גירסה 1.00 - 26.3.2003

מערכת ההפעלה XINU

ניר אדר

מסמך זה הורד מהאתר http://underwar.livedns.co.il.

אין להפיץ מסמך זה במדיה כלשהי, ללא אישור מפורש מאת המחבר.

מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כל הזכויות שמורות ל**ניר אדר**

אנא שלחו תיקונים והערות אל המחבר.

מסמך זה מתאר את מערכת ההפעלה XINU תוך כדי התמקדות ב-XINU גירסה XINU.6pc היא מערכת הדומה מעט ל-UNIX, והיא נוצרה למטרות לימודיות עבור אנשים הבאים ללמוד כיצד בנויות מערכות הפעלה.

מסמך זה איננו מדריך למשתמש במערכת ההפעלה. בתחילה הוא נכתב ככזה, אולם אז השתנה הדגש. מטרת המסמך היא להכיר לקורא את מערכת ההפעלה XINU בצורה כזו שתבהיר כיצד כל אחד מחלקי המערכת עובד בנפרד, וכיצד החלקים השונים של המערכת משתלבים כדי לפעול יחד.

הצגנו את ההנחות השונות ששומשו לכתיבת כל אחד מחלקי מערכת ההפעלה, וכמו כן ציינו באגים הקיימים במערכת ההפעלה על מנת להציג את הבעיות המתעוררות כאשר כותבים מערכת הפעלה - וכן להדגיש את ההבדלים בין כתיבת תוכנית C רגילה לבין כתיבת מערכת הפעלה.

ידע נדרש להבנת מסמך זה: שליטה ב-C, הבנת האסמבלר של 8086, הכרת הדרך בה C מתרגמת את הקוד אל שפת מכונה והכרת מושגים בסיסיים בנושא מערכות הפעלה.

ניתן למצוא מדריכים על נושאים אלו באתר UnderWarrior Project.

כמו כן, מומלץ לקרוא את מסמך זה מול הקוד הרלוונטי של XINU, על מנת להבין את המערכת בצורה הטובה ביותר.

ברצוני להודות לצוות הקורס "מבוא למערכות הפעלה" בטכניון, סמסטר חורף תשס"ג, שענו לשאלות רבות שלי בנושא.

מערכת ההפעלה XINU

כללי

התוכנית הראשית מתחילה לרוץ בפונקציה xmain(). דוגמא לתוכנית בסיסית:

```
#include <kernel.h>
#include <tty.h>

int xmain()
{
	printf("Hello World\n");
	return 0;
}
```

קלט/פלט

קלט:

```
char ch = getc(<device>);
```

getc קוראת תו מההתקן ומחזירה את התו שהתקבל. אם לא קיים תו מחכה, getc תמתין עד לקבלת תו.

פלט:

```
putc(<device>, <char>)
```

putc מדפיסה את התו הנתון אל ההתקן. הקבוע CONSOLE מהווה את מציין ההתקן מקלדת/מסך הסטנדרטי.

חלונות

יצירת חלון תתבצע כך:

```
int hWin1 = open(CONSOLE, "#c1,r1:c2,r2", "fff/bbb");
```

- # מציין שהחלון יפתח עם מסגרת.
- c1, r1 – קורדינטות הפינה השמאלית העליונה.
- c2, r2 – קורדינטות הפינה הימנית התחתונה.
- bbb, fff מחרוזות של 3 אותיות המייצגות את צבעי החלון (foreground, background).
- הצבעים האפשריים: wht, yel, mag, red, cyn, blk, blu, grn.

מעבר בין החלונות יתבצע על ידי המקשים F1-F4. F10 משמש למעבר אל ה-CONSOLE.

פלט לחלון:

```
putc(hWin1, 'a');
fprintf(win1, "variable = %d\n", x);
```

קלט מחלון:

```
ch = getc(win1);
```

קלט/פלט בקבצים

- פתיחת קובץ לקריאה או כתיבה נעשית על ידי open.
- קלט/פלט לקובץ נעשים על ידי שמוש ב-getc ו-putc.
- סגירת קובץ על ידי close(dev).

דוגמא:

```
#include <conf.h>
#include <disk.h>
#include <file.h>

int xmain()
{
	int inputFile, outputFile;
	char ch;

	file1 = open(DOS, "C:\\folder\\input.txt", "r");
	file2 = open(DOS, "C:\\folder\\output.txt", "w");

	while ( (ch = getc(file1)) != EOF ) putc(file2, ch);

	close (file1);
	close (file2);
	return 0;
}
```

טיפול ברשימות ותורים

כללי

- במערכת ההפעלה מספר תהליכים יכולים לרוץ במקביל. במהלך הריצה, בכל רגע, הרבה תהליכים ממתינים למשאב – CPU, קלט/פלט, הודעות ועוד. מכיוון שמספר תהליכים עלולים להמתין לאותו משאב, יש לנהל תורי המתנות למשאבים השונים.
- מנגנון התורים והרשימות של XINU מיועד להחזיק תהליכים. איבר בתור הוא למעשה תהליך הנמצא באותו תור.
- מספר התהליכים מוגבל וכן גם מספר התורים, ולכן ניתן לממש את כל התורים על ידי מערך סטטי אחד, אשר ניתן לו השם q[].
- תהליך יכול להיות בתור אחד לכל היותר בכל רגע נתון, וכן להופיע פעם אחת בלבד באותו תור.
- לכל תהליך במערכת ההפעלה יש מספר המציין אותו, pid. בנושא התורים, מספר זה מציין אינדקס במערך q, המתאים לאותו תהליך. כלומר, אם התו ה-i מופיע בתור כלשהו, אז נאמר שהתהליך עם המציין ה-i שייך לאותו תור.
- מספר התהליכים המקסימאלי במערכת ההפעלה הינו NPROC.
- לכל תור יהיה תא head ותא tail בנוסף לתהליכים שלו. האינדקסים של tail ושל head יהיו בהכרח גדולים מ-(NPROC-1).

- מספר התאים במערך q מוגדר על ידי NQENT, שערכו NPROC + NSEM + MSEM + 4, וזאת על מנת לספק NPROC אינדקסים עבור התהליכים, ועוד head, tail עבור NSEM תורים עבור סמפורים במערכת, ועוד head, tail עבור תור ה-ready ותור ה-sleep.

מבנה רשומה

XINU משתמשת במספר סוגי רשימות: תורים, רשימות ממוינות, רשימות לא ממוינות.

לכל התורים אותו מבנה, המספק את צרכי התורים השונים.

מבנה רשומה:

- qnext – מצביע אל האיבר הבא או אל tail.
- qprev – מצביע אל האיבר הקודים או אל head.
- qkey – מפתח לפיו התור ממוין.

שלושת השדות הם מספרים מסוג int, ומתקיים כי qnext, qprev מכילים אינדקסים של תאים אחרים במערך.

פונקציות macro הקשורות לרשימות

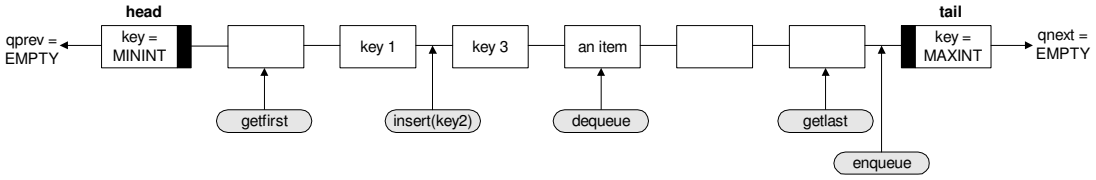
- **isempty(list)** מחזיר true אם הרשימה ריקה.
- **nonempty(list)** מחזיר true אם הרשימה אינה ריקה.
- **firstkey(list)** מחזיר את המפתח של האיבר הראשון אחרי ה-head ברשימה.
- **lastkey(tail)** מחזיר את המפתח של האיבר האחרון ברשימה לפני ה-tail.
- **firstid(list)** מחזיר את האינדקס של האיבר הראשון אחרי ה-head ברשימה.

כאשר המקרו מקבל "list", אנו מתכוונים שהוא מקבל את ה-head.

פונקציות לטיפול ברשימות

- **insert(proc, head, key)** הכנסת תהליך לרשימה לפי מפתח.
- **enqueue(item, tail)** הכנסת איבר בזנב הרשימה (תור).
- **dequeue(item)** ניתוק איבר מהרשימה והחזרת האינדקס שלו.
- **getfirst(head)** הסרת התהליך הראשון ברשימה והחזרת האינדקס שלו.
- **getlast(tail)** הסרת התהליך האחרון ברשימה והחזרת האינדקס שלו.
- **newqueue()** אתחול רשימה חדשה.

הערה לגבי המימוש: המפתח של tail הוא MAXINT. בעזרת ידיעת עובדה זו ממומשת סריקה של אברי התור, כשאנו יכולים לבדוק בכל שלב האם הגענו לסופו.



ניהול תהליכים

טבלת התהליכים

טבלת התהליכים (proctab) שומרת את כל האינפורמציה הדרושה להרצת תהליכים. היא מיוצגת כמערך של רשומות. לכל תהליך מוקצאת רשומה (pentry) המכילה את המידע על מצב התהליך ואת כל המידע על הקשר התהליך הדרוש להרצתו. האינדקס של תהליך בטבלת התהליכים נקרא מציין התהליך – process ID (pid). ניתן להתייחס אל תהליכים בעזרת המציין שלהם.

בטבלת התהליכים ישנן NPROC רשומות.

מבנה של רשומה בטבלת התהליכים:

```
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    phasmsg;            /* nonzero if 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           */
};
```

דגשים

- **pname** הוא בגודל PNMLEN+1, ולכן בפועל אנו מסוגלים לשמור בו PNMLEN תווים.
- **pbase** מכיל את כתובת תחילת המחסנית, שהוקצתה עם יצירת התהליך. נשתמש בשדה זה כשנבוא לשחרר את המחסנית של התהליך כאשר התהליך מסתיים.
- עדיפות התהליך הינה מספר שלם חיובי.

מדיניות זימון ב-XINU

XINU משתמשת בשילוב של תיזמון עדיפויות ו-Round Robin:

- בין תהליכים בעדיפויות שונות, XINU נותנת תמיד עדיפות לתהליכים בעדיפות גבוהה.
- בין תהליכים באותה עדיפות מתבצע Round-Robin.
- העדיפות של תהליך נבחרת בזמן יצירתו, אך ניתנת לשינוי בכל רגע.

החלפת תהליכים

החלפת תהליכים במערכת ההפעלה מבוצעת על ידי קריאה לפונקצית resched. הקריאה יכולה להיות יזומה על ידי התהליך – קריאה ל-()wait, ()signal וכו', או יזומה על ידי מערכת ההפעלה.

בכל רגע נתון רץ רק תהליך אחד – התהליך הנוכחי. התהליכים האחרים ממתינים לביצוע או ממתינים למשאב כלשהו.

לפי הכניסה ל-resched, תהליך יכול לשנות את מצבו (למשל על ידי wait). אם אין תהליך במצב CURRENT ואם אסורה החלפת תהליכים, תתבצע קריאה ל-PANIC.

אם זאת, החלפת תהליכים במצב אסור איננה אפשרית, בהנחה שאין באגים במערכת ההפעלה.

אם קיים תהליך במצב CURRENT (כלומר מערכת ההפעלה יוזמת את ההחלפה), ואם אסורה החלפת תהליכים, resched תסתיים בלי לנסות להחליף תהליכים.

כל החלפות התהליכים ב-XINU נעשים רק באמצעות resched. הפונקציה resched קוראת לפונקצית עזר – ctxsw, המבצעת את ההחלפה בפועל. הנקודה בה ממש מתחלף ההקשר היא השורה ב-ctxsw בה אנו מחליפים את תוכן הרגיסטר SP.

כאשר תהליך 1 קורא ל-resched, החלפת התהליכים מתרחשת לפני סיום resched, אולם resched ממשיכה להתבצע, ובסופה היא מחזירה את הביצוע לתהליך 2.

resched, בדומה לקריאות מערכת אחרות, בודקת את תקינות הפרמטרים המגיעים אליה.

נשים לב כי resched לא נקראת במישרין על ידי המשתמש. היא חלק פנימי של המערכת.

בכל רגע, התהליכים הממתינים למעבד נשמרים בתור ממוין לפי עדיפות. תור זה נקרא ready queue. XINU ניגשת לתור זה על ידי המשתנים rdyhead, rdytail.

בתור ה-ready, האיבר בסוף התור הוא בעל העדיפות הגבוהה ביותר. התהליך הבא שירוץ יהיה התהליך הנוכחי, או איבר זה שבסוף תור ה-ready. התהליך הנוכחי איננו נמצא בתור ה-ready.

לפני שאנו מחליפים את התהליך הנוכחי, אנו צריכים לשמור את נתוני ההקשר שלו, על מנת שנוכל מאוחר יותר לגרום לתהליך להמשיך בפעולתו.

השדה היחידי בטבלת התהליכים המשמש בהחלפת תהליך הוא **pregs** (השומר את ערך SP, מצביע המחסנית). כל שאר נתוני ההקשר נשמרים על המחסנית.

כיצד נשמר כל נתון הקשר?

- **AX, BX, CX, DX** נשמרים על ידי הקור ל-resched אם ערכיהם חשובים לו.
- **SI, DI, BP, Flags** נשמרים על ידי ctxsw על גבי המחסנית.
- **CS, DS, ES, SS** אינם משתנים ולכן אינם נשמרים. (בפסיקה: נשמרים ברמת הפסיקה).
- **IP** נשמר על ידי פקודת call ל-resched. הערך המעודכן ברגע ההחלפה אינו צריך להישמר.
- **SP** ערכו לאחר הכנסת ההקשר למחסנית נשמר בטבלת התהליכים. משתחזר אוטומטית לערכו הקודם בהוצאה מהמחסנית.

משתני התהליך הנוכחי, שדות טבלת התהליכים ותור ה-ready מתעדכנים על ידי resched.

נביט בקוד של ctxsw:

```
_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
```

דגשים:

- הפקודה cli היא למעשה מיותרת, כי אנו קוראים לפונקציה ctxsw תמיד מתוך resched, שכבר דאגה לבטל את הפסיקות.
- push si, push di דוחפות את הרגיסטרים של התהליך הראשון. pop di, pop si שולפות את הרגיסטרים של התהליך השני.

ביצוע resched

	התהליך יוזם החלפת תהליכים	מערכת ההפעלה יוזמת את החלפת התהליכים
הסיבה	בקשת קלט/פלט או בקשה אחרת.	1. תהליך בעל עדיפות רוצה להתבצע. 2. הזמן המוקצב לתהליך עבר. 3. התהליך הנוכחי יזם קריאת מ"ה המשפיעה על תהליך אחר (suspend, chprio לתהליך אחר) וכתוצאה מכך נגרמת החלפת תהליכים.
ביצוע	התהליך משנה את מצבו בטבלת התהליכים לפני הקריאה ל-resched.	
pstate בכניסה ל-resched	שונה מ-PRCURR ושונה מ-PRREADY. אינו משתנה במהלך resched	PRCURR, המשתנה במהלך resched להיות PRREADY.

מתי חוזר לביצוע?	רק לאחר שיסתיים הטיפול בבקשתו.	כששוב יגיע תורו.
ההחלפה מובטחת?	תמיד.	לא. resched רשאית לא להחליף תהליכים.
כשאסורה החלפת תהליכים (pcxflag=0)	מצב אסור - PANIC	לא מתבצעת החלפה.

disable/restore – מניעה הדדית ברמת מערכת ההפעלה

קטעים רבים של קוד מערכת ההפעלה הם קטעים קריטיים אותם נרצה לבצע ללא החלפת תהליכים.

