BP, SP, SI, DI, IP, CS, SS, DS, FLAGS.

כאן נפרט איך נשמר/מוחלף כל אחד מהאוגרים הללו:

האוגרים

AX, BX, CX, DX, FLAGS

אינם צריכים להישמר בהסתעפויות רגילות של טורבו C ולכן נשמרים רק בהקשר של פסיקות על ידי מה שיקרה רוטינת הטיפול בפסיקות של XINU (להלן "סדרן הפסיקות") שאותו נראה בהמשך.

האוגרים

SI, DI, BP

כאשר מדובר בקריאות מערכת, האוגרים הללו נשמרים ומשתחזרים ע"י מוסכמות קריאה של פונקציות C (גם על ידי ctxsw כחלק מזה). במקרה של פסיקה הם נשמרים סופית ע"י סדרן הפסיקות.

האוגרים

CS, SS, DS, ES

אינם משתנים במהלך הריצה של XINU ולכן הם משתמרים ומשתחזרים רק ע"י סדרן הפסיקות.

האוגר

SP

נשמר / משתחזר דרך ה-proctab על ידי ctxsw ובהמשך או, במקרה של קריאת מערכת, ע"י מוסכמות הקריאה של טורבו C או, במקרה של פסיקה, ע"י סדרן הפסיקות.

האוגר

IP

נשמר ומשתחזר בעקבות השינוי ב-SP, שוב במקרה של קריאת מערכת, ע"י מוסכמות הקריאה של טורבו C או, במקרה של פסיקה, ע"י סדרן הפסיקות.

