

תרגול 1 – מבוא

חזרה על נאנד – החומרה

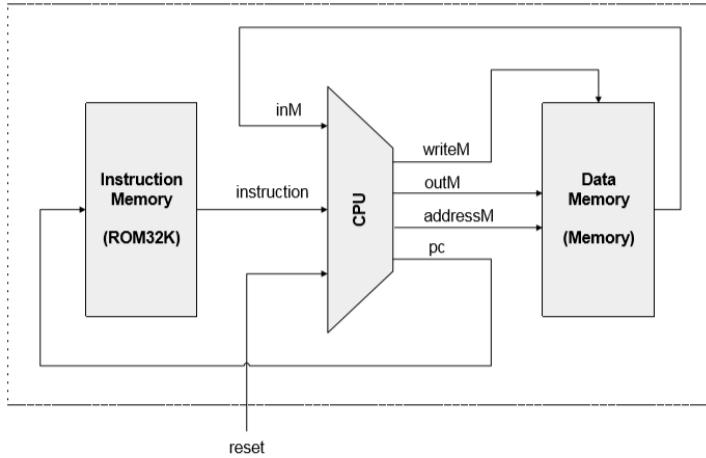


DIAGRAM 5.10: Proposed implementation of the top-most Computer chip.

המעבד: CPU

מכיל ורגיסטרים, העיקריים:

- Instructions register – שומר את הפקודה שנאנכו עומדים לבצע.
- Program counter – איפה הפקודה נמצא בזיכרון.
- Stack pointer – שומר את המיקום בו ה-stack נמצא.

מבצע פקודות:

1. טיפול בננתונים, במידע (קריאה וכתיבה ל-RAM)
2. פעולות אריתמטיות שמעורבות את ה-ALU
3. control flow – קפיצה לפונקציה, לולאות for, תנאים וכו'.

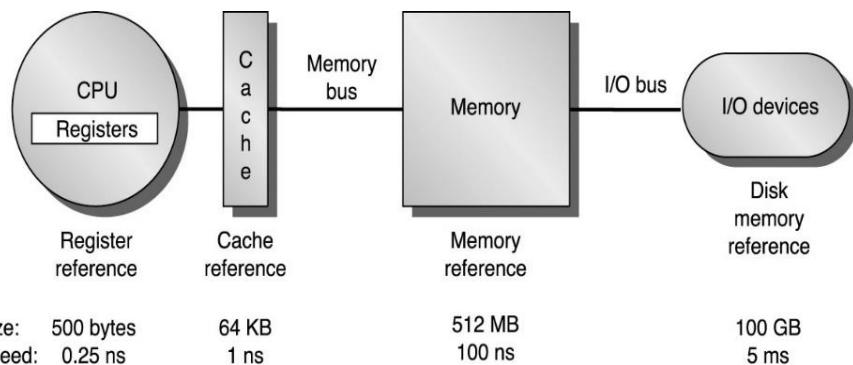
באשר כל פקודה היא רצף של ביטים.

איך ה-CPU מבצע זאת?

1. קורא את הפקודה מהזיכרון.
2. מפענח מה הפקודה.
3. לפי הפקודה:
 - חישוב ב-ALU.
 - כתיבה/קריאה ב-RAM.
 - קריאה/כתיבה ברגיסטרים של ה-CPU.

היררכיית הזיכרון

נרצה זיכרון מהיר, גדול וזול. מוגבלים מבחינת מקום. ב-CPU יש מעט רגיסטרים שהגישה אליהם מאד מהירה. אחרי, ה-Cache שהגישה אליו יותר איטית פי 4 מה-CPU אבל עם יותר מקום. באותו אופן אחרי ה-Cache יש את ה-RAM ואחריו הארד דיסקים, וכל שמתקדמים בהיררכיה הזיכרון גדל אבל הגישה אליו איטית יותר.



Valgrind

כלי שיעזר לdebug. נסתכל על כמה דוגמאות:

דוגמה 1

הבעיה:

הוללה רצה שלוש איטרציות, בסוף האיטרציה השליישית $i=2$, עושים $i++$, لكن $i=3$. נגשים לאיבר הריבועי במערך של שלושה איברים ועושים דרישת זיכרון ל-.bar.

ההרצה של ולגרינד על הקוד (קימפול עם g-, עמ' 18 במצגת) לא תמצא בŁom, כי לא הייתה הקצאה דינמית.

דוגמה 2

הבעיה:

גישה למקום ה-11 במערך עם 10 איברים, דרישת זיכרון.

כאן ולגרינד יזהה גישה למקום שלא הוקצת בזיכרון (עמ' 21). ב

```
#include<stdlib.h>
#include<stdio.h>
typedef struct Foo {
    int arr[3];
    int bar;
} Foo;
int main(int argc, char *argv[]) {
    Foo t;
    int i;
    printf("start\n");
    t.bar = 6;
    printf("t.bar = %d\n", t.bar);
    for (i=0; i<3; i++) {
        t.arr[i] = i+1;
    }
    t.arr[i] += t.arr[0] + t.arr[1];
    printf("t.arr[2] = %d\n", t.arr[2]);
    printf("t.bar = %d\n", t.bar);
    return 0;
}
```

```
#include<stdlib.h>
#include<stdio.h>
void foo(int n) {
    int i;
    int *a = (int*) malloc(n*sizeof(int));
    a[0] = 1;
    printf("a[0] = 1\n");
    a[1] = 1;
    printf("a[1] = 1\n");
    for (i=2; i<=n; i++) {
        a[i] = a[i-1] + a[i-2];
    }
    printf("a[%d] = %d\n", i, a[i]);
}
free(a);
}
int main(int argc, char *argv[]) {
    foo(10);
    return 0;
}
```

```
#include<stdlib.h>
#include<stdio.h>
void foo(int n) {
    int *a = (int*) malloc(n*sizeof(int));
    int i;
    for (i=0; i<n; i++) {
        a[i] = i*i;
        printf("a[%d] = %d\n", i, a[i]);
    }
}
int main(int argc, char *argv[]) {
    foo(10);
    return 0;
}
```