אומנם מערכת ההפעלה יכולה שלא ליזום החלפת תהליכים, אולם פסיקות חומרה עלולות לגרום להחלפה, ולכן בקטעים קריטיים נרצה למנוע פסיקות חומרה.

disable זהו מקרו, המכבה את הפסיקות, ושומר בפרמטר המועבר אל disable את מצב הפסיקות לפני כיבוין. **restore** זהו מקרו המבצע את הפעולה ההפוכה. הוא משחזר את הדגלים מהמשתנה הנתון לתוך רגיסטר הדגלים. נשים לב, ש-restore רק משחזר את רגיסטר הדגלים הקודם. הוא איננו בהכרח מאפשר שוב פסיקות.

כל קריאת מערכת עטופה, וחייבת להיות עטופה, בצמד disable/restore.

מאחר ש-disable שומר את המצב הנוכחי של הדגלים ו-restore מחזיר את אותו המצב, ניתן להשתמש בהם בצורה מקוננת, תוך שמירה על הכללים הבאים:

1. לא יתקיימו שתי קריאות ל-disable עם אותו פרמטר ל-disable.
2. הפונקציה הקוראת ל-disable היא הקוראת ל-restore בסיום הקטע הקריטי.

מקרו נוסף הוא **enable** המאפשר להדליק את דגל הפסיקות, ללא קשר לערכו האחרון בעת כיבויו. לא רצוי להשתמש במאקרו זה, ו-XINU במימושה הנוכחי אינה משתמשת בו.

למשתמש אסור לקרוא ל-disable/restore/enable. במערכות הפעלה מתקדמות המשתמש אינו יכול לקרוא להן.

הפונקציה ready

הפונקציה מקבלת מציין תהליך ומכניסה אותו לתור ה-ready.

בדרך כלל מיד אחרי קריאה לפונקציה ready נוסיף קריאה לפונקציה resched.

אם זאת, הקריאה ל-resched לא שולבה בתוך הפונקציה ready, מכיוון שיתכן לעיתים מצב בו נרצה להעביר כמה תהליכים מתור כלשהו אל תור ה-ready. במקרה כזה, קריאה ל-resched אחרי הכנסת כל איבר תהיה בזבוז. במקרה כזה נרצה לקרוא ל-resched פעם אחת, לאחר הכנסת כל האיברים לתור ה-ready. מכיוון שכך, לא הוספה לפונקציה ready קריאה ל-resched בסופה.

הפונקציה kill

פונקציה זו מקבלת מציין תהליך והורגת את התהליך.

הפונקציה משחררת את המחסנית של התהליך. במידה והתהליך נמצא ברשימה כלשהי, הפונקציה מוציאה אותו ממנה (אנו יודעים אם תהליך היה ברשימה כלשהי לפי המצב הנוכחי שלו).

בסיום הפונקציה מסמנת את התא של התהליך ב-proctab כתא פנוי.

במידה ואין תהליכים נוספים במערכת ההפעלה, הפונקציה kill מסיימת את פעולתה של מערכת ההפעלה XINU.

נשים לב כי ב-kill מופיעה קריאה ל-resched, במקרה שהתהליך שאנו הורגים הוא התהליך הנוכחי. לכאורה אסור לנו להשתמש ב-resched, כי היא ניגשת אל המחסנית של התהליך שהרגנו, וכבר שחררנו את המחסנית, אולם, kill עטופה ב-disable/restore, כלומר מנענו פסיקות, ולכן ידוע בוודאות כי הזיכרון הראשי נותר ללא שינוי מאז ששחררנו את המחסנית, ולכן מותר לקרוא ל-resched.

יצירת תהליכים חדשים

יצירת תהליכים חדשים נעשית על ידי קריאת מערכת ההפעלה create.

כל תהליך במערכת רשאי ליצור תהליכים אחרים ללא הגבלה, ובכל עדיפות. התהליך נוצר במצב PRSUSP. התהליך הנוצר יקבל מחסנית בגודל המתבקש על ידי פרמטר, בעל העדיפות המתבקשת, השם המתבקש, וכן פרמטרים לפי הנתון המוגדר ב-create. קוד התהליך הנוצר יתחיל להתבצע

בתחילת הפונקציה המועברת כפרמטר. create בודקת אם קיימת רשומה פנויה בטבלת התהליכים, בודקת את תקינות הפרמטרים, ומקצה מחסנית לכל תהליך. create יוצרת מחסנית המחקה פעולת קריאה ל-ctxsw. מכאן שכאשר תתבצע החלפת ההקשר, לא נצטרך לבצע טיפול במקרה מיוחד עבור תהליך חדשה. כאשר מסתיימת פעולת השיגרה הראשית, יש לסיים את פעולת התהליך. לכן create מחקה על המחסנית את פעולה הקריאה לפונקציה ה"קוראת" לשגרה הראשי, ושמה שם את כתובת השגרה userret. שיגרה זו מבצעת kill לתהליך הנוכחי. דגש: כאשר יוצרים תהליך חדש, והוא נהפך לפעיל בעזרת ctxsw, איננו חוזרים אל resched כאשר ctxsw מסתיימת, אלא אל הפונקציה המועברת כפרמטר.

הפונקציה הפנימית ()newpid מביאה id עבור התהליך החדש. היא איננה מסמנת id זה כתפוס. create, לאחר שהתהליך נוצר בהצלחה, תעשה זאת. הפונקציה ()newpid מוגדרת LOCAL, כלומר פונקציה פנימית השייכת למנגנון התהליכים, ושאר חלקי מערכת ההפעלה לא מכירים אותה.

נקודה למחשבה: האם לא יכול להיווצר מצב, ששני תהליכים מנסים ליצור תהליך חדש בו זמנית, ושני התהליכים החדשים מקבלים את אותו PID? התשובה היא לא. כאשר נוצר תהליך חדש, create משתמשת ב-disable על מנת לחסום זמנית את פסיקות המערכת, וכך פעולת יצירת התהליך הינה אטומית ולא יכולה להיות מופרעת בדרך.

דוגמא ליצירת תהליכים:

```
void prA()
{
	while (1) putc(CONSOLE, 'A');
}

void prB()
{
	while (1) putc(CONSOLE, 'B');
}

int xmain()
{
	resume(create(prA, INITSTK, INITPRIO, "proc 1", 0));
	resume(create(prB, INITSTK, INITPRIO, "proc 2", 0));
	return 0;
}
```

משתני ניהול תהליכים

המציין של התהליך הנוכחי שמור במשתנה הגלובלי currpid, המאפשר התייחסות לתהליך הנוכחי מקוד מערכת ההפעלה.

התהליכים המחכים למעבד שמורים בתור דו כווני על פי עדיפותם. תור זה נקרא ready queue. XINU ניגשת לתו זה על ידי המשתנים rdyhead, rdytail.

חשוב: אסור להשתמש במשתנים אלו בתוכניות המשתמש. אלו משתנים פנימיים של מערכת ההפעלה.

התהליך הריק (The NULL process)

דרישה מהפונקציה resched היא שבכל יציאה מפונקציה זו, יהיה תהליך במצב PRCURR.

מתעוררת השאלה מה מערכת ההפעלה צריכה לעשות כאשר אין אף תהליך מוכן לריצה.

כפי שמומשה, הפונקציה resched יכולה רק להחליף בין תהליך אחד לשני (ולא ליצור תהליכים), ולכן הפיתרון הוא יצירת תהליך שתמיד יהיה מוכן לחישוב. תהליך זה נקרא NULL process.

תהליך זה יקבל את העדיפות המזערית האפשרית, ולכן יהפך לנוכחי רק כאשר אין אף תהליך אחר הדרוש את המעבד. כל פעולתו היא לולאה אינסופית שאינה מבצעת דבר.

מציין התהליך, ה-pid, של תהליך ה-NULL הינו 0.

השהית תהליכים

ניתן להשהות את פעולתו של תהליך ולמנוע ממנו להפך לתהליך הנוכחי.

חידוש פעולת התהליך יעשה על ידי בקשה מפורשת שתינתן על ידי תהליך אחר.

לא ניתן להשהות תהליך שאינו במצב PRREADY או PRCURR. תהליך מושהה מקבל את המצב PRSUSP. תהליך יכול להשהות את עצמו (אך לא לחדש את פעולת עצמו).

ביצוע ההשהיה:

```
suspend(pid);
```

חידוש הפעולה:

```
resume(pid);
```

suspend מבצע את ההשהיה בצורה שונה עבור תהליכים במצב PRCURR ו-PRREADY.

כאשר תהליך במצב PRREADY, נוציא מרשימת ה-ready ואז נקבע אותו להיות PSUSP.
כאשר תהליך במצב PRCURR, נקבע אותו להיות במצב PRSUSP ונקרא ל-()resched.

הפונקציה suspend מחזירה את עדיפות התהליך המושהה. נשים לב שדגימת ערך העדיפות של התהליך נעשה אחרי הקריאה ל-()resched. התוצאה: אם תהליך השהה את עצמו, כאשר התהליך יחזור מההשהיה הוא יקבל את העדיפות הנוכחית שלו, ולא את העדיפות שהייתה לו לפני שהוא השהה את עצמו (בזמן שהוא היה מושהה, תהליכים אחרים יכלו לשנות את עדיפות זו).

הערות:

- תהליך חדש נוצר תמיד במצב PRSUSP.
- אסור להשעות את NULL Process!
- ב-suspend יש בדיקה מיותרת: ()isbadpid וגם pid==NULLPROC.

סגירת מערכת ההפעלה

סיום ריצת XINU מתבצע על ידי השיגרה xdone.

הפונקציה מסיימת פעולות קלט/פלט, משחזרת את ווקטור הפסיקות למצבו לפני XINU, מודיעה הודעת סיום ויוצאת ל-DOS.

קריאות מערכת ההפעלה הקשורות לתהליכים

כאשר משתמש רוצה לקבל מידע על התהליכים במערכת ההפעלה, עליו להשתמש בקריאות המערכת הבאות:

- getpid – מחזירה את המציין של התהליך הנוכחי (ערכו של המשתנה currpid).
- getprio – מקבלת pid ומחזירה את העדיפות של התהליך הרצוי.
- chprio – משנה את העדיפות של תהליך.
- getprio():– מקבלת id של תהליך, ומחזירה את עדיפותו. בודקת האם קיים בכלל תהליך בעל ה-id הזה. אם אין תהליך כזה, מבצעים restore() ולאחר מכן מחזירים SYSERR. מתעוררת בעייה, כי ראשית עשינו restore, ולכן מותרות כעת פסיקות. יתכן מצב בו פתאום נוצר תהליך ב-id הנתון וכשחוזרים מ-getprio היא עדיין תחזיר SYSERR.

נפתור בעיה זו, על ידי ההנחה כי הצורה הנכונה של הקריאה לפונקציה היא כרונולוגית, ולכן בגלל שבעצם getprio התבצעה לפני יצירת התהליך החדש – זה בסדר שהחזרנו SYSERR.

אולם, ישנה בעייה נוספת: יתכן שאחרי פקודת ה-restore נכנס תהליך אחר, ומת התליך בעל ה-id הנ"ל, ורק אז אנו מחזירים את עדיפותו. זוהי שגיאה, כי אם התהליך מת, אין לנו גישה לנתונים אלו לגביו.

כדי לתקן את הבאג, יש לשמור את העדיפות במשתנה מקומי לפני ה-restore, ואז להחזיר את ערך המשתנה המקומי. למשל:

```
retValue = pptr->pprio;
restore(ps);
return retValue;
```

בפונקציה chprio באג זה נפתר.

chprio():

בפונקציה זו ישנם שני באגים. הראשון: אמורה להיות בפונקציה זו קריאה ל-resched, ובפועל אין כזו.

השני: לא שינינו את המקום של התהליך בתור ה-ready.

פתרון אפשרי לבאגים: לפני הפקודה restore(ps), נוסיף את הקוד הבא:

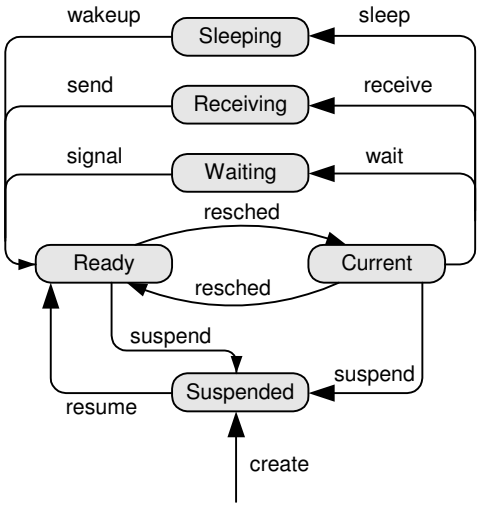
```
if (pptr->pstate == PRREADY)
{
       dequeue(pid);
       ready(pid);
}
resched();
```

proc.h

- NULLPROC – תהליך התמיד מוכן לריצה – בעדיפות מינימלית.
- currpid – משתנה גלובלי שהוא ה-id של התהליך הנוכחי.
- numproc – משתנה גלובלי שהוא מספר התהליכים הרצים כעת.

המצבים האפשריים לתהליכים

השרטוט הבא מסכם את המצבים האפשריים לתהליכים. המילים על הקשתות מציינות את הפונקציות בהן אנו משתמשים כדי לעבור ממצב למצב. פירוט הפונקציות יופיע בהמשך המסמך.



הערה:

כאשר באים לתכנן מערכת הפעלה יש לשים לב ולתכנן מראש את כל המצבים האפשריים עבור תהליכים, וזאת מכיוון שפונקציות רבות של מערכת ההפעלה מגיבות שונה לתהליכים הנמצאים במצבים שונים, ונרצה לאפיין את התגובה שלהן מראש.

טענה חשובה

אם אנחנו יודעים מהו המצב הנוכחי של התהליך, אנו יודעים באיזו רשימה הוא נמצא.

יישום

כל הפונקציות הפועלות על q מניחות שהשדות qnext, qprev של התהליך תקינים. אנו מבססים את ההנחה הזו על ידי שאנו בודקים כל פעם את המצב הנוכחי של תהליך, לפני שאנחנו מבצעים עליו פעולה הקשורה לרשימות.

למשל:

נניח שיצרנו כרגע תהליך. התהליך במצב SUSP ולכן אינו מופיע באף רשימה. אם נקרא כעת לפונקציה kill ונהרוג את התהליך, kill תבדוק את מצב התהליך, ולא תוציא אותו מהתור אם הוא במצב SUSP.

סמפורים

לעיתים כדי לבצע משימה אחת מחלקים אותה למספר תהליכים. למרות שכל תהליך הינו עצמאי, נרצה לאפשר לתהליכים גם לתאם את פעולותיהם, כדי לאפשר למשימה הראשית להתבצע נכונה.

אחד מסוגי התיאום הינו מניעה הדדית.

מערכת ההפעלה מיישמת מניעה הדדית עבור קטעי קוד קריטיים בעזרת disable/restore.

תהליכי משתמש יכולים גם לבצע מניעה הדדית. במקרה של מניעה הדדית של תהליכי משתמש, לא בהכרח כל התהליכים ייחסמו, כפי שקורה ב-disable/restore.