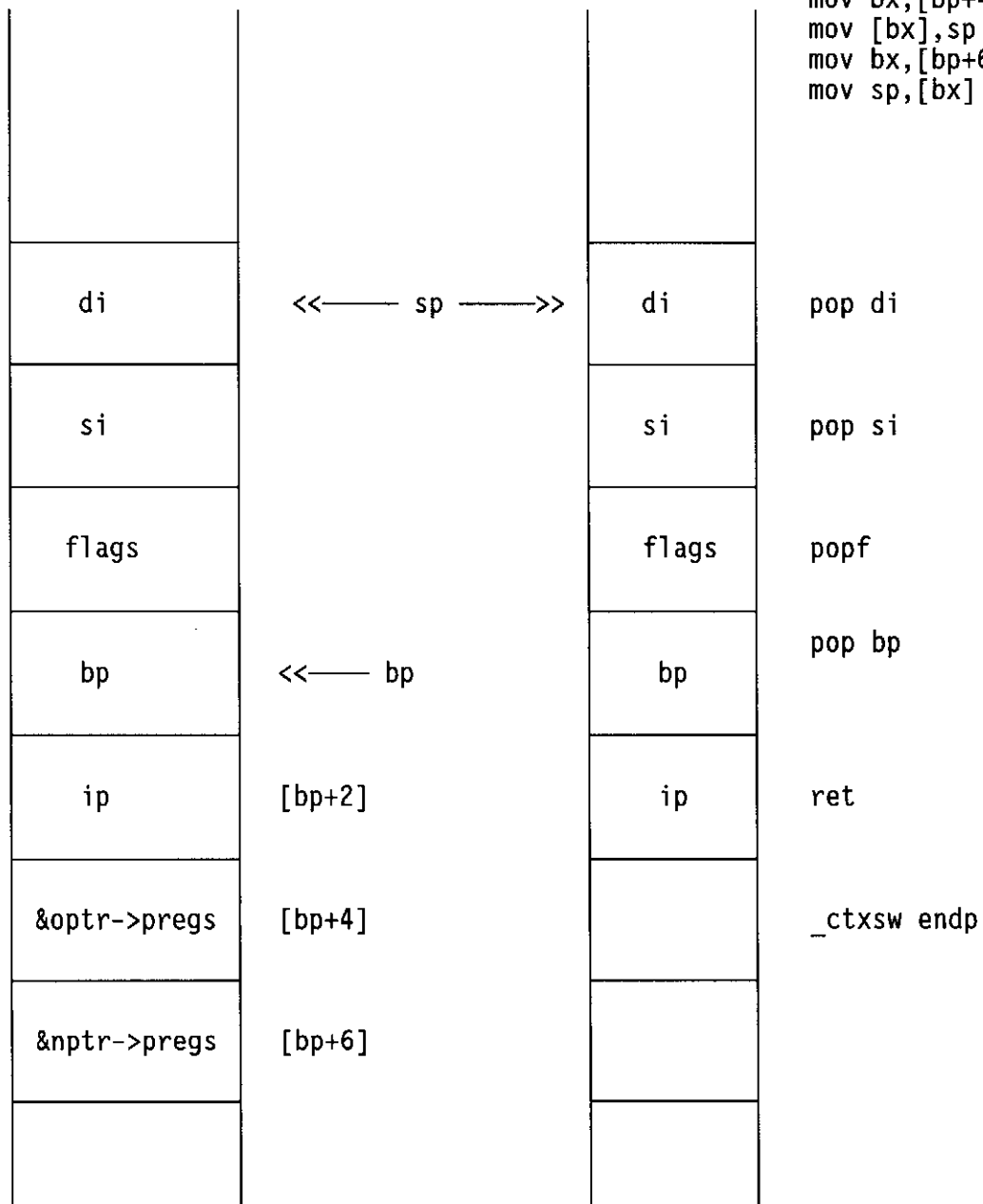
מחסנית תהליך יוצא

מחסנית תהליך נכנס

```

_ctxsw proc near
    push bp
    mov bp,sp
    pushf
    cli
    push si
    push di
    mov bx,[bp+4]
    mov [bx],sp
    mov bx,[bp+6]
    mov sp,[bx]

```




```

/* proc.h - isbadpid */
/* 8088 version */

/* process table declarations and defined constants */

#ifndef NPROC
#define NPROC 30
#endif

/* process state constants */

#define PRCURR '\01' /* process is currently running */
#define PRFREE '\02' /* process slot is free */
#define PRREADY '\03' /* process is on ready queue */
#define PRRECV '\04' /* process waiting for message */
#define PRSLEEP '\05' /* process is sleeping */
#define PRSUSP '\06' /* process is suspended */
#define PRWAIT '\07' /* process is on semaphore queue*/

/* miscellaneous process definitions */

#define PNMLEN 9 /* length of process "name" */
#define NULLPROC 0 /* id of the null process; it is always eligible to run */

#define isbadpid(x) (x<=0 || x>=NPROC)

/* process table entry */

struct pentry {
    char pstate; /* process state: PRCURR, etc. */
    int pprio; /* process priority */
    int psem; /* semaphore if process waiting */
    int pmsg; /* message sent to this process */
    int phasmgs; /* nonzero iff pmsg is valid */
    char *pregs; /* saved environment */
    char *pbase; /* base of run time stack */
    word plen; /* stack length in bytes */
    char pname[PNMLEN+1]; /* process name */
    int pargs; /* initial number of arguments */
    int (*paddr)(); /* initial code address */
};

extern struct pentry proctab[];
extern int numproc; /* currently active processes */
extern int nextproc; /* search point for free slot */
extern int currpri; /* currently executing process */

```

```
/* resched.c - resched */
```

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

```
/*-----
 * resched -- reschedule processor to highest priority ready process
 *
 * Notes:      Upon entry, currpid gives current process id.
 *              Proctab[currpid].pstate gives correct NEXT state for
 *              current process if other than PRCURR.
 *-----
 */
```

```
int resched()
{
    register struct pentry *optr; /* pointer to old process entry */
    register struct pentry *nptr; /* pointer to new process entry */

    optr = &proctab[currpid];
    if ( optr->pstate == PRCURR )
    {
        /* no switch needed if current prio. higher than next */
        /* or if rescheduling is disabled ( pcxflag == 0 ) */
        if ( sys_pcxget() == 0 || lastkey(rdytail) < optr->pprio
            || ( (lastkey(rdytail) == optr->pprio) && (preempt > 0) ) )
            return;
        /* force context switch */
        optr->pstate = PRREADY;
        insert(currpid,rdyhead,optr->pprio);
    } /* if */
    else if ( sys_pcxget() == 0 )
    {
        kprintf("pid=%d state=%d name=%s",
                currpid,optr->pstate,optr->pname);
        panic("Reschedule impossible in this state");
    } /* else if */

    /* remove highest priority process at end of ready list */

    nptr = &proctab[ (currpid = getlast(rdytail)) ];
    nptr->pstate = PRCURR; /* mark it currently running */
    preempt = QUANTUM; /* reset preemption counter */
    ctxsw(&optr->pregs,&nptr->pregs);

    /* The OLD process returns here when resumed. */
    return;
}
```

```

; ctxsw.asm - _ctxsw

    include dos.asm

    dseg
; null data segment
    endds

    pseg

    public _ctxsw

;-----
; _ctxsw -- context switch
;-----
; void ctxsw(opp,npp)
; char *opp, *npp;
;-----
; Stack contents upon entry to ctxsw:
;     SP+4 => address of new context stack save area
;     SP+2 => address of old context stack save area
;     SP   => return address
; The addresses of the old and new context stack save areas are
; relative to the DS segment register, which must be set properly
; to access the save/restore locations.
;
; The saved state consists of the current BP, SI and DI registers,
; and the FLAGS register
;-----
_ctxsw proc    near
    push      bp
    mov       bp,sp                ; frame pointer
    pushf     ; flags save interrupt condition
    cli       ; disable interrupts just to be sure
    push      si
    push      di                  ; preserve registers
    mov       bx,[bp+4]           ; old stack save address
    mov       [bx],sp
    mov       bx,[bp+6]           ; new stack save address
    mov       sp,[bx]
    pop       di
    pop       si
    popf      ; restore interrupt state
    pop       bp
    ret
_ctxsw endp

    endps

end

```