דוגמה 3**הבעיה:**

אין free, דליפת זיכרון. ולגרינט מזהה (עמ' 24)

```
#include<stdlib.h>
#include<stdio.h>
#define DIM 1000000
int main(int argc, char *argv[]) {
    float *a;
    int i;
    a = (float*) malloc(DIM*sizeof(float));
    for (i=0; i<DIM; i++) {
        a[i] = i;
        if (i == DIM-1000) {
            printf("i == %d", (DIM-1000));
            a[DIM+i] = a[i];
            a[DIM-1] = a[i];
        }
    }
    printf("Done");
    free(a);
    return 0;
}
```

דוגמה 4**הבעיה:**

segmentation fault, הגענו לכתובת לא חוקית
מחוץ ל-segment שהוקצה. (עמ' 27)

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[]) {
    int y;
    y += 1;
    printf("Done\n");
    return 0;
}
```

דוגמה 8**הבעיה:**

у לא אוטחל, ניתן להוות בקומpileציה אם
מкомפלים עם -Wall. (עמ' 38)

דוגמה 9**הבעיה:**

(עמ' 40). invalid write ,free (.

```
#include<stdlib.h>
#include<stdio.h>
int main(int argc, char *argv[]) {
    int arrLen = 10;
    int* arr;
    int i;
    arr = (int*) malloc(arrLen*sizeof(int));
    for(i=0; i<arrLen; i++) {
        arr[i] = i;
    }
    free(arr);
    for(i=0; i<arrLen; i++) {
        arr[i] = i*2;
    }
    printf("Done\n");
    return 0;
}
```

דוגמה 10**הבעיה:**

(עמ' 43). invalid free ,free (.

```
#include<stdlib.h>
#include<stdio.h>
int compute(int len, int* arr) {
    int sum = 0, i;
    for(i=0; i<len; i++) {
        sum += arr[i];
    }
    free(arr);
    return sum;
}
int main(int argc, char *argv[]) {
    int arrLen = 10, sum, i;
    int* arr;
    arr = (int*) malloc(arrLen*sizeof(int));
    for(i=0; i<arrLen; i++) {
        arr[i] = i;
    }
    sum = compute(arrLen, arr);
    free(arr);
    printf("sum = %d", sum);
    return 0;
}
```

```
<100|134>orenstal@pond:~/os17/tirgut11% valgrind example
==13036== Memcheck, a memory error detector
==13036== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==13036== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for copyright info
==13036== Command: example
==13036==
==13036== Invalid free() / delete / delete[] / realloc()
==13036==   at 0x4C2CDFB: free (vg_replace_malloc.c:530)
==13036==   by 0x400643: main (ex10.c:28)
==13036==   Address 0x51d9040 is 0 bytes inside a block of size 40 free'd
==13036==   at 0x4C2CDFB: free (vg_replace_malloc.c:530)
==13036==   by 0x4005C5: compute (ex10.c:11)
==13036==   by 0x400634: main (ex10.c:27)
==13036== Block was alloc'd at
==13036==   at 0x4C2BBCF: malloc (vg_replace_malloc.c:299)
==13036==   by 0x4005F1: main (ex10.c:21)
==13036==
sum = 45==13036==
==13036== HEAP SUMMARY:
==13036==     in use at exit: 0 bytes in 0 blocks
==13036==   total heap usage: 2 allocs, 3 frees, 1,064 bytes allocated
==13036==
==13036== All heap blocks were freed -- no leaks are possible
==13036==
==13036== For counts of detected and suppressed errors, rerun with: -v
==13036== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
<101|0>orenstal@pond:~/os17/tirgut11%
```

GDB (GNU debugger)

קימפול עם הדגל -g:

- gcc -o myProg -g test.c
- g++ -o myProg -g test.cpp

ולאחר מכן צריך להוסיף:

`gdb myProg`

פקודות בסיסיות

- **run:** starts executing the program.
- **break:** inserts breakpoint in which the execution will suspend.
- **next:** executes the next line (even if it's function call), unless there is a suspending execution event (later today)
- **step:** steps into the next line.
- **cont:** continues the execution till the next suspending execution event or end of execution.
- **print var:** prints the value of *var*.

Breakpoints:

- break *function* (break *main*)
- break <*line_num*> (break 4)
- break *filename:function* (break *temp.c:main*)
- break *filename: <line_num>* (break *temp.c:2*)
- break **address* (break *0x12345)
- Etc.

נזכיר לדוגמה 1, גישה לאיבר הרבייעי בזיכרון בגודל 3 ודרישה של *bar*, ולגרירד לא עוזר כי אין הקצתה דינמית. נdfsis *main* break ונירץ את התוכנה. התוכנה רצה עד שהיא נתקלה ב-breakpoint, מdfsis את *t.bar*. (בעמ' הבא)

הפעולות האלה לא כל כך עוזרו לנו ולכן יש את פקודת **watch**: נבחר משתנה, והתוכנה תעצור ברגע שהוא משתנה.

- watch *var* (watch *arrSize*)
- watch *condition* (watch *i==4*)
- watch **address* (watch *0x12345)

במקרה של הבעה שלנו, יוכל לדעת איפה *t.bar* משתנה (הרצה בעמ' 7)

דרך נוספת לתפיסת הpag יכולת להיות ע"י הדפסת הכתובת: (Print &*t.bar* (to get the address) או ע"י .Watch *add_of_t.bar פקודת watch על הכתובת:)