נגדיר **קטע קריטי** בתור קטע קוד בו יש להבטיח מניעה הדדית עם ביצוע של קטעי קוד שמריצים תהליכים נוספים. בעת כניסה לקטע קריטי נקרא לפונקציה wait ובסיומו נקרא ל-signal.

נשים לב שבשפה עילית, גם אם נתון לנו קטע קריטי בן שורה אחת, יש לעטוף אותו בצמד wait/signal, וזאת מכיוון שקטע קריטי בן שורה אחת בשפה עילית, עלול להיתרגם לקטע בן כמה שורות בשפת מכונה, שבין כל אחת מהן יכולה להתרחש החלפת הקשר.

הגדרה

סמפור הוא אמצעי תאום עבור תהליכים, באמצעותו ניתן, בין השאר, להגביל את הגישה בו זמנית אל משאב משותף. פעולת wait גורמת לתהליך לחכות עד אשר יתבצע signal שישחרר אותו. אם בוצע signal לפני כן, התהליך ממשיך בלי לחכות.

אם בוצעו מספר פעולות signal, על כל signal שבוצע תאופשר פעולת wait נוספת ללא המתנה.

בעת אתחול הסמפור ניתן ערך המציין את מספר פעולות ה-wait שיכולות לא לחכות, ללא קריאות ל-signal כלל.

<u>מסקנה:</u> סמפור המאותחל ל-1, וכל קטע קוד קריטי העטוף ב-wait/signal מממשים מניעה הדדית.

מימוש סמפורים ב-XINU

1. לכל סמפור משויך מונה בעל ערכים שלמים.
2. תהליך המבקש לחכות קורא ל-wait(s). כתוצאה מכך יפחת s ב-1. אם הערך המתקבל שלילי התהליך יחכה, אחרת, התהליך ימשיך לעבוד בלי לחכות.
3. תהליך המבקש לשחרר תהליך אחר יקרא ל-signal(s). כתוצאה מכך יגדל ערך s ב-1. אם הערך המתקבל חיובי, אין תהליך מחכה. אם הערך אינו חיובי, משתחרר אחד התהליכים המחכים לסמפור.
4. wait, signal הינם פעולות אטומיות.
5. ערכו המוחלט של מונה הסמפור: אם הוא חיובי - מספר פעולות ה-wait שיכולות להתבצע ללא המתנה. אם הוא שלילי - מספר התהליכים המחכים לסמפור.
6. תהליך שבא לחכות על הסמפור נכנס לסוף התור של הסמפור. כאשר אנו מוציאים (signal) תהליך מהרשימה, אנו מוציאים תהליך מראש התור. כתוצאה מכך: תהליך שנכנס ראשון לרשימת ההמתנה יצא ממנה ראשון, יקבל רשות לבצע את הקטע הקריטי ראשון.

מבנה הנתונים לסמפורים ב-XINU

- **מצב התהליך PRWAIT**:

תהליך המחכה לסמפור יימצא במצב PRWAIT.

- **תור לכל סמפור**:

לכל סמפור נשמר תור FIFO של התהליכים המחכים לו.

- **מצב הסמפור**:

קיימת טבלת סמפורים בה נשמר מצבו של כל סמפור (מונה הסמפור, מיקום התור שלו).

מציין הסמפור הינו האינדקס בטבלה זו.

תור התהליכים המחכים לסמפור נשמר בטבלת התורים ורשומת הסמפור מצביעה אל מקום ראש וזנב התור.

מבנה רשומה בטבלת הסמפורים, הגדרות הקשורות למימוש סמפורים

```
struct sentry {                     /* semaphore table entry         */
       char   sstate;               /* the state SFREE or SUSED      */
       short  semcnt;               /* count for this semaphore      */
       short  sqhead;               /* q index of head of list       */
       short  sqtail;               /* q index of tail of list       */
};

extern struct sentry semaph[];
extern int    nextsem;

#define       isbadsem(s)           (s<0 || s>=NSEM)
```

פונקציות הקשורות לסמפורים

- **wait(sem)** המתנה על סמפור.
- **signal(sem)** שחרור תהליך ממתין על סמפור.
- **screate(count)** מקצה סמפור חדש ומחזירה את האינדקס שלו.
- **sdelete(sem)** מחיקת סמפור והכנסת כל התהליכים הממתינים לו לרשימת ה-ready.
- **scount(sem)** מחזירה את ערכו הנוכחי של הסמפור sem.
- **signaln(s, count)** שקולה לביצוע count פעמים של signal(s) אך יעילה יותר. ניתן להשתמש בפונקציה זו למימוש גרף תלויות.
- **sreset(s, count)** משנה את ערך הסמפור לערך חיובי כלשהו, מעבירה את כל הממתינים עליו לתור ה-ready.

- **wait()**

פעולת wait:

1. בודקת שהסמפור חוקי.
2. מורידה את מונה הסמפור ב-1.
3. אם המונה שלילי:
   - משנה את מצב התהליך הנוכחי ל-PRWAIT.

- מכניסה לרשומת התהליך את מזהה הסמפור.
- מכניסה את התהליך לתור הממתינים של הסמפור.
- קוראת ל-resched כדי להוציא את התהליך הנוכחי מהמעבד.

לא ניתן להפעיל את הפונקציה על סמפור שלא אותחל או שכבר שוחרר. הקריאה ל-resched במהלך הפונקציה הכרחית, כי אחרי שהכנסנו את התהליך הנוכחי לתור המתנה, אין תהליך נוכחי במערכת.

- **signal()**

פעולת signal:

1. בודקת שהסמפור חוקי.
2. מוסיפה 1 למונה הסמפור.
3. אם לפני ההוספה היה הסמפור שלילי, מוציאה את הראשון בתור הסמפור, מכניסה אותו לתור ה-ready וקוראת ל-resched.

לא ניתן להפעיל את הפונקציה על סמפור שלא אותחל או שכבר שוחרר.

הקריאה ל-resched במהלך הפונקציה הכרחית, אולם המערכת תעבוד גם בלעדיה. ההכרחיות היא מכיוון שייתכן שלתהליך החדש עדיפות גבוהה יותר מזה של הנוכחי, ולכן אחרי שנעיר אותו, התהליך החדש הוא זה שצריך לרוץ. עם זאת, אם לא נעיר אותו, הוא יתעורר בעצמו בפעם הפעם ש-resched תיקרא, לכל המאוחר בפסיקת השעון הבאה.

הפונקציות wait, signal מקיימות ומניחות את קיום השמורה הבאה:

כשמונה הסמפורים איננו שלילי, תור הממתינים לסמפור ריק. כאשר המונה שלילי, הערך המוחלט שלו מציין את מספר הממתינים בתור.

כדי לקיים את השמורה:

בגלל ש-wait מקטינה את המונה, היא מוסיפה את התהליך הנוכחי לתור אם המונה שלילי.

בגלל ש-signal מגדילה את המונה, היא מוציאה איבר מהתור במידה והתור איננו ריק.

- **screate()**

פעולת screate:

1. מקבלת כארגומנט את הערך ההתחלתי של המונה.
2. בודקת שהמונה איננו שלילי (כדי להימנע מהפרת השמורה)
3. קוראת לפונקציה newsem() כדי לקבל מזהה של סמפור פנוי, ולתפוס אותו.
4. אם נמצא סמפור פנוי, מעדכנים את המונה שלו.
5. אין צורך באתחול תור הסמפור, וזאת מכיוון שעל מנת לחסוך זמן בשעת יצירת הסמפורים, המערכת מקצה את ראשי וזנבות התורים עבור כל הסמפורים האפשריים כבר בזמן האתחול.

- **newsem()**

פונקציה פנימית בעזרתה screate() מוצאת מציין סמפור פנוי. הפונקציה סורקת את טבלת הסמפורים ומוצאת תא פנוי, או מחזירה SYSERR אם אין כזה. כמו כן הפונקציה מסמנת את התא שנמצא כתפוס.

הפונקציה מסתמכת על כך שבעת אתחול המערכת הוצב הקבוע SFREE לכל רשומות הסמפורים.

הפונקציה מוגדרת כ-LOCAL, כלומר פונקציה פנימית של מנגנון הסמפורים, שלשאר חלקי מערכת ההפעלה אין גישה אליו.

בגלל שהרבה יותר סביר שביטויים שגויים בתוכנית יהיו מתורגמים ל-0 או 1, הפונקציה newsem מתחילה להקצות סמפורים דווקא מסוף מערך הסמפורים, ומקטינה בכך את הסיכוי שתהליך יחכה על הסמפור הלא נכון.

- **sdelete()**

פעולת sdelete:

1. בודקת שהסמפור חוקי ואינו כבר משוחרר.
2. מסמנת בטבלת הסמפורים שהסמפור משוחרר.
3. עוברת על תור הממתינים לסמפור ומעבירה אותם אל תור ה-ready.
4. קוראת ל-resched.

בסוף לולאת ה-while המשחררת את כל התהליכים מבצעים resched, וכך יתכן שאחד מהתהליכים אלו יהפוך לפעיל. נשים לב כי מחיקת סמפור כאשר יש תהליכים המחכים לו היא פעולה מסוכנת. ישנן מערכות הפעלה בהם לא ניתן לשחרר סמפור אשר מחכים עליו תהליכים.

מדוע דווקא אנו שמים את התהליכים בתור ה-ready אם מוחקים את הסמפור, ולא למשל מעבירים את התהליכים למצב PRSUSP? שתי סיבות עיקריות: ראשית, לא ידוע אם יהיה בעתיד תהליך שהולך להעיר את התהליך המושהים. שנית, בדרך כלל קוראים לפונקציה זו במקרים קיצוניים בלבד, שבמילא הולכים להרוג את מערכת ההפעלה ולשחרר משאבים, ולכן הבחירה איפה לשים את התהליכים איננה חשובה כל כך.

- **scount()**

בפונקציה זו יש בעיה: scount מחזירה SYSERR במקרה של שגיאה, אולם SYSERR הוא ערך חוקי למונה סמפור, ולכן במקרה שהפונקציה מחזירה SYSERR, הקורא לפונקציה לא יכול לדעת אם זהו ערך חוקי או לא.

- **signaln()**

פעולת signaln:

1. פעולתה זהה לקריאה ל-signal n פעמים.
2. הקריאה ל-signaln יעילה יותר מכיוון שהיא מבצעת resched רק פעם אחת.
3. הפעולה חיונית למימוש גרף תלויות, וזאת מכיוון שאם היינו משתמשים ב-signal, יתכן שאחד התהליכים שאנו מעירים הוא בעדיפות גדולה מזו של התהליך שקרא ל-signaln, ואז פעולת signaln תיקטע ובמקומה יתחיל לרוץ התהליך שהוער.

שימושים אפשריים לסמפורים

מניעה הדדית:

הסמפור יאותחל ל-1.

כל אחד מהתהליכים A, B, יבצע את הקטע הבא:

```
wait(mutex);
func();
signal(mutex);
```

סנכרון בין תהליכים:

הסמפור יאותחל ל-0.

יצרן	צרכן

produce();
 	
signal(sem);
	wait(sem);
	
	signal(sem);

ניהול מספר מופעים של משאב:

הסמפור יאותחל למספר המופעים.

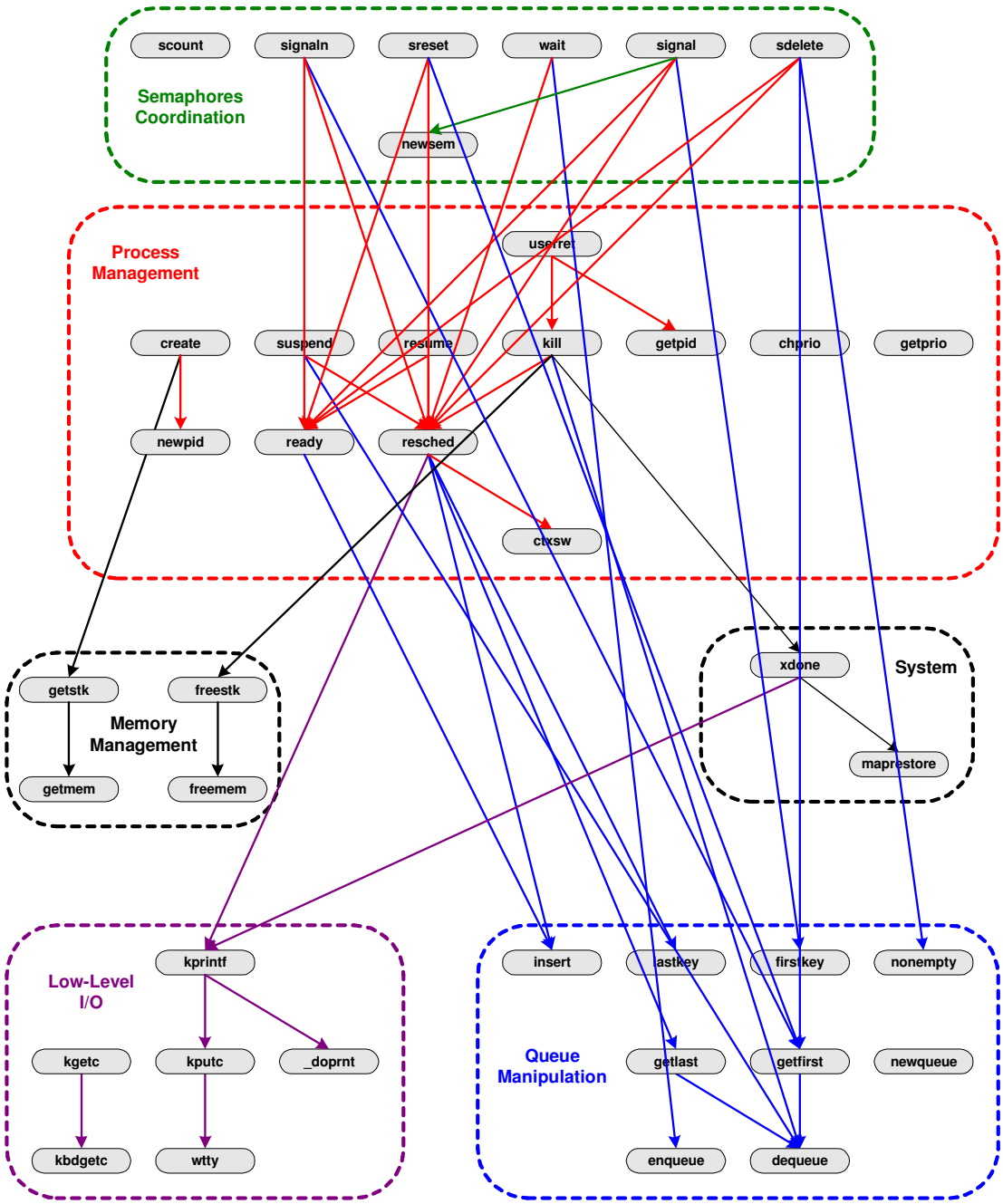
Buffer remove	Buffer insert

wait(avail);
 	
chr = get(array);
 	
signal(places);
	wait(places);
	
	put(array, chr);
	
	signal(avail);

בעיה בפונקציה kill

כאשר אנו מתייחסים לסמפורים, אנו יכולים לראות בעיה הקיימת בפונקציה kill. ייתכן שתהליך יהיה בשני קטעים קריטיים בו זמנית. אם נהרוג את התהליך הזו נהיה בבעיה, כי בטבלת התהליכים נשמר רק אחד מהסמפורים אותם התהליך מחזיק.

סיכום – קשרים בין הפונקציות השונות ב-XINU



הודעות ב-XINU

אופני משלוח הודעות

הודעה ניתן לשלוח במספר אופנים:

1. מתהליך לתהליך - Unicast
2. מתהליך לקבוצה - Multicast
3. מתהליך לכולם - Broadcast
4. מתהליך לתיבת דואר - Mailbox

הודעה ניתן לקבל:

1. מתהליך מסוים - one to one
2. מתהליך כלשהו - any to one

סינכרוניזציה בין משלוח וקבלה

- **גישה סינכרונית:**

- השולח ממתין עד שהמקבל קורא את ההודעה.
- המקבל ממתין עד למשלוח הודעה (אם עדיין לא נשלחה).

- **גישה אסינכרונית:**

- השולח שולח את ההודעה וממשיך בביצוע ללא המתנה.
- המקבל בודק אם קיימת עבורו הודעה וממשיך מיד ללא המתנה.

- **גישות מעורבות:**

- שולח סינכרוני ומקבל אסינכרוני.

- שולח אסינכרוני ומקבל סינכרוני (שולח לא ממתין, מקבל ממתין).

גרסת השולח האסינכרוני והמקבל הסינכרוני נמצאת בשימוש רב, כולל ב-XINU.

סוגי העברת הודעות ב-XINU

- משלוח לא חוסם (על ידי send) - השליחה היא תמיד לתהליך מסויים
- Broadcast, Multicast אינן מיושמות ב-XINU.
- קבלה חוסמת ולא חוסמת (receive, recvclr).

שימושי העברת הודעות

- העברת מסרים.
- תיאום בין תהליכים (בנוסף לסמפורים).

הגבלות

- הודעה היא באורך שני בתים בלבד (integer).
- כשמגיעים מספר מסרים, רק הראשון מתקבל והשאר אובדים.
- חייבים להכיר את התהליך אליו שולחים הודעה.
- תהליך המקבל הודעה איננו יודע מי הוא השולח.

דגש

על send לשמור הודעות במקום בו התהליך הנמען יוכל לקבל אותן.

לא ניתן לשמור הודעות אלו בזיכרון התהליך השולח, מכיוון שהוא עלול להסתיים לפני שהתהליך הנמען ייקרא את ההודעה. כמו כן, לא ניתן לשמור את ההודעות בזיכרון של הנמען, מכיוון שמתן גישה לתהליך שולח לזיכרון של הנמען היא בעיית אבטחה.

הפיתרון ש-XINU מציעה: הוספת שדות בטבלת התהליכים עבור הודעות.

מימוש ב-XINU

- משלוח ההודעות לא חוסם. קבלת ההודעות יכולה להיות חוסמת או לא חוסמת.
- בקבלה חוסמת, אם לא ממתינה לתהליך הודעה, הוא מחכה עד שמגיעה הודעה.
- השולח משאיר את ההודעה בטבלת התהליכים בשדות המתאימים לכך.
- אם השולח רואה כי התהליך המקבל נמצא במצב מחכה להודעה, השולח מחזיר את התהליך המקבל לריצה.
- מצב תהליך מחכה - PRRECV.
- בטבלת התהליכים, לכל תהליך ישנם שדות בשם pmsg, phasmsg. הודעה הנשלחת לתהליך מוכנסת לשדה pmsg ברשומת התהליך. השדה phasmsg מסמן האם יש הודעה ממתינה.

פונקציות הקשורות להודעות

- **send(pid, msg)** שולחת את ההודעה msg אל התהליך pid. אם מחכה כבר הודעה, הפונקציה מחזירה כישלון. קוראת ל-resched לאחר שליחת ההודעה.
- **receive()** מחכה להודעה ומחזירה אותה כשזו מגיעה.
- **recvclr()** אם ישנה הודעה, מחזירה אותה. אחרת, מחזירה OK.
- **sendf(pid, msg)** שולחת את ההודעה msg אל התהליך pid. אם מחכה כבר הודעה, היא נדרסת על ידי ההודעה החדשה.
- **sendn(pid, msg)** שולחת את ההודעה msg אל התהליך pid. אם מחכה כבר הודעה, הפונקציה מחזירה כישלון. איננה קוראת ל-resched.

- **send()**

פעולת send:

1. ניגשת לרשומת התהליך המקבל.
2. אם קיימת הודעה שעדין לא טופלה מחזירה שגיאה.

3. מסמנת ברשומה שיש הודעה לא מטופלת ומכניסה את ההודעה.
4. אם התהליך היה במצב המתנה להודעה, מעבירה אותו לתור ה-ready וקוראת ל-resched.

- **sendf()**

פועלת בדיוק כמו send, אולם כותבת את ההודעה בכל מקרה, גם אם הייתה רשומה הודעה קודמת שלא טופלה. sendf משמשת לשליחת הודעות דחופות שאסור שיתעלמו מהן.

חשוב לשים לב שבדומה ל-send, גם sendf מעירה את התהליך עליו היא פועלת.

- **sendn()**

פועלת בדיוק כמו send, אולם אם התהליך המקבל היה במצב המתנה להודעה, sendn מכניסה אותו לתור ה-ready אך איננה קוראת ל-resched.

Sendn משמשת לשליחת הודעות בזמן פסיקות, בהן אסור לקרוא ל-resched.

- **recvclr()**

נשים לב שפונקציה זו הינה בעייתית כאשר ההודעה היא OK.

לא ניתן להבדיל בין המקרה בו ההודעה היא OK, לבין המקרה שאין הודעה מחכה לתהליך.

ניהול זיכרון

מבנה הזיכרון ב-PC

זיכרון חופשי	מחסנית	bss	data	text
אזור המיועד להקצאת זיכרון דינאמי	מחסנית התוכנית	משתנים ומבנים גלובליים לא מאותחלים	משתנים ומבנים גלובליים מאותחלים	קוד התוכנית
סגמנט נתונים ds, ss				**סגמנט קוד - cs**

מבנה הזיכרון ב-XINU

Heap – שטח להקצאת זיכרון לתהליכים				מחסנית מקורית	bss	data	text
שטח פנוי להקצאות	מחסנית 2	מחסנית 1	שטחים מוקצים דינמית	מחסנית של תהליך NULL	משתנים ומבנים גלובליים לא מאותחלים	משתנים ומבנים גלובליים מאותחלים	קוד התוכנית
סגמנט נתונים ds, ss							**סגמנט קוד - cs**

ניהול זיכרון ב-XINU

- ב-XINU כל הזיכרון ניתן לגישה מכל התהליכים.
- תהליך המבקש בלוק זיכרון מקבל מצביע לבלוק הזיכרון שהוקצה עבורו. לא מונעים מתהליכים אחרים לגשת אל אותו הזכרון.
- תהליך יכול להעביר לתהליך אחר מצביע לבלוק מוקצה, כדי שהתהליך האחר יוכל לגשת לאותו המקום בזיכרון ולשלוף ממנו נתונים.

זיכרון להקצאה דינמית

- בלוק הזיכרון הפנוי במערכת איננו בהכרח רציף. אם הקצנו בלוקים, ולאחר מכן שחררנו חלק מהם, יתכנו "חורים" בזיכרון הפנוי.
- כדי להקצות שטח זיכרון חדש עלינו לדעת היכן השטחים הפנויים. לשם כך אנו מחזירים רשימה מקושרת שלהם. על ראש הרשימה מצביעה הרשומה memlist.
  נדגיש כי memlist הוא משתנה המצביע אל תחילת הזיכרון הפנוי, ואיננו חלק מהזיכרון הפנוי.
- בכל שטח פנוי אנו מחזיקים את גודלו ואת המצביע את השטח הפנוי הבא, לפי המבנה mblock.
  mblock הוא מבנה המכיל שני שדות: next, mlen. גודלו של מבנה זה הוא ארבעה בתים. מכאן:
  - נלקחים 4 בתים מכל שטח פנוי לניהול הרשימה.
  - כל שטח פנוי צריך להיות בגודל של לפחות 4 בתים.
  - כל הקצאה צריכה להיות לפחות בגודל 4 בתים, וזאת על מנת שכשנפנה את הבלוק, יהיו לפחות 4 בתים בהם נשתמש לרשימה המקושרת.

המבנה:

```
typedef struct mblock
{
       struct mblock *mnext;
       word   mlen;
} MBLOCK;
```

- משיקולי נוחות, נבחר ב-XINU שכל ההקצאות יהיו בכפולות של 4 בתים שכתובתם היא כפולה של 4. כאשר מבקשים הקצאת שטח זיכרון שאינו כפולה של 4 בתים, יוקצה שטח זכרון בגודל של הכפולה הבאה של 4 בתים.
- כל השטחים הפנויים מוצבעים על ידי רשימת הפנויים, בסדר מונוטוני לפי מיקומם בזיכרון. שטחים מוקצים לא יופיעו ברשימה.
- לא יופיעו ברשימה שני שטחים פנויים רצופים. במקרה כזה ניתן לאחד אותם לשטח גדול יותר.

שיטות הקצאה

1. **First Fit:**

הקצאת הזיכרון הינה מתוך השטח הפנוי המתאים הראשון שנמצא ברשימה. הסריקה מתחילה בתחילת רשימת הפנויים, וברגע שנמצא שטח ריק גדול מספיק, הוא נתפס לצורך הקצאה.

זוהי השיטה בה מערכת ההפעלה XINU משתמשת.

2. **Next Fit:**

ההקצאה הנה בדומה ל-First Fit אך מתוך השטח הפנוי הבא. שומרים מצביע למקום ממנו התקיימה ההקצאה האחרונה, ומתחילים לחפש ממקום זה באופן ציקלי על כל הרשימה.

3. **Best Fit:**

סורקים את כל רשימת הפנויים וההקצאה היא מתוך השטח הפנוי הקטן ביותר האפשרי.

4. **Worst Fit**:

סורקים את כל הרשימה וההקצאה היא מתוך השטח הפנוי הגדול ביותר.

בכל מקרה לאחר בחירת השטח הפנוי להקצאה, מקצים ממנו שטח בגודל הרצוי. אם נותר חלק מהשטח לא מוקצה, מחזירים אותו לרשימה בגודלו הנותר.

עבור כל אחת משיטות ההקצאה, ניתן למצוא דוגמאות בהן היא עדיפה על פני השיטות האחרות.

שחרור שטחי זיכרון

- כאשר משחררים שטח זיכרון, נבדוק האם השטחים השכנים לו פנויים. אם כן, נאחד את הקטע המשתחרר עם שכניו, על מנת לקבל את השטח הפנוי הגדול ביותר האפשרי.
בצורה זו אנו מקטינים את אורך הרשימה, וגם מאפשרים להקצות קטעי זיכרון גדולים יותר.
- מצבים אפשריים כאשר אנו משחררים בלוק זיכרון:
    - הזיכרון תפוס משני צדדי הבלוק.

- הזיכרון פנוי משני צדדי הבלוק.
- הזיכרון פנוי באחד מצדדי הבלוק.
- מקרי קצה: הבלוק נמצא בהתחלת הרשימה המקושרת, או בסופה.

פונקציות הקשורות לניהול זיכרון

• **getmem(nbytes)**	הפונקציה מקצה שטח זיכרון באורך nbytes מתוך ה-heap.
• **freemem(block, nbytes)**	הפונקציה משחררת שטח זיכרון לשימוש חוזר. הפונקציה מקבלת כפרמטר מצביע אל התחלת הבלוק וגם את גודל הבלוק לשחרר.

שמורה המנוהלת על ידי הפונקציות לטיפול בזיכרון:

בלוקים ברשימת הזיכרון הפנוי ממוינים בסדר עולה לפי כתובתם.

- **getmem()**

המערכת אינה יכולה להבטיח שזיכרון שהוקצה לתהליך בעזרת getmem ישתחרר כשתהליך מסתיים, וזאת מכיוון שהיא איננה שומרת איזה זיכרון שייך לכל תהליך.

לכן: תהליך חייב לשחרר את כל הזיכרון שהוא הקצה מהערימה לפני שהוא מסתיים.

הפונקציה משתמשת במקרו ()roundew, המקבל מספר ומחזיר את העיגול שלו כלפי מעלה למספר הבא המתחלק ב-4. המקרו מוסיף 3 למספר, ואז מחזיר ממנו את שארית החלוקה שלו ב-4.

מכיוון שהרשימה המקושרת היא רשימה חד כיוונית, דרושים שני מצביעים כדי לסרוק ולמצוא את הבלקו המבוקש.

כאשר נמצא בלוק בגודש המבוקש: אם הבלוק בדיוק בגודל הרצוי, getmem מוחקת את הבלוק מרשימת הבלוקים הפנויים ומחזירה את כתובתו. אחרת, הפונקציה מחזירה בלוק בגודל nbytes מתחילת הבלוק הפנוי שמצאנו, ומשרשרת את שאר הבלוק הפנוי בחזרה אל רשימת הפנויים.

כאשר אנו מפצלים בלוק בצורה זו, המשתנה הפנימי leftover מצביע אל תחילת הקטע אותו נרצה לשרשר. החישוב כדי למצוא את leftover הוא פשוט: leftover נמצא במרחק nbytes בתים מתחילת הבלוק.

נשים לב שהוספת nbytes ל-p לא תשיג את התוצאה הרצויה, עקב אריתמטיקת המצביעים של שפת C. לכן, כדי לבצע חישוב מספרי, ראשית אנו ממירים את p ל-(*char), ולאחר מכן אנו ממירים את p בחזרה ל-(*struct mblock).

בפונקציה יש באג: הפונקציה בודקת אם כמות הזיכרון המבוקשת שווה ל-0, ואז מחזירה שגיאה, אבל היא אינה בודקת מקרה בו כמות הזיכרון המבוקשת קטנה מ-0.

- **freemem()**

בדומה ל-getmem, שני המצביעים p, q רצים על רשימת הזיכרון הפנוי. אנחנו עוצרים כאשר כתובת הזיכרון אותו רוצים לשחרר נמצאת ביניהם.

פונקציה זו מורכבת יותר מ-getmem, מכיוון שכאשר נשחרר זיכרון, אם ייווצרו מספר בלוקים ריקים ברצף נרצה לאחד אותם לבלוק גדול אחד.

q מצביע אל הבלוק הפנוי שלפני הבלוק שרוצים לשחרר. p מצביע אל הבלוק הפנוי שאחרי הבלוק שרוצים לשחרר.

אם q הוא NULL, אין בלוק פנוי לפני. אם p הוא NULL, אין בלוק פנוי אחרי.

בפונקציה יש באג: הפונקציה בודקת אם כמות הזיכרון המבוקשת שווה ל-0, ואז מחזירה שגיאה, אבל היא אינה בודקת מקרה בו כמות הזיכרון המבוקשת קטנה מ-0.

בפונקציה זו יש גם באג נוסף: הפונקציה איננה משרשרת את הזיכרון הפנוי כראוי.