```
#include<stdlib.h>
#include<stdio.h>
typedef struct Foo {
    int arr[3];
    int bar;
} Foo;
int main(int argc, char *argv[]) {
    Foo t;
    int i;
    printf("start\n");
    t.bar = 6;
    printf("t.bar = %d\n", t.bar);
    for (i=0; i<3; i++) {
        t.arr[i] = i+1;
    }
    t.arr[i] += t.arr[0] + t.arr[1];
    printf("t.arr[2] = %d\n", t.arr[2]);
    printf("t.bar = %d\n", t.bar);
    return 0;
}
```

```
<104|0>orenstal@pond:~/os17/tirgut11% gdb example
GNU gdb (Debian 7.11.1-2) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from example...done.
(gdb) break main
Breakpoint 1 at 0x400545: file ex1.c, line 13.
(gdb) run
Starting program: /cs/grad/orenstal/os17/tirgut11/example

Breakpoint 1, main (argc=1, argv=0x7fffffff4d8) at ex1.c:13
13          printf("start\n");
(gdb) next
start
14          t.bar = 6;
(gdb) print t.bar
$1 = 0
(gdb) next
15          printf("t.bar = %d\n", t.bar);
(gdb) print t.bar
$2 = 6
(gdb) cont
Continuing.
t.bar = 6
t.arr[2] = 3
t.bar = 9
[Inferior 1 (process 13267) exited normally]
(gdb) 
```

```

Type `apropos word' to search for commands related to 'word'...
Reading symbols from example...done.
(gdb) break main
Breakpoint 1 at 0x400545: file ex1.c, line 13.
(gdb) run
Starting program: /cs/grad/orenstal/os17/tirgut11/example

Breakpoint 1, main (argc=1, argv=0x7fffffff4d8) at ex1.c:13
13          printf("start\n");
(gdb) next
start
14          t.bar = 6;
(gdb) next
15          printf("t.bar = %d\n", t.bar);
(gdb) watch t.bar
Hardware watchpoint 2: t.bar
(gdb) cont
Continuing.
t.bar = 6

Hardware watchpoint 2: t.bar

Old value = 6
New value = 9
main (argc=1, argv=0x7fffffff4d8) at ex1.c:22
22          printf("t.arr[2] = %d\n", t.arr[2]);
(gdb) cont
Continuing.
t.arr[2] = 3
t.bar = 9

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
 libc_start_main (main=0x400536 <main>, argc=1, argv=0x7fffffff4d8, init=<optimized out>, f
  at ../csu/libc-start.c:325
325  .../csu/libc-start.c: No such file or directory.
(gdb) cont
Continuing.
[Inferior 1 (process 13605) exited normally]
(gdb)

```

פקודות שימושיות נוספתות:

- delete <break_num>: remove breakpoint
- disable <watch_num>: remove watch point
- help: help function
- bt: prints the backtrace (in case the program crashed)
- up: going to the calling function context (for example: to print variables values)
- down: going back down the function stack, one function at a time.
- info breakpoints: prints a list of all the defined breakpoints and watches.

– מבוסס בעצם עם GDB, GUI שעושה את כל הדברים האלה. **CLion**

Debugging system call

מוטיבציה לתרגיל 1

- בתרגיל נמדד זמנים של פעולות שונות שבמציע המעבד.
- קרייה לפונקציה ריקה, כדי למדוד את overhead של היכנס ויצאת מהפונקציה. בסדר גודל אחר מפקודה פשוטה.
- קרייה לפונקציה ריקה של מערכת הפעלה, זה יהיה בסדר יותר גדול מפונקציה **System call**.

– דיבוג תוכנה שמבצעת `call system`. התוכנית מדפיסה את כל ה-`system calls` שהתוכנית שאנו חנכו מדברים השתמשה בהם. מראה את כל הארגומנטים, הערך שלהם והפלט שהקירה החזירה.

פתרונות של התוכנה:

- Shows system calls, arguments, and return values
- **-t** to display when each call is executed
- **-T** to display the time spent in the call
- **-e** to limit the types of calls
- **-o** to redirect the output to a file
- **-s** limit the length of print strings.

דוגמה:

פתחת קובץ ע"י הפקודה הבאה:

int open(const char *pathname, int flags);

- Opens a file
- Returns **file descriptor (fd)**, which identifies the file in future operations
- $fd=0 \rightarrow$ standard input, $fd=1 \rightarrow$ standard output, $fd=2 \rightarrow$ standard error

קריאה/כתיבה בקובץ, הפקנציות מקבלות את ה-`fd`, `buffer` ואת הגודל שלו. שתיהן מוחזרות `int` שמתאר כמה בתים כתבנו:

ssize_t read(int fd, void *buf, size_t count);

- Reads from `fd` to `buf` between 1 to `count` bytes
- Returns the number of bytes that were read (zero for `eof`)

ssize_t write(int fd, const void *buf, size_t count);

- Writes from `buf` to `fd` between 1 to `count` bytes
- Returns the number of bytes that were written

```
open("/usr/share/locale/en_GB/LC_MESSAGES/coreutils.mo", O_RDONLY) =-1
ENOENT (No such file or directory)
write(2, "ls: ", 4)           = 4
write(2, "cannot access /python/", 22) = 22
open("/usr/share/locale/en_US/LC_MESSAGES/libc.mo", O_RDONLY) =-1
ENOENT (No such file or directory)
open("/usr/share/locale/en/LC_MESSAGES/libc.mo", O_RDONLY) =-1 ENOENT
(No such file or directory)
open("/usr/share/locale/en_GB/LC_MESSAGES/libc.mo", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=1474, ...}) = 0
mmap(NULL, 1474, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f30b2df1000
close(3)                      = 0
write(2, ": No such file or directory", 27) = 27
write(2, "\n", 1)              = 1
close(1)                      = 0
```

ההרצה של `strace` עם הפקודה:

strace ls /python/
 ls היא פקודה שמצויה בכל הקבצים והתיקות בתיקייה מסוימת) והתקינה python לא קיימת - No .such file or directory

בדוגמה הבאה מరיצים **strace ls python/** כאשר התקיימה כן קיימת:

```
stat("python/", {st_mode=S_IFDIR|0755, st_size=4096, ...}) = 0
openat(AT_FDCWD, "python/", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
fcntl(3, F_GETFD)            = 0x1 (flags FD_CLOEXEC)
getdents(3, /* 8 entries */, 32768) = 240
getdents(3, /* 0 entries */, 32768) = 0
close(3)                    = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x79ac55f24000
write(1, "proj1 proj2 proj3 proj4 proj"..., 41) = 41
close(1)                    = 0
```

הרצה של הפקודה **strace wc sample2.in** סופרת כמה בתים יש בקובץ:

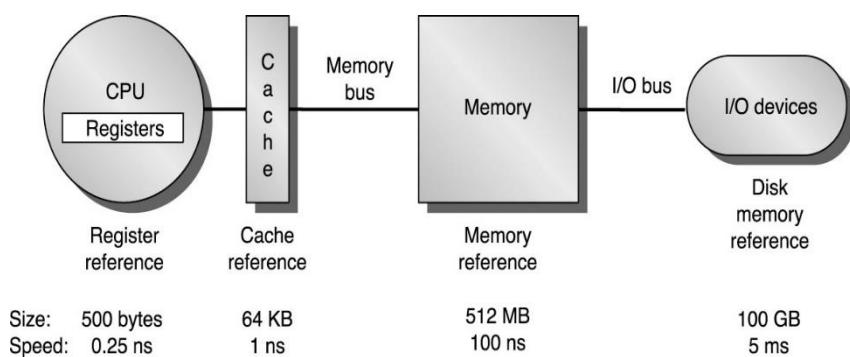
```
stat("sample2.in", {st_mode=S_IFREG|0777, st_size=490, ...}) = 0
open("sample2.in", O_RDONLY)      = 3
read(3, "\\" The path of the righteous man "..., 16384) = 490
open("/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache",
O_RDONLY) = 4
fstat(4, {st_mode=S_IFREG|0644, st_size=26066, ...}) = 0
mmap(NULL, 26066, PROT_READ, MAP_SHARED, 4, 0) =
0x7f81a4c88000
close(4)                      = 0
read(3, "", 16384)             = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 4), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f81a4c87000
write(1, " 1 96 490 sample2.in\n", 23) = 23
close(3)                      = 0
close(1)                      = 0
```

תרגול 2 – interrupts

חומרה ודיברונות

חזרה – דגשים על המעבד:

1. יכול להריץ ברגע נתון רק פקודה אחת.
2. המידע אליו הוא עבד שמור ברגיסטר, לא יכול לעבוד ישירות עם מידע מהזיכרון.
3. כל סוג ההוראות שהמעבד יכול לבצע בתובות בשפת מקונה:
 - פעולה עם מידע
 - פעולות אריתמטיות
 - Control flow



היררכיה זיכרון:

1. בתוך ה-**CPU** יש במה רегистרים שכמות המידע בהם לא גודלה אבל הגישה מהירה.
2. אחרי מגע ה-**cache** שיבול לאחסן מעט יותר מידע עם גישה איטית יותר.
3. **Main memory** – ה-RAM הוא חיצוני למעבד ובו נשמרות הוראות התוכנה והדעתה. נמוך ברגע שמכבים את המחשב. יכול להחזיק יותר מידע, לא מאד גדול, גישה איטית יותר מה-**CPU**.
4. מה שלל הדיסק, לא נמוך בכיבוי המחשב. הרבה יותר גדול מכל השאר. החישרונו הוא שהוא הרבה יותר איטי.

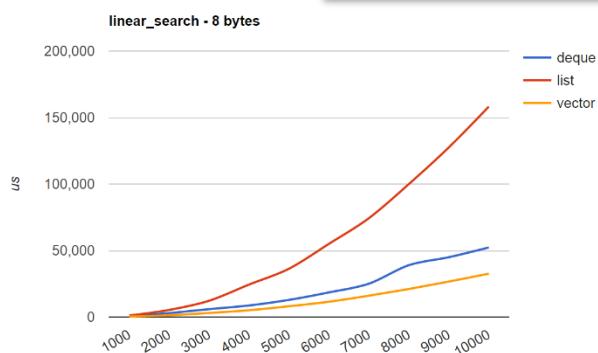
נתבונן בשתי תכניות למעבר על מטריצה – תכנית 1 עוברת שורה באשר בשורה אחת עוברים על כל העמודות, ו-2 קודם על העמודות. הסיבוכיות של שתי השיטות תהיה זהה ($O(n^2)$, אבל שיטה 1 תהיה יותר מהירה. למה? הזיכרון הוא איזשהו מערך אורך והתאים נשמרים בצורה רציפה, באיזשהו ווקטור אורך לפי השורות. בשקוריים מהזיכרון קוראים לפי איזשהו בлок ושמימים בו Cache. Cache – מה מספרים של השורה שמורים ב-**Cache** ולאחר מכן אין צורך לזכור. גישה 1 היא רציפה לזכרון והמעבר יותר מהיר. גישה 2 היא אקראית.

Approach 1

```
for (h=0; h<height; ++h){
    for (w=0; w<width; ++w){
        img[h][w] = 0;
    }
}
```

Approach 2