נביט כעת בקוד הפונקציה, וננתח מה היא צריכה לעשות ומה היא עושה, על מנת לזהות את הבאג.

```
/*-------------------------------------------------------------------------
 *  freemem  --  free a memory block, returning it to memlist
 *-------------------------------------------------------------------------
 */
SYSCALL       freemem(block, size)
char *block;
word size;
{
        int     ps;
        struct mblock *p, *q;
        char    *top;

        size = roundew(size);
        block = (char *) truncew( (word)block );
                                        (1)
        ┌──────────────────────────────────────────────────────────────────┐
        if ( size==0 || block > maxaddr || (maxaddr-block) < size ||
                        block < end )
                return(SYSERR);
        disable(ps);
        (char *)q = NULL;
                                (2)
        ┌──────────────────────────────────────────┐
        for( p=memlist.mnext ;
                        (char *)p != NULL && (char *)p < block ;
                        q=p, p=p->mnext )
                                ;
                                        (3)
        ┌──────────────────────────────────────────────────────────────────┐
        if ( (char *)q != NULL && (top=(char *)q+q->mlen) > block
                    || (char *)p != NULL && (block+size) > (char *)p ) {
                                        └──────────────────────────┘
                                                (4)
                restore(ps);
                return(SYSERR);
        }
                                (5)
        ┌──────────────────────────────────────┐
        if ( (char *)q != NULL && top == block )
                q->mlen += size;
        (6)
        else {
                ((struct mblock *)block)->mlen = size;
                ((struct mblock *)block)->mnext = p;
                memlist.mnext = (struct mblock *)block;
                (char *)q = block;
        }
        /* Note that q != NULL here */
                                        (7)
        ┌──────────────────────────────────────────────────────────────────┐
        if ( (char *)p != NULL
                        && ((char *)q + q->mlen) == (char *)p ) {
                q->mlen += p->mlen;
                q->mnext = p->mnext;
        }
        restore(ps);
        return(OK);
}
```

ננתח כעת את הפונקציה.

1. **תקינות הפרמטרים:**

בשורה (1) אנו בודקים שלא גלשנו מגבולות הזיכרון הפנוי להקצאה – מלמעלה ומלמטה.

2. **סריקת רשימת הבלוקים הפנויים:**

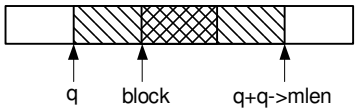
שורה (2): שני המצביעים q ,p מטיילים על הרשימה.

בסוף הלולאה: q מצביע אל הבלוק הפנוי שלפני הבלוק שרוצים לשחרר. p מצביע אל הבלוק הפנוי שאחרי הבלוק שרוצים לשחרר.

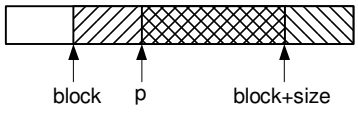
3. **בדיקת חוקיות הבלוקים:**

לאחר שקבענו את ערכי q, p, נבדוק שחוקי לשחרר את הבלוק המבוקש. ישנם שני מצבים בהם לא חוקי לשחרר את הבלוק:

אפשרות 1: הבלוק שהמשתמש רוצה לשחרר זהו בלוק זיכרון חופשי.



אפשרות 2: הבלוק שהמשתמש רוצה לשחרר גדול מדי, והוא חודר לתוך האזור הפנוי הבא.



שורה (3) מטפלת באפשרות מספר 1.

שורה (4) מטפלת באפשרות מספר 2.

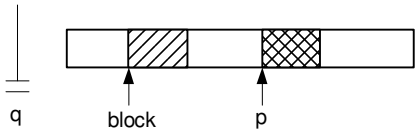
4. **מצבים אפשריים עבור מיקום הבלוק:**

- q==NULL וגם p==NULL, כלומר אין אף בלוק זיכרון פנוי.

מצב זה מטופל בשורה (6):

- שמים בתחילת block את הרשומה החדשה.
- memlist מצביע כעת על block.

- q==NULL וגם p!=NULL, כלומר אין בלוק פנוי לפני הבלוק שרוצים לשחרר, אולם יש בלוק פנוי אחריו.



טיפול ראשון למצב זה בשורה (6):

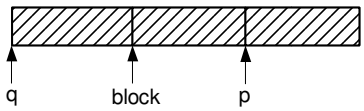
- שמים בתחילת block את הרשומה החדשה.
- memlist מצביע כעת על block.

כעת נבדוק האם צריך לאחד את הבלוק הפנוי החדש עם הבלוק הפנוי שאחריו או שלא.

אם לא צריך לאחד את הבלוקים, סיימנו. אחרת, שורה (7) דואגת לאיחוד.

- q!=NULL וגם p!=NULL. במקרה זה ישנן מספר אפשרויות:

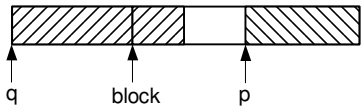
אפשרות 1: משלושת הבלוקים נוצר כעת בלוק גדול אחד.



טיפול ראשון בשורה (5): מרחיבים את q לכלול גם את block.

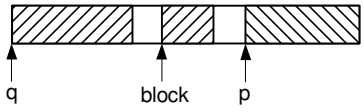
טיפול שני בשורה (7): מרחיבים את q לכלול גם את p, ומוציאים את p מהרשימה.

אפשרות 2: block צמוד ל-q אבל לא ל-p.



שורה (5) מטפלת במקרה זה. q גדל כדי לכלול את block.

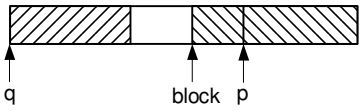
אפשרות 3: block נמצא בין q ל-p, אבל הוא איננו צמוד לאף אחד מהם.



במקרה זה אנו רואים את הבאג שבפונקציה freemem:

שורה (6) גורמת ש-block יהיה הבלוק הראשון ברשימה. כל הזיכרון הפנוי שהיה לפני block נאבד.

אפשרות 4: block לצמוד ל-p אבל לא ל-q.



גם מקרה זה לא מטופל כראוי על ידי freemem. גם במקרה זה, הזיכרון שלפני block נאבד.

שברור - Fragmentation

שברור הוא מצב בו יש זיכרון פנוי במערכת, אך לא ניתן להשתמש בו מסיבות שונות - לרוב מפני שהוא נמצא בזיכרון בצורה שאינה רציפה - כחתיכות קטנות ורבות של זיכרון.

נחלק את השברורים לשני סוגים: שברור פנימי ושברור חיצוני.

שברור חיצוני: מצב בו ישנם שטחי זיכרון פנוי, אולם הם קטנים מדי להקצאה.

התוכנית הבאה תדגים שברור חיצוני. נניח שהזיכרון הפנוי הוא בגודל 4000 בתים.

```
for (i = 0; i < 1000; ++i) arr[i] = getmem(4);
for (i = 0; i < 500; ++i) freemem(arr[i*2], 4);
p = getmem(12);
```

ההקצאה getmem(12) תיכשל. למרות שבשלב זה ישנם 2000 בתים פנויים, ההקצאה בגודל 12 לא יכולה להצליח עקב שברור חיצוני.

שברור פנימי: מצב בו ישנם שטחי זיכרון שהוצאו מרשימת הפנויים כתוצאה מהקצאה, אך בהכרח לא נמצאים בשימוש. נשים לב שב-XINU כל הקצאה היא בכפולות של 4, לכן תהליך המבקש, למשל, הקצאה של 9 בתים, יקבל מצביע אל הזיכרון המוקצה, ויוכל להשתמש ב-9 בתים מהמקום שהוחזר לו. למעשה, ההקצאה הייתה בגודל 12, ושלושת הבתים שהוקצו מעבר לדרישה אינם פנויים לשימוש אף אחד מהתהליכים. התוכנית הבאה תדגים שברור פנימי. נניח כי הזיכרון הפנוי הוא בגודל 4000 בתים.

```
for (i = 0; i < 400; ++i)
{
       arr[i] = getmem(10);
       if (arr[i] == NULL) break;
}
printf("Only %d bytes out of 4000 allocated\n", 10 * i);
```

לאחר 333 הקצאות הלולאה תיפסק, וזאת כי בפעול ההקצאות היו בגודל של 12 בתים, ולכן רק 333 הקצאות יכולות להצליח בפועל.

עבור המבקש הוקצו רק 3330 מתוך 4000 בתים שביקש (שהיו פנויים).

בעיות אפשריות הקשורות לזיכרון

בעיה 1 - תהליך הניגש למשתנים הנמצאים על מחסנית של תהליך אחר

דוגמא:

```
int xmain()
{
       int sem = screate(0);
       if (sem == SYSERR) return -1;
       resume(create(proc, INITSTK, INITPRIO, "proc", 1, &sem));
       signal(sem);
       return 0;
}

int proc(int *p_sem)
{
       ...
       wait(*p_sem);
       ...
}
```

המחסנית של xmain משתחררת ואיתה גם sem כאשר xmain מסתיימת. proc יכול לגשת אל זיכרון שהוא כרגע לא מוקצה.

נניח שהיינו מנסים לפתור את הבעיה בעזרת הקצאה דינאמית, אז הייתה מתעוררת בעיה חדשה: מי ישחרר את ההקצאה כשאף אחד לא ישתמש במשתנה. כמו כן עולה השאלה: כיצד ניתן לדעת מתי אף אחד לא משתמש בו יותר.

ניהול פסיקות

נבדיל בין שלושה סוגי פסיקות:

- פסיקות חומרה - פסיקות שעון, ערוץ תקשורת וכו'.
- פסיקות תוכנה - נוצרות על ידי המשתמש.
- חריגות - דוגמת חילוק באפס או נפילת מתח.

פסיקות חומרה

- פסיקות חומרה (Interrupt) הן אמצעי המאפשר להתקנים חיצוניים (למשל מקלדת, מסך, דיסק קשיח) לקבל ולתת שירותים למעבד.
- מנגנון הפסיקות נועד להפריד את הטיפול באירועי חומרה מקוד המשתמש.
- המנגנון מאפשר התקשרות המעבד עם התקנים (לשם נתינה וקבלת שרותים מהתקנים, קבלת מידע על מצב ההתקנים) באופן אסינכרוני וללא בדיקות חוזרות ונשנות (למשל polling)

פסיקות ב-XINU

במכונת ה-PC, כאשר מתרחשת פסיקה, המעבד דוחף באופן אוטומטי את FLAGS ואת CS:IP אל המחסנית. כאשר חוזרים מהפסיקה חוזרים בעזרת פקודה מיוחדת, האומרת למעבד לשלוף את שני ערכים אלו.

מכיוון שקריאות לפסיקות מתבצעות בעזרת מנגנון קפיצה מיוחד, לא ניתן בדרך כלל לכתוב שגרת פסיקה בשפות עיליות כמו C. עם זאת, כתיבה באסמבלר תהפוך את מערכת ההפעלה קשה להבנה ולשינוי.

הפתרון של XINU: לנתח את הפסיקות בשני שלבים. כל הפסיקות יופנו לפונקציה קטנה באסמבלר, שתכונה משגר הפסיקות - "Interrupt Dispatcher". הפונקציה אחראית לשמירת ושחזור הרגיסטרים.

- פונקצית האסמבלי המופעלת בעת הפסיקות והקוראת לפונקציות בשפת C הינה הפונקציה intcom.
- על מנת לתמוך בקריאה לפונקציות המקוריות, נשמור מידע עבור כל אחת מהפסיקות בטבלה, בה קיים עבור כל פסיקה דגל האומר האם לקרוא לקוד ה-ROM-BIOS או לא.

ווקטור הפסיקות

- ווקטור הפסיקות הינו טבלה הכוללת לכל מספר פסיקה את הכתובת של קוד הפסיקה שצריך להתבצע.
- טבלה זו נמצאת במקום קבוע בזיכרון.
- התוכנה יכולה לעדכן את הכתובות של קודי הפסיקה בווקטור הפסיקות, בעוד המעבד ניגש לכתובות אלו רק על מנת למצוא את הכתובת של קוד הפסיקה.
- ווקטור הפסיקות מאותחל בזמן עליית מערכת ההפעלה. עד אתחול ווקטור הפסיקות, אין לאפשר ביצוע פסיקות, ודגל הפסיקות צריך להיות קבוע. בזמן אתחול ווקטור הפסיקות מוצבת הכתובת של קודי הפסיקה השונים לתוך הכניסות המתאימות בווקטור הפסיקות.

עדכון ווקטור הפסיקות

- לעיתים נרצה לשנות את קודי הפסיקה של פסיקות מסויימות בזמן ריצת מערכת ההפעלה. שינוי ההצבעה בווקטור הפסיקות נקרא **Revectoring**.
- במקרים אחרים נרצה להוסיף עוד פעולה לביצוע בזמן פסיקה מסוימת, אך לא לבטל את הפעולות המתבצעות בקוד הפסיקה הקיים. לשם כך, נבצע Revectoring לקוד הנוסף, והקוד הנוסף יהיה אחראי לקרוא לקוד הישן (בדומה לקריאה לפונקציה).

יישום

המערך sys_imp כולל את כל המידע הנדרש לפסיקות XINU.

לכל פסיקה מוקצאת רשומה הכוללת את המידע הבא:

- פקודת המכונה call intcom.
- כתובת קוד ה-BIOS שהוצבע במקור מווקטור הפסיקות.
- כתובת פונקצית הפסיקה של XINU.

- מספר הפסיקה בווקטור הפסיקות.
- זיהוי פנימי של ההתקן.
- דגל המסמן האם לקרוא לקוד ה-BIOS לפני הקריאה לקוד ה-XINU.
- גודל כל רשומה מסוג intmap הינו 14 בתים.

מבנה רשומה:

```
struct intmap {             /* device-to-interrupt routine mapping  */
       char   ivec;         /* interrupt number                     */
       char   callinst;     /* the call instruction                 */
       word   intcom;       /* common interrupt code                */
       word   oldisr_off;   /* old int. service routine offset      */
       word   oldisr_seg;   /* old int. service routine segment     */
       int    (*newisr)();  /* pointer to the new int. ser. routine */
       word   mdevno;       /* minor device number                  */
       word   iflag;        /* if nonzero, call the old isr         */
};
```

קבועים והגדרות הקשורים לטבלה זו:

```
extern struct intmap far *sys_imp;       /* pointer to intmap table */
extern int    nmaps;              /* number of active intmap entries*/

#define       isbaddev(f)   ( (f)<0 || (f)>=NDEVS )

/* In-line I/O procedures */

#define       getchar()     getc(CONSOLE)
#define       putchar(ch)   putc(CONSOLE,(ch))
#define       fgetc(unit)   getc((unit))
#define       fputc(unit,ch)      putc((unit),(ch))

extern int    _doprnt();          /* output formatter              */
```

- ווקטור הפסיקות מצביע על פקודת call intcom שברשומה המתאימה במערך sys_imp. לכן בעת פסיקה תקרא intcom כפונקציה מתוך הפסיקה.

- intcom היא פונקציה כללית המשרתת את כל הפסיקות. אנו יודעים לאיזו פסיקה לקרוא לפי המיקום של intcom. הבית שאחרי פקודה זו נמצא על המחסנית ככתובת החזרה, בכתובת ss:bp+2.
- על ידי ביצוע mov bx, [bp+2], אנו מקבלים ב-bx את "כתובת החזרה" המצביעה לתוך רשומת הפסיקה. כתובת שגרת הפסיקה המקורית נמצאת ב- $(cs:[bx]):(cs:[bx+2])$.
- כתובת שגרת הפסיקה של XINU היא cs:[bx+4]. כתובת mdevno היא cs:[bx+6].
- כתובת iflag המציין האם לקרוא לשגרה המקורית היא cs:[bx+8].
- בסוף פעולתה, intcom מוציאה את כתובת החזרה מהמחסנית, ומשתמשת ב-iret כדי לחזור לנקודה בה הייתה לפני הקפיצה לטבלה.
- טבלת הפסיקות נמצאת בסגמנט התוכנית, cs, שהינו לא נגיש עבור תוכניות C רגילות. מכיוון שאנו רוצים לאתחל את המערך מתוך C, מוגדר המשתנה sys_imp המכיל מצביע (סגמנט + offset) אל הטבלה.

intcom()

- קוראת לקוד הפסיקה הישן במידה ש-iflag דולק.
- מכבה את דגל הפסיקות, כדי לוודא שקוד הפסיקה הישן לא הדליק אותו.
- שומרת את ערכי כל הרגיסטרים.
- אם הפסיקה ארעה כאשר התבצע קוד של XINU:
  - קוראת לקוד הפסיקה של XINU.
- אם הפסיקה ארעה מתוך קוד שאינו של XINU
  (למשל בזמן ביצוע פסיקת BIOS- ROM כאשר דגל הפסיקות דולק):
  - נשמרת כתובת המחסנית הישנה ומותקנת מחסנית חדשה בתוך סגמנט הנתונים
    של XINU (כי יתכן שהמחסנית הישנה אינה בסגמנט זה, ואילו XINU דורש SS = DS סגמנט הנתונים שלו).
  - נמנעת החלפת תהליכים ע"י איפוס pcxflag.
  - נקרא קוד הפסיקה של XINU.
  - משוחזרים pcxflag והמחסנית הישנה.