```
for (w=0; w<width; ++w){
    for (h=0; h<height; ++h){
        img[h][w] = 0;
    }
}
```



אם נשווה בין מבני נתונים של ווקטור, רשימה מקושרת ו-deque בך שוחצים לעבר על כל המערך בחיפוש לינארי, בעיקרו לשולשות המעבר הוא ($O(n)$). אבל ווקטור משתמש בגישה רציפה, ורשימה מקושרת היא אקראית. Deque יש במה בלוקים ששמורים באופן רציף אבל מחוברים ברשימה מקושרת.

User mode & kernel mode

הגדרות

← **Program (תוכנית)** – קובץ בינהרי שנitin להריץ אותו - executable. המידע שומר על הדיסק כל עוד לא הרכינו אותו.

← **Process (תהליך)** – קורה ברגע שמריצים קוד, איזשהו מופע של התוכנית. כל תהליך מקבל אזרור משלה בזיכרון שמחולק לאזרור של הקוד ולאזרור של הדאטה שלו. כאשר יש מעבד אחד יש רק process אחד.

← **פלט קלט (Input/Output)** – התקני החומרה אליהם נשלח הקלט או מהם נפלט הפלט.

Kernel – הליבה של מערכת הפעלה, שליטה מלאה על כל מה שקרה במחשב. כל תוכנית שרצה במצב kernel היא תוכנית **trusted** – יכולה לעשות כל מה שהיא רוצה. תוכנית שהיא **untrusted** מוגבלת ורצה ב-user mode. בקוד שירוץ במערכת יהיה בkernel וקוד שאנו חנו נרץ יהיה ביוזר.

User mode vs. kernel mode

ה-CPU יכול להיות ב-user mode או ב-kernel mode.

– User mode

- לא ניתן לבצע פקודות מסוימות, כמו halt.
- ניתן לגשת רק לזיכרון שהוקצה.
- אין גישה ישירה לחומרה.

– Kernel mode

- המערכת מריצה תוכניות שהן trusted. יכולה לגשת לכל מקום בזיכרון ואין מניעה לביצוע פעולה כלשהי.
- שולט ב-processing.

מטרת מערכת הפעלה, שהיא בעצם kernel, היא לחתת לתהליכי רוץ. כמו כן מערכת הפעלה נוותנת שירותים-processes האחרים לגשת למקומות בהם לא יכולים לגשת.



מעבר בין מצבים

System calls – מנגנון בו שולחים בקשה ל-OS כדי לקבל שירות לגשת למקום שלא ניתן לגשת. אינטראפיס שמערכת הפעלה מספקת ל-processes האחרים. כאשר שולחים system call המעבד עובר במצב של kernel, מבצע את הבקשה, ואחריו שבוצעה חזרה מ-kernel ל-user.

↓ System calls

↓ Privilege instructions

↑ Interrupts

Interrupts

חשוב לסבך בין רכיבי החומרה למערכת הפעלה, וה-interrupts עוזרים לכך. פתרון אפשרי: לולאה שרצה על כל רכיבי החומרה ובודקת אם יש מידע לרכיב שציריך לעבוד. הבעיה בפתרון היא שהוא בזבזני, יוכל לגרום לאיבוד מידע. לכן קיימים interrupts (פסיקות) שיודיעו ל-CPU בשיש פעולה שהסתירה.

interrupt: סיגנל שהמעבד מקבל ומשנה את סדר ביצוע הغانבים. ברגע ש-interrupt קורה המעבד מפסיק את כל ה-processes, מטפל ב-interrupt ובশמשים חזר לפעולות הקודמות.

Interrupt controller – רכיב חומרה שמחובר להתקנים האחרים, כל פעם שאחד מהם רוצה להעביר הודעה הוא שלוח אותה חשמלי וה-controller מעביר את הודעה המעבד יחד עם מי העביר את הודעה.

- interrupt הוא א-סינכרוני וכן גם הטיפול בו, המשתמש לא מודע לזה שהוא קרה.

סוגי interrupts

External/hardware – מגיע מרכיבי החומרה. דוגמאות:

- לחיצה על המקלדת
- מעבר עבר
- סיום קריאת מידע מדיסק
- טינמר

Internal/software – בא מהקוד, exception. קורה תוך כדי ריצת התוכניות. דוגמאות:

- חליקה ב-0.
- שגיאת סגמנטציה.
- Privileged instruction
- Invalid instruction
- Page fault

external interrupts עם handler

פונקציה שמוראה במערכת הפעלה שהמטרה שלה לטפל ב-interrupt ספציפי. המעבד יודע איזה פונקציה לבצע לפני interrupt vector – ווקטור בזיכרון במקום ידוע שמכיל פוינטורים לפונקציות. ה-interrupt controller שלוח למעבד את מספר ה-interrupt שקרה. המעבד ייגש לאותו האינדקס בווקטור, ויבצע את הפונקציה הכתובה בכתובת אליה מצביע הפוינטור באינדקס.

המכניזם:

1. – **Getting the interrupt** – ה-interrupt controller שלוח למעבד סיגナル אלקטרוני שמורה על הצורף בטיפול ב-interrupt ועיצוב התוכנית.

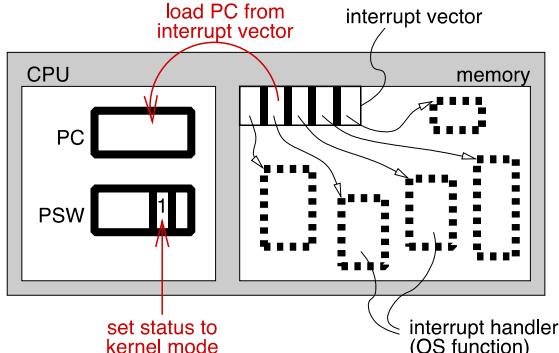
2. – **Saving current size** – בשעוצרים את התוכנית שרצה צריך לשמר את המצב הנוכחי של ה-process. לאחר מכן יש רגיסטרים שצורך לשמור בהם PC, ולנקוט את ה-CPU בשביול ה-interrupt handler.

3. – **Transfer control & service the request** – העברת השליטה דרך הווקטור לפונקציה שמטפלת

באותו interrupt. עובר ל-mode kernel. הדלקת הרגיסטר (program status word) PSW, שבו אחד מהביטים מתאר את המצב הנוכחי – user או kernel. אח"כ הוא מחפש בווקטור את המיקום בו נמצאת הפונקציה המתאימה.

4. – **Previous state is restored** – באשר מסייםים חזרם לקוד, משחזרים את המצב הקודם ושמוראים את הערךם ברגיסטרים המקוריים.

5. – **Return control** – השליטה על ה-CPU חוזרת ל-process שרך קודם.



התמודדות עם internal interrupt

- לרוב exceptions יגרמו לעצירת התוכנית, ויש exceptions מסוימים שמערכת הפעלה יודעת להתמודד איתם.
- באשר exception קורה, נשמרת הכתובת של ה-instruction שהביאה לכך.
- ניתן לתת למערכת הפעלה אופציה להריץ handler כדי לטפל בבעיה, ו-handler יכול להיות גם מהיזדר (try and except).
- אם יש handler הוא ירוץ עד שישים ואז התוכנית תמשיך. אם אין, התוכנית תקרוס.
- התמודדות זהה להתמודדות עם interrupt חיצוני, אבל ללא השלב הראשון, כי ה-CPU עצמו שולח את ה-interrupt.

– Trap

Trap הוא סוג של exception, בקשה שאנחנו מבקשים מהמעבד לבצע מהריצה הרגילה של התוכנית, ומדילקה interrupt ספציפי שגורם למעבר ל-kernel mode. הדרך בה calls system מוממשות.

דוגמה – int3:

הפקודה היא call system שחדיבגר נוספת וגורמת לעצירת התוכנית באיזושהי .breakpoint