- משוחזרים כל הרגיסטרים.
- כתובת החזרה מ- intcom מוצאת מהמחסנית ומתבצע iret.

<u>הערה:</u> ב-intcom, קביעת ה-pcxflag ל-0 נעשית אחרי החלפת המחסנית ולא לפני, אולם זו איננה בעיה מכיוון שהפסיקות מכובות בקטע זה, ולכן לא תתכן החלפת תהליכים.

מבנה המחסנית בזמן הקריאה ל-intcom

Stack Offset	Stack Contents
0	es
1	ds
2	di
3	si
4	dx
5	cx
6	bx
7	ax
8	bp
9	ret address to intmap table, offset 4
10	IRET offset address
11	IRET segment address
12	IRET flags
13	user code

- intcom מתחילה את פעולתה בדחיפת bp, ax, bx אל המחסנית.

אתחול ווקטור הפסיקות

- אתחול רשומות הפסיקה ב-sys_imp נעשה לכל פסיקה, בה XINU משתמש, בנפרד.
- האתחול כולל שינוי הכתובת בוקטור הפסיקות והגדרת רשומת הפסיקה ב-sys_imp.
- אתחול זה נעשה על ידי הפונקציה mapinit.
- תהליך האתחול:
  - שמירת כתובת ה-isr הקודם ברשומת הפסיקה.
  - שינוי ווקטור הפסיקות שיצביע על call intcom ברשומת הפסיקות.
  - אתחול שדות נוספים, ביניהם iflag.

שחזור ווקטור הפסיקות

- בסיום XINU משוחזרות הכתובות של הפסיקות בווקטור הפסיקות כדי לאפשר לDOS לפעול. הפעולה מתצבעת על ידי הפונקציה maprestore.
- בפונקציה זו יש באג: ב-XINU ייתכן מצב שבו נחליף את אותה פסיקה כמה פעמים, שיהיו כמה פונקציות שייטפלו באותה פסיקה. במקרה זה maprestore לא עובדת נכון: אנו משחזרים את אותו ווקטור פסיקה כמה פעמים, אולם לא נשחזר נכון את ווקטור הפסיקה כפי שהיה לפני XINU. פתרון: לשחזר את הווקטור מסוף המערך sys_imp לתחילתו, ולא כפי שנעשה - מתחילתו לסופו.

Reentrant code והימנעות מהחלפת תהליכים

נאמר שפונקציה היא reentrant אם מספר קריאות לפונקציה ממספר תהליכים יכולים לבצע את הקוד שלה בו זמנית. מערכות הפעלה המנהלות מספר תהליכים בו זמנית מלאות בפונקציות מסוג reentrant, למשל, ב-XINU resched היא פונקציה כזו. שגרות reentrant לרוב משתמשים במשתנים מקומיים המוקצים על מחסנית המשתמש. גישות למשתנים גלובליים נעשים על ידי פונקציות כאלו מתוך קטעים העטופים ב-disable/restore.

שגרות BIOS אינן reentrent. מכיוון שקריאות לשגרות BIOS לעיתים רבות מפעילות שוב את הפסיקות, הן אינן יכולות לסמוך על עצמן שהן יהיו מסוגלות להגן על המשתנים הגלובליים שלהן במידה ואותה פסיקה תתרחש ממספר תהליכים. ב-XINU פונקציה אחת בלבד אחראית על החלפת

התהליכים - resched. הדרך היחידה ש-resched יכולה להיקרא מתוך שיגרת BIOS היא עקב פסיקות. כמו כן, ב-XINU הפונקציה intcom מקצה מחסנית מקומית בה שגרת ה-BIOS שומרות את הנתונים שלהן.

לפיכך - **אסור להחליף תהליכים כאשר אנו בזמן פסיקות BIOS.**

המשתנה pcxflag משמש כעוצר החלפת התהליכים. כאשר הדגל קבוע ל-0, החלפת התהליכים של XINU מבוטלת. כאשר המשתנה הוא 1, החלפת התהליכים נעשית כרגיל.

לפיכך intcom קובעת דגל זה ל-0 כאשר המחסנית המקומית משמשת פסיקת BIOS.

שליטת תהליך על החלפת התהליכים

לעתים תהליך ירצה למנוע החלפת תהליכים, ואם זאת להשאיר את הפסיקות פעילות. לשם כך נועדים המקרואים xdisable, xrestore. מקרואים אלו פועלים בדומה ל-disable/restore, רק שהם שומרים את ה-pcxflag. אם החלפת התהליכים נמנעת בצורה זו, והתהליך הנוכחי קורא ל-resched (בצורה לא ישירה כלשהי, למשל בעזרת send או signal), אזי resched חוזרת בלי שהיא מבצעת דבר.

את החלפת התהליכים נעשית על ידי כך שהתהליך משנה את המצב שלו וקורא ל-resched, למשל על ידי הפקודות wait, receive, אז המערכת נותרת במצב בלתי אפשרי.

מניעת החלפת תהליכים חייבת לקיים את הכלל הבא: **תהליך המונע החלפת תהליכים יכול לקרוא אך ורק לפונקציות המשאירות את התהליך במצב current.**

שגרות BIOS אינן יכולות לשנות את המצב של התהליך הנוכחי, ולכן הן בטוחות לשימוש עם מניעת החלפת תהליכים.

כללים לניתוח פסיקות

פסיקות עלולות לשנות מבני נתונים גלובליים, דוגמת חוצצי ה-I/O. אי לכך, הפונקציות צריכות למנוע מתהליכים אחרים להפריע להן. אנו משיגים זאת על ידי הרצת שגרות הפסיקות כאשר הפסיקות כבויות.

XINU מממשת את הכלל הבא: **הפסיקות יהיו מכובות כאשר המשגר קורא לשגרות בשפה עילית. השגרה בשפה עילית חייבת לדאוג להשאיר את הפסיקות כבויות עד שהיא מסיימת לעדכן את מבני הנתונים הגלובליים.**

כאשר כותבים פסיקות, עלינו להתייחס למספר נקודות:

- הפסיקה איננה יכולה להשאיר את הפסיקות מבוטלות לזמן רב, שכן אז התקנים לא יפעלו כראוי. (המעבד יתעלם מהפסיקות שהם מבקשים).

- הפסיקות יכולות להתרחש במהלך ריצת כל אחד מהתהליכים במערכת. בפרט, פסיקות חייבות להיות מתוכננות לרוץ כראוי אפילו אם הן מורצות מהתהליך ה-NULL.

לכן, עליהן לשמור את הכלל הבא: **שגרות פסיקה יכולות לקרוא לפונקציות המשאירות את התהליך הרץ במצב current או ready בלבד.**

לכן, פסיקות יכולות להשתמש בפקודות כגון send או signal, אולם לא בפקודות כגון wait.

החלפת תהליכים תוך כדי ניתוח פסיקה

פסיקות אינן יכולות לאפשר פסיקות נוספות במהלך ריצתן. כמו כן הן חייבות להשאיר את מבני הנתונים הגלובליים במצב תקין לפני שתתחרש החלפת תהליכים. לכאורה נראה שאסור לאפשר החלפת תהליכים מתוך פסיקות, מכיוון שמעבר לתהליך אחר עלולה להפעיל את הפסיקות, בו שוב תתרחש הפסיקה, וכך הלאה עד שנגלוש ממחסנית הפסיקות.

עם זאת, אפשור החלפת תהליכים זוהי אפשרות חשובה, מכיוון שזו הדרך היחידה בה פסיקות יכולות להשפיע על התהליך הנוכחי.

אפשור החלפת תהליכים בזמן פסיקות היא פעולה בטוחה בהנחה ש: 1) שגרות פסיקה משאירות נתונים גלובליים במצב חוקי לפני החלפת התהליכים. 2) פונקציה איננה מפעילה פסיקות אם לא היא זו שכיבתה אותן.

נשים לב שעד כה קיימנו את טענה זו מבלי לנסח אותה. השילוב של disable/restore בכל שגרת מערכת מממש התנהגות זו.

שעון זמן אמת

שעון זמן אמת ב-PC

שעון זמן אמת הוא שעון המכצע פסיקה כל פרק זמן קבוע.

ב-PC, השעון מבצע פסיקות בתדירות של 18.2 פעמים בשניה (2^{16} פעמים בשעה).

פסיקת שעון זמן אמת נדרשת לפעול מהר ולסיים את פעולתה לפני פסיקת השעון הבאה, שכן אחרת יאבדו פסיקות שעון בקביעות.

שימושים ב-XINU

- יישום מדיניות החלפת התהליכים.
- שרותי תזמון (sleep) - מאפשר לתהליך שהורדם להתעורר לאחר הזמן הנדרש.
- שעון תאריך (זמן שעבר מהפעלת מערכת ההפעלה).

Delta List

הבעיה: אנו רוצים לבנות מנגנון התומך ב-sleep. תהליכים ישנים ייחכו בתור המתנה מיוחד, וכל פסיקת שעון אנו נעיר את התהליכים שהגיע זמנם להתעורר. כאשר ישנם תהליכים ישנים רבים, יכולה להתעורר בעיה אם נשתמש בתור רגיל, מכיוון שכל פעם נצטרך לסרוק את כל התור לחפש תהליכים שצריכים להתעורר - פעולה הלוקחת זמן.

הפתרון: ניצור רשימת המתנה, בה התהליכים יהיו ממוינים לפי הזמן שבו הם צריכים להתעורר. המפתח של כל איבר ברשימה יכיל את מספר מחזורי השעון שהתהליך רוצה לישון מעבר למספר המחזורים של התהליך הנמצא לפניו ברשימה.

ראש הרשימה יוחזר במשתנה בשם **clockq**. הפונקציה **insertd(pid, head, key)** מכניסה את התהליך pid לרשימה clockq, עם הערך key. הערך key המועבר אל הפונקציה הוא זמן יחסי לזמן הנוכחי.

הפונקציה insertd מקיימת את השמורה הבאה: בכל רגע במהלך החיפוש, גם key וגם q[next].qkey מגדירים השהיה יחסית לזמן בו האב הקדמון יתעורר.

הכנסת תהליך למצב sleep

- תוכניות המשתמש אינן ניגשות אל ה-delta list באופן ישיר, אלא בעזרת קריאות מערכת.
- קריאת המערכת sleept(n) משהה את התהליך הקורא לה ל-n פסיקות שעון. היא מבצעת זאת על ידי הכנסת התהליך ל-delta list. כמו כן, היא קובעת את מצב התהליך ל-PRSLEEP.
- לשימוש ב-sleept יש מגבלה, והיא שמספר מחזורי השעון שאנו יכולים לחכות מוגבל על ידי הגודל של int, ולכן אנו מוגבלים ל-$\left(2^{15}-1\right)$ מחזורי שעון (כ-1800 שניות - חצי שעה).
- על מנת לעקוף את המגבלה, קיימת פונקציה נוספת, בשם sleep. sleep מקבלת זמן בשניות ולא במחזורי שעון, וישנה כמות שניות כמבוקש. אנו עוקפים את מגבלת מחזורי השעון בכך שאם אנו מבקשים לחכות יותר מכמות מסויימת של שניות (כמות המוגדרת על ידי הקבוע **TICSN**), אז הפונקציה sleep מחלקת את ההשהיה למספר קריאות ברצף לפונקציה sleept, שבכל אחת מהן היא תחכה **TICSD** מחזורי שעון.
- אם קוראים לפונקציה sleep עם פרמטר 0, מתבצע resched(). מכיוון שאסור לקרוא ל-resched כאשר פסיקות מופעלות, זוהי שיטה לגרום לתהליך לבצע לעצמו resched().

בעייה במימוש של sleep

במידה והזמן אותו יש לחכות קטן מ-TICSN, תיקרא sleept פעם אחת בלבד. במידה והזמן ארוך יותר, יש לחלק אותו למספר חלקים. הפונקציה sleept תקרא מספר פעמים, זו אחר זו, כך שביחד נחכה את הזמן הרצוי.

כעקרון, במידה שבזמן t נקראת sleep לחכות זמן Δ אזי התהליך צריך להתחדש בזמן $t+\Delta$.

יתכן שיהיו תהליכים בעלי עדיפות גבוהה יותר, ולכן נצפה שתהליך יתחדש לא לפני זמן $t+\Delta$.

במידה ובזמן $t+\Delta$ אין תהליכים בעלי עדיפות גבוהה יותר או זהה לתהליך שחיכה, נרצה שהוא יחזור להתבצע. יותר מכך, אם בזמן זה ישנם תהליכים בעדיפות גבוהה יותר, נרצה שהוא יחזור להתבצע מיד כאשר אין תהליכים בעדיפות גבוהה יותר ממנו.

אולם, נביט במקרה הבא: תהליך א' מבקש לחכות 10000 שניות. 500 שניות אחר כך נוצר תהליך ב', בעל עדיפות גבוהה הפעולה 9000 שניות. נרצה לראות מתי תהליך א' יגמור לחכות.

התשובה: לאחר 18420 שניות!

הסבר: ב-sleep ישנה בעיה הנובעת מכך שהלולאה מתבצעת בעדיפות של התהליך הקורא.

ייתכן כי לאחר תחילת השינה ייכנס לריצה תהליך בעל עדיפות גבוהה יותר. במקרה זה, לא תתבצע הלולאה כל עוד התהליך עם העדיפות הגבוהה יותר רץ.

בדוגמא המוצגת, כאשר התהליך בעל העדיפות הגבוהה נכנס לפעולה הוא מעקב את ביצוע ה-sleept, ולכן בסיום פעולת תהליך ב', ישארו עדיין 8 איטרציות של הלולאה + 280 מחזורי שעון בהם יישן תהליך א'.

טיפול בפסיקת השעון - clkint

תיאור:

פונקצית השירות intcom קוראת לפונקציה **clkint** כאשר מתרחשת פסיקת שעון.

מכיוון שהפונקציה נקרא דרך משגר הפסיקות, clkint מניחה שבכניסה אליה הפסיקות כבר כבויות.

מילת המפתח INTPROC המשמשת כטיפוס המוחזר של הפונקציה, באה להבהיר לקורא את הנחה זו.

הפרמטר המועבר לפונקציה הוא פרמטר דמה שמועבר אוטומטית על ידי intcom.

משתנים גלובלים הקשורים לפונקציה:

- **tod** – time of day - מספרי מחזורי השעון מרגע הפעלת המערכת.
- **defclk** – שעון דחוי - דגל האומר האם אנו במצב שעון דחוי או לא. (יוסבר בהמשך)
- **clkdiff** – מונה מחזורים דחויים.
- **slnempty, sltop** – דגלי עזר המיועדים למנוע חישובים מיותרים בזמן הפסיקה. בעזרת slnempty ניתן לדעת האם התור לא ריק. אם יש תהליכים בתור, sltop מכיל את כתובת המפתח של הראשון מביניהם.
- **preempt** – מונה הזמן עד לקריאה הבאה ל-()resched.

פעולת הפונקציה:

- הגדלת שעון המערכת.
- טיפול בתהליכים ישנים.
- טיפול בהחלפת תהליכים אם יש בכך צורך.

הערת תהליכים ישנים - wakeup

הפונקציה wakeup מעבירה לתור ה-ready את כל התהליכים שזמן ההשהיה שלהם עבר.

כמו כן היא קובעת את המשתנים הגלובליים slnempty, sltop לפני שהיא קוראת ל-resched (כי resched יכולה להעביר לתהליך אחר, בו פסיקות השעון יהיו פעילות).

בעיה בפונקציה kill

כאשר מוציאים תהליך ישן, לא מעדכנים את clockq, ולכן יתכן שתהליכים יתעוררו טרם זמנם.

שעון דחוי

מנגנון הטיפול בשעון כולל אפשרות בשם "טיפול בשעון דחוי".

כאשר משתמשים באפשרות זו, המערכת סופרת את פסיקות השעון, אך אינה גורמת להתרחשות אירועים כלשהם.

בדרך זו נוכל לדעת על אירועים שהיו צריכים לקרוא.

הסיבה למנגנון כזה: המערכת מבטלת פסיקות כאשר מתרחשת החלפת הקשר. כאשר מידע מגיע מהמקלדת זו אינה בעיה, אולם כאשר מידע מגיע מהתקן אחר מהיר יותר זו בעיה, מכיוון שאם תתרחש החלפת תהליכים עם פסיקות חסומות, אנו עלולים לאבד מידע.

לפיכך - מנגנון ה-I/O צריך לאסור החלפת הקשר לתקופות קצרות, בזמן שהפסיקות צריכות להשאר פעילות.

מנגנון השעון הדחוי גורם לדחיית, אך לא ביטול, של הפעולות שהיו צריכות להתרחש, וזאת אם חלף זמן קצר מרגע שהייתה צריכה להתבצע פעולה. במידה ואפשרי, הפעולה תתבצע כשהמערכת תחזור למצב נורמלי.

פונקציות הקשורות לשעון דחוי:

- **stopclk** שמה את השעון במצב דחוי.
- **strtclk** מוציאה את השעון ממצב דחוי.

מספר תהליכים כלשהו יכול לבקש מהשעון להיות דחוי, ולכן אנו מגדילים את defclk ב-1 לכל בקשה. בקשת שחרור השעון ממצב דחוי נעשית על ידי הקטנת defclk.

כל עוד defclk חיובי, שגרת פסיקת השעון מונה את הפסיקות ב-clkdiff ולא מנתחת אותן.

כאשר defclk מגיע ל-0, strtclk מנסה לתפוס את האירועים שהיו אמורים לקרוא, וזאת על ידי הקטנת מספר מחזורי השעון שהצטברו מההשהיה של תהליכים ישנים, והערת תהליכים שזמנם להתעורר הגיע.

ניהול קלט/פלט

קלט/פלט – רמה נמוכה

הפונקציה **kbdgetc()** קוראת ומחזירה את התו הבא מה-BIOS Keyboard buffer. אם אין תו זמין, היא מחזירה את הערך NOCH.

מימוש הפונקציה: הפונקציה בודקת אם יש תו בחוצץ בעזרת קריאת BIOS בשם KBDPEND. אם לא קיים תו בחוצץ, kbdgetc() מחזירה NOCH.

אם קיים תו בחוצץ, הפונקציה קוראת לשגרת BIOS נוספת על מנת לקבל את התו.

בעיה: אם תהליך שני יתחיל לפעול בין הקריאה הראשונה לקריאה השנייה, הוא עלול לקבל את הערך מהמקלדת לפני שלתהליך הראשון תהיה הזדמנות לעשות זאת. במקרה זה, כאשר התהליך הראשון יחזור לפעול הוא יגלה חוצץ ריק, והוא יושהה עקב הקריאה השנייה לפונקצית ה-BIOS.

כדי למנוע ממצב זה לקרות, הפונקציה מנקה את **pcxflag** לפני הקריאות לשגרות BIOS, ומונעת בדרך זו מתהליכים אחרים לרוץ באותו זמן. ערך pcxflag משוחזר לפני החזרה מהפונקציה.

הפונקציה **kgetc()** מחזירה את התו הבא מהמקלדת. אם אין כזה, היא מחכה עד שיגיע תו.

נביט במימוש של הפונקציה:

```
int kgetc(d)
int d;                              /* dummy parameter          */
{
       int    ch;
       while ( (ch=kbdgetc()) == NOCH );
       return ( (ch==RETURN) ? NEWLINE : ch );
}
```

נשים לב שבפונקציה זו ממומש busy-wait.

הסיבה שזו איננה בעיה: פונקציה זו משמשת בעיקר ל-debug ולמקרים בהם הפסיקות מבוטלות, ולכן זו הדרך היחידה לקבל את לחיצות המקלדת.

קלט/פלט שאינו תלוי בהתקן

- מאפשר לתוכנית המשתמש לרוץ על סוגי ציוד שונים ללא שינוי בקוד.
- תוכנית אינה צריכה להכתב מראש לסוג ציוד מסוים. ניתן להחליט על הציוד המסוים בזמן ריצה מתוך מבחר הציוד המקושר למחשב.
- התוכנה המטפלת בציוד פיסי מסוים נקראית device driver. תוכנה זו שקופה למשתמש, הניגש אליה רק באצמעות שירותי מערכת ההפעלה.
- פונקציות קלט/פלט הן כלליות לכל סוגי הציוד ומהוות "אופרטורים מופשטים" הפועלים על כל סוגי הציוד.

אופרטורים מופשטים ב-XINU

אופרטור	פעולה	ערך חזרה
getc(fd)	קורא תו קלט אחד	התו עצמו או EOF
putc(fd, ch)	כותב תו פלט אחד	סטטוס
read(fd, buf, n)	קורא n תווים לתוך החוצץ buf	מספר התאים שנקראו.
write(fd, buf, n)	כותב n תווים מתוך החוצץ buf	מספר התאים שנכתבו.
open(fd, arg1, arg2)	מאפשר פתיחת קבצים על ההתקן fd. fd מציין את ההתקן עליו יפתח הקובץ open מחזיר מציין חדש לקובץ עצמו.	המציין החדש
close(fd)	סוגר את fd. הערה: ישנם התקנים שאסור לסגור כגון CONSOLE.	סטטוס
seek(fd, pos)	מאפשר קביעת מקום הקריאה/הכתיבה הבא בקובץ. לא רלוונטי לצג.	סטטוס
control(fd, pos)	מאפשר פונקציות נוספות להתקנים. למשל עבור צג מגדיר אפשרות הדהוד וגם מצבי עבודה שונים.	סטטוס
init(fd)	פונקצית אתחול להתקן בזמן טעינת המערכת.	סטטוס

פעולות סינכרוניות ואסינכרוניות

- קריאה סינכרונית לשגרת קלט/פלט הינה קריאה הממתינה לסיום הפעולה
- קריאה אסינכרונית הינה קריאה הממשיכה בביצוע הקוד בלי להמתין לסיום הפעולה
- בד"כ יותר נוח להשתמש בקריאות סינכרוניות. כך למשל פעולת getc מהתקן תחזור עם התו שקראה.
- מערכת ההפעלה מאפשרת קריאות סינכרוניות שאינן תופסות את המעבד בזמן ההמתנה.

- האופרטורים המופשטים המוגדרים ב- XINU הינם סינכרוניים.

קישור אופרטורים

- האופרטורים המופשטים נועדו להסתיר את פרטי החמרה מתכניות המשתמש ולהפוך אותן לבלתי תלויות בחמרה הספציפית.
- המערכת דואגת למפות שמות מופשטים (כמו CONSOLE) ואופרטורים מופשטים (כמו READ) להתקנים פיזיים ו- device drivers.
- שאלה מרכזית בתכנון מערכת הפעלה הינה: מתי יתבצע הקישור?
- קישור מאוחר (למשל בזמן ריצה) מאפשר לתכניות משתמש גמישות רבה, אולם נושא תקורה גדולה.
- קישור מוקדם (למשל בזמן הידור) מאפשר יעילות רבה אולם מחייב הידור מחדש עם כל שינוי בקונפיגורציה.

- ב- XINU (כמו מערכות רבות אחרות) מתבצע קישור בין האופרטורים המופשטים בתכניות המשתמש לבין הפונקציות הספציפיות המממשות אותם בזמן ריצה.
- לדוגמא, getc(fd) יקושר לפרוצדורת קריאה מלוח המקשים אם ערכו של fd הוא CONSOLE, ויקושר לפרוצדורת קריאה מדיסק אם ערכו של fd הינו מזהה של קובץ.
- הגדרות ההתקנים, הקישור בין ההתקנים המופשטים לבין ה- device drivers נעשה בזמן בנית המערכת.

מימוש ב-XINU

מוגדרת טבלת התקנים. לכל התקן יש מציין שהוא האינדקס שלו בטבלת ההתקנים. בכל כניסה בטבלה ש מצביעים לאופרטורים הספציפיים לאותו התקן וכן נתונים נוספים הדרושים להתקן. לכל אופרטור מופשט מועבר מציין ההתקן כפרמטר. האופרטור המופשט משתמש בפרמטר זה על מנת לגשת אל הכניסה המתאימה בטבלת ההתקנים.

כניסה בטבלת ההתקנים מוגדרת כך:

```
/*-----------------------------------------------------------------------
 * Format of each entry is:
 *
 * device number, device name,
 * init, open, close,
 * read, write, seek,
 * getc, putc, cntl,
 * port addr, device input vector, device output vector,
 * input interrupt routine, output interrupt routine,
 * device i/o block, minor device number
 *-----------------------------------------------------------------------
 */

/* Device table declarations */
struct devsw  {                   /* device table entry            */
       int    dvnum;
       char   dvnam[8];
       int    (*dvinit)();
       int    (*dvopen)();
       int    (*dvclose)();
       int    (*dvread)();
       int    (*dvwrite)();
       int    (*dvseek)();
       int    (*dvgetc)();
       int    (*dvputc)();
       int    (*dvcntl)();
       int    dvport;
       int    dvivec;
       int    dvovec;
       int    (*dviint)();
       int    (*dvoint)();
       char   *dvioblk;
       int    dvminor;
       };
```

דוגמא - הפונקציה read:

```
read(descrp, buff, count)
int descrp, count;
char *buff;
{
       struct devsw  *devptr;

       if (isbaddev(descrp) )
              return(SYSERR);
       devptr = &devtab[descrp];
       return(       (*devptr->dvread)(devptr,buff,count) );
}
```

ionull ,ioerr

קיימים אופרטורים שאינם לוונטים להתקנים מסויימים. בטבלת ההתקנים חייבת להיות פונקצית טיפול גם באופרטורים אלו, וזאת מכיוון שהאופרטורים המופשטים נגישים לטבלת ההתקנים בלי בדיקות האם הפעולה המבוקשת חוקית עבור אותו התקן. לשם כך מוגדרות שתי פונקציות טיפול סטנדרטיות:

ionull - תמיד מחזירה OK. משמשת עבור אופרטורים שאינם רלוונטים אך לא נורא אם יתבצעו.

ioerr - תמיד מחזירה SYSERR. משמשת עבור אופרטורים שמוסכם שאין לקרוא להם עבור ההתקן (פעולות לא הגיוניות).

שינוי מודי פעולת ה-driver

בקשת שינוי מצב נעשת על ידי בקשת control. כמו כן, control משמשת לבקשות נוספות שאינן נעשות על ידי שאר האופרטורים. הבקשות המיושמות ב-driver של ה-console:

- ch = control(CONSOLE, TCNEXTC) מחזירה את התו הבא בקלט (בלי לקרוא אותו)
- control(CONSOLE, TCMODER) שינו מוד קלט ל-RAW
- control(CONSOLE, TCMODEC) שינוי מוד קלט ל-Cooked
- control(CONSOLE, TCMODEK) שינוי מוד קלט ל-Cbreak
- control(CONSOLE, TCECHO) אפשור הדהוד
- control(CONSOLE, TCNOECHO) ביטול הדהוד
- n=control(CONSOLE, TCICHARS) מחזיר את מספר התווים בחוצץ הקלט

שם ההתקן

לכל התקן מתאים שם, מחרוזת.

כדי לקבל את מציין ההתקן מתוך שמו נשתמש בפונקציה getdev().

הפונקציה עובדה בסיבוכיות O(n), אולם עובדה זו לא מפריעה לנו, כי לרוב הפונקציה נקראת רק פעם אחת עבור התקן, ומאותו רגע משתמשים במזהה ההתקן שלו.

Terminal Driver

- Terminal Driver הוא Device Driver עבור מסוף (מקלדת ומסך).
  דרייבר - מהווה ממשק בין האפליקציה ובין החומרה.
- הדרייבר מופעל מהאפליקציה על ידי קריאות מערכת ההפעלה.
- מצד האפליקציה: האפליקציה יוזמת קריאות מערכת הפעלה לשם קריאה וכתיבה.
- מצד החומרה: החומרה מבצעת פסיקות חומרה כאשר מגיע תו מהמקלדת. בתואמי PC הצג ממופה לזכרון ולכן אין עבורו פסיקות חומרה. במחשבים אחרים מקבלים פסיקות חומרה לשם משלוח התוים לפלט.
- ה-driver נחלק לשני חלקים נפרדים - חלק עליון המטפל בבקשות האפליקציה, וחלק תחתון המטפל בחומרה. שני חלקי ה-driver מדברים ביניהם דרך חוצץ, המאכסן בתוכו את התוים או הבקשות שנשלחים מהאפליקציה לחומרה או מהחומרה אל האפליקציה, וזאת עד שידרשו על ידי הצד המקבל אותם.

חלוקה לחלק עליון ותחתון

החלק העליון מתקשר עם תהליכי המשתמש. החלק העליון לא מתקשר עם החומרה באופן ישיר, ובמקום זאת הוא שולח בקשות אל החלק התחתון, שמבצע אותן כאשר הוא מתפנה מהבקשות הקודמות.

חלוקה זו חשובה ביותר כשמתכננים מתאמי התקנים, וזאת מכיוון שהיא מפרידה בין ניתוח הבקשות הרגיל לבין ניתוח הנדרש על ידי החומרה הספציפית.

שימוש בחוצצים

מתאם ההתקן משתמש בחוצצים כדי לשמור מידע העובר בין תהליכי המשתמש אל ההתקנים. כמו כן החוצצים יכולים לשמש להעברת מידע מההתקנים אל תהליכים המשתמש.

- **קלט** - כאשר מגיע תו מהמקלדת מופעלת פסיקה, האחראית להעברת התו לחוצץ הקלט (חלק תחתון). getc מוציאה תו מחוצץ הקלט ומעבירה אותו לאפליקציה. אם אין תווים בחוצץ, getc גורמת לתהליך לחכות עד שיגיע תו (חלק עליון). החלק התחתון גם אחראי לשחרור תהליך הממתין לקלט כשהקלט הגיע.
- **פלט** - החלק העליון (putc) נקרא מהאפליקציה ומעביר תווים לחוצץ הפלט. כאשר מתמלא החוצץ, התהליך מתבקש לחכות עד לפינוי מקום. החלק התחתון נקרא ע"י פסיקות (לא ב- PC) ושולח תו פלט במידה שממתין תו בחוצץ.