```
\`breakpoint.c
int main() {
    int i; for(i=0; i<3;i++) {
        __asm__("int3");
    }
}
```



```
(gdb) run
Starting program: /tmp/a.out
Program received signal SIGTRAP, Trace/breakpoint
trap. 0x00000000004004fb in main ()
```

דוגמה – open:

open שייכת לסדריה הסטנדרטית ויש בה ביצוע ישר של call system. ברגע שיש קריאה לפונקציה: ← trap ← קריאה ל-() ← שבירת הארגומנטים במקום מגדר בזיכרון ששייך למערכת הפעלה ← Kernel mode) טיפול ב-trap ← ביצוע הפונקציה ← שבירת הפלט במקום מגדר בזיכרון ← (User mode) open תקרא את המידע בזיכרון ותחזר ← (User mode) אותן.

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
...
int fd;
fd=open("/tmp/foo");
if( fd < 0 ) {
    perror( "Error opening file" );
    //equivalent to:
    //printf("Error opening file:
    %s\n",strerror(errno));
}
```

בקוד מראה איפה נמצא הקובץ וגם מסמל אם הפתיחה הצליחה או לא. לעיתים כדאי לעשות handlers مثل עצמנו, כך שאם הפונקציה תעוף נדע מה:

ערכים נוספים אפשריים:

| Error name | Error code (number) | Message |
|------------|---------------------|---------------------------|
| ENOENT | 2 | No such file or directory |
| EINTR | 4 | Interrupted system call |
| EIO | 5 | I/O error |
| EACCES | 13 | Permission denied |
| EBUSY | 16 | Device or resource busy |

פקודות man

איןטרפייס, מחולק לקטגוריות:

executable programs or shell commands – **Man 1** .1system calls – **Man 2** .2C library – **Man 3** .3

MKDIR(1) User Commands MKDIR(1)

NAME
mkdir - make directories

SYNOPSIS
mkdir [OPTION]... DIRECTORY...

DESCRIPTION
Create the DIRECTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

-m, --mode=MODE
 set file mode (as in chmod), not a=rwx - umask

-p, --parents
 no error if existing, make parent directories as needed

-v, --verbose

Manual page mkdir(1) line 1 (press h for help or q to quit)

דוגמה:הרצת man 1 mkdir מותנת תיאור לפקודה
של ה-shell .shell

MKDIR(2) Linux Programmer's Manual MKDIR(2)

NAME
mkdir - create a directory

SYNOPSIS
#include <sys/stat.h>
#include <sys/types.h>

```
int mkdir(const char *pathname, mode_t mode);
```

DESCRIPTION
mkdir() attempts to create a directory named pathname.

The argument mode specifies the permissions to use. It is modified by the process's umask in the usual way: the permissions of the created directory are (mode & ~umask & 0777). Other mode bits of the created directory depend on the operating system. For Linux, see below.

The newly created directory will be owned by the effective user ID of the process. If the directory containing the file has the set-group-ID

Manual page mkdir(2) line 1 (press h for help or q to quit)

הרצת man 2 mkdir מסבירה על הקראיה
ל-system call .system call

תרגול 3 – unix signals threads

signals

סיגנלים – נוטיפיקציה ל-process שנשלחו ע"י מערכת הפעלה ומטופל ע"י התהיליך. ברגע שיש סיגנל התהיליך צריך להפסיק את הפעולות שעשה ולטפל בו, ולאחר הטיפול לחזור לפעולות.

טריגרים לסיגנלים

1. קלט מהמשתמש – המשתמש שולח סיגナル ע"י לחיצה על המקלדת. קורה interrupt ומערכת הפעלה שולחת סיגナル לתוכנה לעזרו.
2. בקשה ממערכת הפעלה לשולוח סיגナル בזמן מסוים – interrupt יגרום למערכת הפעלה לשולוח סיגナル.
3. מערכת הפעלה או תהיליך אחר שולח סיגナル.
4. Software interrupt – מערכת הפעלה מקבלת interrupt ושולחת סיגナル להפסקת התהיליך.