החוצצים מאפשרים להפריד בין שני חלקי ה- driver.

- מאפשרים לקלט להגיע לפני שהאפליקציה מבקשת אותו. המתאם יכול לקבל נתונים, ולשים אותם בחוצצים, בהמתנה עד שתהליכי המשתמש ידרשו אותם.
- מאפשרים לאפליקציה לבצע פלט לפני שהחומרה מוכנה לכך ובלי לחכות לסיום הפלט ממש - פעולות ה-I/O יכולות להתבצע במקביל לתהליכי המשתמש, ולא לעכב אותם.
- מאפשרים שבירת בלוקים לתווים בודדים: למשל בדיסקים: הקריאה היא בבלוקים של 512 בתים, אך האפליקציה מעוניינת בתווים בודדים בכל קריאה. כך נשמר שאר הבלוק לצורך קריאת התווים הבאים.

תאום בין החלק העליון לתחתון

ניתן לתאר את היחס בין החלק העליון לתחתון כיחס יצרן/צרכן, המיושם באופן אידיאלי על ידי סמפורים.

בקלט - החלק התחתון מייצר תווים והחלק העליון צורך אותם.

בפלט - החלק העליון מייצר תווים והחלק התחתון צורך אותם.

בעיה: התקן הפלט איטי מאוד ביחס למעבד, ולכן חוצץ הפלט יתמלא במהירות. בנוסף, במערכות בהן משתמש התקן הפלט בפסיקות לא יכול החלק התחתון של הפלט להמתין לתווים.

פתרון: ב-XINU הפכו את כוון ההסתכלות בפעולות הפלט: החלק התחתון מייצר מקומות פנויים, ואילו החלק העליון צורך אותם.

בקלט - החלק העליון עוצר את התהליך כאשר לא הגיע קלט, עד שהחלק התחתון יעיר אותו.

בפלט - החלק העליון ממתין להתפנות מקומות בחוצץ, במידה שהוא התמלא. כמו כן החלק העליון צריך להעיר את התחתון במידה שהחלק התחתון הפסיק את פעילותו מפאת התרוקנות החוצץ.

יחסי יצרן - צרכן

בקלט: החומרה מייצרת תוים ומכניסה אותם לחוצץ. האפליקציה צורכת את התוים, וממתין במידה שהחוצץ ריק.

בפלט: טבעי להסתכל על האפליקציה כיצרן, ועל החומרה כצרכן. בחלוקה כזו החומרה צריכה לחכות לאפליקציה. למעשה, כשהחוצץ מתמלא, האפליקציה ממתינה להתפנות מקום.

נוח יותר להסתכל על החומרה כיצרן מקומות פנויים בחוצץ, ועל האפליקציה כצרכן המקומות הפנויים. כאשר חלקו התחתון של ה- driver מבצע פלט לתו, הוא מפנה את המקום של התו בחוצץ. כאשר החלק העליון מבקש לכתוב תו, הוא דורש מקום פנוי בחוצץ, ומחכה למקום פנוי במידה שאין כזה.

בשני המקרים, החלק העליון מעכב את התהליך, והחלק התחתון מחדש את פעולת התהליך שעוכב.

בפועל נשתמש לשם כך בסמפורים:

החלק התחתון מייצר ומסמן ב- signal. החלק העליון צורך ומסמן ב- wait.

דגש

signal מותרת מתוך פסיקה. wait לא! (כי יתכן אז מצב שנוציא את NULLPROC מתור ה-ready).

תור בקשות

ב- driver-ים צריך ליישם לעתים תורי בקשות.

תור הבקשות הוא המבנה העיקרי המקשר בין החלק העליון של מתאם ההתקן לבין החלק התחתון. לכל מתאם התקן יש תור בקשות משלו, ואופן הבקשות משתנה בהתאם להתקן.

במקרה של ה- tty driver הסמפורים מטפלים לנו בתור של המחכים, בעוד הבקשות הן פשוטות: כתוב תו מסוים או קרא תו. ב- driver של דיסק הבקשות מורכבות יותר: קריאה ממקום מסוים או כתיבה למקום מסוים. אנו צריכים לשמור את הבקשה עצמה עד שנוכל לטפל בה. אין אנו מעוניינים שהסמפורים יטפלו לנו בתור הבקשות, שכן אנו רוצים לבצע אופטימיזציה על הסדר של הבקשות: היות שלוקח זמן ארוך יחסית להעברת הראש ממקום למקום, נרצה שהראש יזוז קודם לבקשה קרובה ואח"כ לרחוקה, כדי לחסוך במרחק שעליו לעבור.

ישום ב- XINU

ב- PC הצג ממופה זיכרון, ובפלט לצג אין צורך לחלק את ה- driver לחלקים, אלא ניתן לכתוב עליו ישירות. עם זאת, מהסיבות שהוזכרו לעיל, קיימת החלוקה לחלק עליון ולחלק תחתון, וכמו כן, החלק התחתון מיושם על ידי תהליכים (במקום בפסיקות).

בלוק בקרה

לכל התקן tty מוגדר בלוק בקרה הכולל את כל הנתונים של ההתקן: (בקובץ tty.h)

- החוצצים
- מצייני הסמפורים
- מודי הפעולה:
  - טיפול בסוף שורה: icrlf, ocrlf
  - הדהוד: iecho
  - אופני הקלט: imode
  - עצירת גלגול
  - ועוד... (הרבה)

החוצצים

שלושה חוצצים מעגליים (ציקליים):

- חוצץ הקלט ibuff
- חוצץ הפלט obuff
- חוצץ ההדהוד ebuff

חוצץ ההדהוד מאפשר לבצע הדהוד לתווי הקלט שהגיעו. כל החוצצים מעגליים ולכל אחד מהם מצביעי ראש וזנב. בהכנסה: מוסיפים לראש. בהוצאה: מוציאים מהזנב.

כמו כן יש מונה הסופר את מספר התווים הנמצאים בחוצץ.

מעקב אחרי קריאת קלט - חלק עליון

- תוכנית משתמש מבצעת getc(CONSOLE).
- הפונקציה הכללית getc() משתמשת במציין CONSOLE וקוראת לפונקצית הקלט המתאימה הנמצאת באינדקס זה בטבלה devtab[] - ttygetc().
- הפונקציה ttygetc() מקבלת כפרמטר מצביע לכניסה בטבלת ה-devtab[] ומוצאת על פי ה-dvminor את הכניסה המתאימה בטבלה ה-tty[]. בכניסה זו נמצאים כל הנתונים הקשורים לקלט/פלט של התווים.
- התוכנית מחכה (על ידי wait) לתו בחוצץ הקלט. (ה-signal יגיע מתהליך הקלט - ttyiproc).
- התוכנית מוציאה תו מחוצץ הקלט ומחזירה אותו למשתמש.

```
ttygetc(devptr)
struct devsw  *devptr;
{
       int    ps;
       char   ch;
       struct tty   *iptr;

       disable(ps);
       iptr = &tty[devptr->dvminor];
       wait(iptr->isem);          /* wait for a character in buff   */
       ch = iptr->ibuff[iptr->itail++];
       --iptr->icnt;
       if (iptr->itail     >= IBUFLEN)
              iptr->itail = 0;
       restore(ps);
       return(ch);
}
```

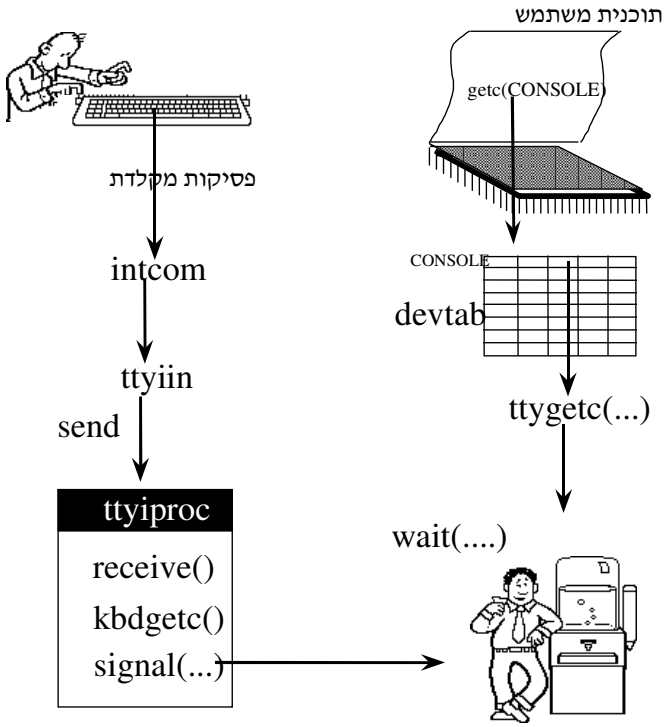
קלט - חלק תחתון

תהליך הקלט - ttyiproc רץ בלולאה אינסופית ומבצע:

1. מחכה להודעה שהגיע תו. עם הקשת תו נשלחת הודעה להתליך הקלט על ידי השגרה לטיפול בפסיקת מקלדת (ttyiin).
2. תהליך הקלט קורא את התו על ידי פונקצית BIOS, ופועל לפי אופן הטיפול בקלט (mode).

getc

- כאשר המשתמש קורא ל- getc(CONSOLE) ממופה הקריאה דרך טבלת ההתקנים לפונקציה ttygetc()
- בפונקציה ttygetc() נבדק החוצץ, ואם אין תו הפונקציה ממתינה על סמפור (ולכן אינה צורכת זמן מעבד)
- כאשר המשתמש מקיש על מקש, נוצרת פסיקה.
- פונקצית השיגור intcom קוראת לפונקצית הטיפול ttyiin.
- בזמן אתחול המערכת יוצרת XINU תהליך בשם ttyiproc
- תהליך זה מטפל בתווים שהגיעו מהמקלדת. פרוצדורת הפסיקה מסמנת לו שהגיע תו חדש ע"י שליחת הודעה.
- התהליך מטפל בתו ומעיר את ttygetc ע"י signal.



putc

- כאשר המשתמש קורא ל- putc(ch,CONSOLE) ממופה הקריאה דרך טבלת ההתקנים לפונקציה ttyputc()
- בפונקציה ttyputc() נבדק החוצץ, ואם אין מקום ריק (כלומר החוצץ מלא) הפונקציה ממתינה על סמפור
- כאשר מתפנה מקום בחוצץ, כותבת ttyputc את התו בחוצץ ושולחת הודעה לתהליך הפלט ttyoproc.
- התהליך ttyoproc נמצא בלולאה אינסופית וכותב תווים מהחוצץ באמצעות wtty.
- כאשר נכתבים תווים (ולכן נוצרים מקומות פנויים) ttyoproc מבצע signal על הסמפור
- כאשר אין תווים בחוצץ, מחכה התהליך להודעה מ- ttyputc על הגעת תו חדש
- כאשר קבל הודעה על תו חדש שהגיע לחוצץ מתעורר לחיים וממשיך בלולאת כתיבת התווים.

סימני מים

הבעיה:

כאשר החוצץ מלא, ותהליך מבקש לכתוב מספר תוים נגיע למצב הבא:

1. התפנה מקום אחד בחוצץ.
2. המקום נצרך ואין מקום פנוי נוסף ⇐ resched.
3. מתפנה מקום נוסף.
4. חזור ל- 1.

במצב זה, תהליך המבקש לכתוב מספר תוים יבצע resched על כל תו. פעולה זו הינה יקרה, ובעיקר מיותרת.

למשל, תהליך המבקש לכתוב 100 תוים כאשר יש רק 30 מקומות פנויים בחוצץ, ע"י (CONSOLE, write buf, 100), יכתוב 30 תוים ואח"כ על כל תו נוסף תתבצע החלפת תהליכים.

סימני מים - הפתרון:

נדאג לכך שהחלק התחתון ישחרר מקומות פנויים בקבוצות, ולא בבודדים. הקבוע OBMINSP קובע את גודל הקבוצה (20). כאשר מספר המקומות הפנויים קטן מ- OBMINSP נפסיק לסמן לסמפור על התפנות המקומות למשך OBMINSP מקומות נוספים, נסמן את כולם לסמפור בבת אחת.

בכך אנו חוסכים מספר רב של החלפות תהליכים.

הערות ודגשים

- אם נרצה, למשל, לגרום ל-XINU לנתח את הקלט של המשתמש ולהגיב בהתאם - למשל - אם נרצה שבמקרה שיוקש רצף מסוים של תווים XINU תבצע פעולה כלשהי, נוסיף את הקוד המתאים ב-()ttyiproc.
- המיקום בו נוצרים התהליכים ttyiproc, ttyoproc. באיתחול המערכת main קוראים ל-sysinit. הפונקציה קוראת לאופרטור המופשט (0)init, כאשר 0 מציין את מזהה ההתקן (tty). init קוראת ל-ttyinit, שיוצרים את התהליכים של tty.
- עדיפות תהליך הפלט: TTYOPRIO. עדיפות תהליך הקלט: TTYIPRIO.

נספח א' – דוגמאות

תורים

דוגמא 1

שגרת המערכת הבאה עוברת על תור ומדפיסה את כל האיברים בו.

השגרה מקבלת את ראש התור.

```
void prnList(int list)
{
	int iIndex = firstid(list);
	int ps;
	disable(ps);
	printf("%10s%10s%10s\n", "PID", "ProcName","Prio");
	while (iIndex < NPROC)
	{
		printf("%10d%10s%10d\n", iIndex, proctab[iIndex].pname,
			proctab[iIndex].pprio);
		iIndex = q[iIndex].qnext;
	}
	restore(ps);
}
```

דוגמא 2

הפונקציה הבאה מציגה פונקציונאליות דומה לזו של enqueue, אולם היא מוסיפה איבר בתחילת הרשימה ולא בסופה. נשים לב, שבדומה לפונקציות האחרות הפועלת על תורים, היא איננה עטופה ב-disable/restore.

```
int putfirst(int item, int list)
{
	struct qent	*hptr;		/* points to head entry	*/
	struct qent	*mptr;		/* points to item entry	*/
	hptr = &q[list];
	mptr = &q[item];
	mptr->qprev = list;
	mptr->qnext = hptr->qnext;
	q[hptr->qnext].qprev = item;
	hptr->qnext = item;
	return(item);
}
```

נספח ב' - כתיבת שגרות עבור XINU

שגרות מערכת

- כל שגרת מערכת עטופה ב-disable/restore.
- שגרת מערכת חייבת לבדוק את הפרמטרים שהיא ניגשת אליהם. במידה והיא משנה או ניגשת אל מבנים פנימיים, היא צריכה לבדוק גם שהמבנים הפנימיים נמצאים במצב תקין.
- בסוף כל מסלול בשגרה (המסתיים ב-return) צריך לקרוא ל-restore.
- במידה ורוצים ששגרת המערכת תחזיר מידע מתוך משתני מערכת ההפעלה, יש לשמור את המידע במשתנה מקומי לפני שמבצעים restore, ורק אז להחזיר את המידע.

שגרות משתמש

- קטעים קריטיים של תוכניות משתמש צריכים להיות עטופים ב-wait/signal, אפילו אם הם בני שורה אחת.