סיגנלים שנייתן לשולוח מהמקלדת

- Ctrl-C – סיגナル לסגירת תוכנית **SIGINT**.
 - \ - סיגナル לסגירת התוכנית **SIGQUIT**.
 - Ctrl-Z – עצירת התוכנית עד ששולחים סיגナル להמשך התוכנית **SIGTSTP**.
- דרך הפקודה **kill** ניתן לשולח את הסיגנלים:

kill [option] pid

- ID – **pid**, באמצעות המספר אפשר לשולוח סיגנלים לתהיליך מסוים.
- בכתיבת **-kill** רואים את האפשרויות לשילוח סיגנלים.
- הפקודה **fg pid** ממשיכה את התהיליך.

שליחה מתהיליך אחר: יש גם user sig שזאת תוכנית אחת ששולחת אחרת.

int kill(pid_t pid, int sig)

- Strace עוקב אחרי סיגנלים.

טיפול בסיגנלים

כל סיגナル יש איזושהי התנהלות דיפולטיבית שהטהיליך מבצע כדי להתמודד אותו כמו ליצאת מהתוכנית, להמשיך, להתעלם ועצור. יש אפשרות לשנות את התמודדות הדיפולטיבית באמצעות הפונקציה **signal** – שמקבלת את הסיגナル ואת הפונקציה שתתפעל באשר ישלוח.

signal(signum,newHandler)

- לא אפשרי לסיגנלים **KILL** ו-**STOP**.

דוגמה: התוכנית רושמת לסיגナル SIGINT את הפונקציה **catch_int** שתיקרא ברגע שייתקבל הסיגナル. בפונקציה עצמה מגדירים שפעם הבאה שייקרא סיגナル נוסף SIGINT הפונקציה תקרוא שוב וכן קובעים שה-**handler** זה יהיה קבוע.

```
#include<stdio.h>
#include <unistd.h>
#include <signal.h>
void catch_int(int sig_num) {
    //install again!
    signal(SIGINT, catch_int);
    printf("Don't do that\n");
    fflush(stdout);
}
int main(int argc, char* argv[]) {
    signal(SIGINT, catch_int);
    for ( ; )
        //wait for a signal
        pause();
}
```

ניתן להשתמש בפונקציות הבנויות הבאות:

- **SIG_IGN** – מביאה להטעלנות מהסיגנל.
- **SIG_DFL** – שחרור ההתנהגות הדיפולטיבית של הסיגナル.

Masking signals

לפעמים נרצה להתעלם באופן זמני מסיגנלים וליחסם אותם, כדי שלא יפריעו ליריצה של תהליכי החסימה מתבצעת ע"י הפונקציה **sigprocmask** – הפונקציה מקבלת קבוצה של סיגנלים ופרמטר **how** שמקבל אחד משלושת הערךים הבאים:

1. **SIG_BLOCK** – להוסיף סיגנל
2. **SIG_UNBLOCK** – למחוק את הקבוצה
3. **SIG_SETMASK** – לקבוע את הקבוצה

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGTERM);
sigprocmask(SIG_SETMASK, &set, NULL);
//blocked signals: SIGINT and SIGTERM
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_BLOCK, &set, NULL);
//blocked signals: SIGINT, SIGTERM, SIGALRM
sigemptyset(&set);
sigaddset(&set, SIGTERM);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_UNBLOCK, &set, NULL);
//blocked signals: SIGINT and SIGALRM
```

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset)

Oldset הוא פוינטר לסט שדרסנו במקרה שנרצה לשחרר, אם זה לא רלוונטי נשלח NULL.

```
#include<stdio.h>
#include <unistd.h>
#include <signal.h>
void catch_int(int sig_num) {
    printf("Don't do that\n");
    fflush(stdout);
}
int main(int argc, char* argv[]) {
// Install catch_int as the
// signal handler for SIGINT.
    struct sigaction sa;
    sa.sa_handler = &catch_int;
    sigaction(SIGINT, &sa, NULL);
    for ( ; )
        //wait until receives a signal
        pause();
}
```

sigaction

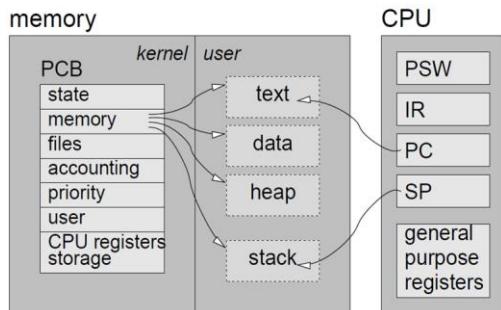
```
int sigaction(int sig, struct sigaction *new_act, struct sigaction *old_act);
```

פונקציה קבועה – קובעים לSIGNEL איזשהו handler עד שימושים, וחוסמת Signalsים כך שיהיו חסומים רק בהנדלה. מגדירים struct מסוג sigaction ושמם בו פונטט לפונקציית handler ול-*mask*, signal.

אם לא קובעים signal mask ההתנהגות הדיפולטיבית היא לחסום את הSIGNEL עליו עשינו את ה-.handler.

Threads

בביכול כל תהליך הוא בפני עצמו, אבל בעצם יש time sharing ואיזושהי הרצה לשירותים. בשביל זה המעבד צריך לשומר ב-PCB, מבנה נתונים ב-kernel memory, את סביבת הפעולה של התהליך –

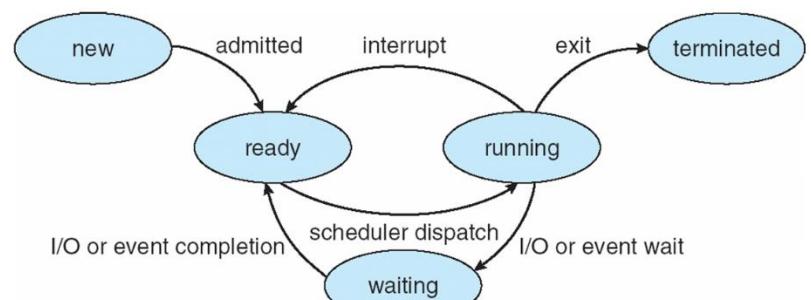


- הרגיסטרים של ה-CPU (PC, SP) (PC, SP) (PC, SP).
- זיכרון.

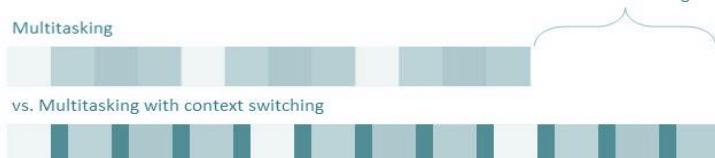
- סביבת עבודה (קבצים וכו').
- מידע על עדיפות התהליכי.

- הרשותות של user מסוים להרצתה של תהליכים.
- מצבו שבו התהליך נמצא ברגע.

דיאגרמת המצבים של התהליכים:



Context Switching



זמן החלפת משימות contexts switch מערכות הפעלה מבצעת פעולות לשם כך ורקיים overhead. כדי למדוע את זה ניתן להשתמש ב-threading.

Thread

Threads – מעין משימות שאנו רוצים לבצע. דומה לתהילך רק ש-thread חי בתוך תהילך. לכל Thread יש משימה משלו, וכל אחד בצהה רץ בצויה "מקבילית" בלבד, אך שמתאפשרת מקביליות בין המשימות. המעבר בין ה-threads מביא ל-overhead קטן יותר.

- לכל thread יש stack ורגיטרים ב-CPU משלו.
 - מה שימושי בינהן threads שונים של תהילך זה ה-heap, איזור הקוד, הדיברונות קבצים פתוחים וכו'.
- המשותף בין threads

שני סוגי של threads

.1 – **User level threads** – היוזר ממש, מערכת הפעלה לא מודעת לכך ו מבחינה רץ תהילך אחד.

.2 – **Kernel level threads** – מערכת הפעלה מודעת ל-threads ומנהלת אותם.

User level threads

ממומשים ע"י ספירת thread, שמכילה קוד ליצירה, סיום, החלפה ו-scheduling של ה-threads. מערכת הפעלה לא מודעת למשמעות זהה, במקרה שעשינו block כל תהילך ייחסם וכך כל ה-threads בתוך התהילך.

יתרון: החלפה בין thread אחד לאחר קורית ב-user mode ובר מחיר ה-context switch יהיה פחות יקר, תקורה נמוכה יותר.

מעבר בין threads

1. עצירת את thread הנוכחי שרצה.
2. שמירת את המצב שלו באיזשהו מקום.
3. בחירת thread אחר וקפיצה אליו.

המעבר מתבצע בעזרת שתי פונקציות מובנות:

- **Sigsetjmp** – שומרת מצב כוכבי של thread ב-struct thread ב-SP,CPU, מיקום ו-mask (לפי ה.Parameter). (savesigs)

```
sigsetjmp(sigjmp_buf env, int savesigs)
```

הפונקציהמחזירה 0 אם הגענו לשורה זו במהלך מתחילך שמירה, במצב אחר, אם הגיענו דרך siglongjmp היא תחזיר פרמטר של kp.

הברך רוח שומר את מצב ה-SP,CPU,signal mask,PC. מה שלא נשמר זה משתנים גלובליים, משתנים דינמיים, ומה שימושי בכל ה-threads האחרים.

Siglongjmp – קופצת למקומ שעשינו לו Sigsetjmp, ומשוחררת מה שדרוש. •

```
sigsetjmp(sigjmp_buf env, int savesigs)
```

מניחים שנשמרו בבר ערכיהם ל-env.

Kernel level threads vs user level threads

יתרונות:

| User level threads | Kernel level threads |
|--|---|
| <ul style="list-style-type: none"> Switching between threads is cheaper Often don't need to worry about concurrent access to data structures Scheduling can be application-specific | <ul style="list-style-type: none"> Blocking is done on a thread level Multiple threads can possibly be executed on different processors Scheduler can make intelligent decisions amongst threads and processes |

חסרונות:

| User level threads | Kernel level threads |
|--|---|
| <ul style="list-style-type: none"> Can't enjoy the benefits of a multi-core machine One blocked thread blocks the entire process Can suffer from poor OS scheduling | <ul style="list-style-type: none"> Greater cost for switch between threads Need to pay more attention to shared resources |

מה עדיף?

לפעלת OS קرنל (שעדין יהיה אפשר למשל לקבל פקודות מהמקלחת)

אם רצים למנוע את התקורה עדיף יוזר.

תרגול 4 – threads

Threads pools

כנich שיש משימה שרוצים לבצע, מחלקים אותה למספר תתי משימות וכל thread מבצע תתי משימה. הבעה שיש ליצור threads pool, thread לכל משימה, תקורה גדולה. המערכת יכולה לגמור את היזכרון ולהשתמש בהרבה משאים בו זמןית.

פתרון: **threads pool** – יצירתpool של הרבה threads מראש. בשיש משימה חדשה, נלקח thread חדש מהpool, מבצע את המשימה, ווחזר לפול בשימושים.

כיתן למשתמש בaporen פשוט ע"י thread וראשי שמחכה לבקשתו, בודק אם יש thread פנוי בpool. אם אין, יחכה שייהיה ואז יrix אותו:

```
While (true) {  
    Wait for request r  
    Wait for thread t in the pool to be  
    available  
    Run r on t  
}
```

הבעיה: לא תמיד נרצה לחכות שייהיה thread פנוי, נרצה שה-threads יסונכרנו אחד עם השני.

פתרון אפשרי:

ישנו תור של משימות, כל פעם שהתוכנית רוצה לבצע משימה היא מכניסה אותה לתור וועוזבת אותה. ה-threads בודקים את התור ומושכים ממנה משימות לבצע.

סיכום בשימוש בpool

דיליפות (לא יוצרים threads חדשים אחרי שייצרים את הפול):

- קירה exception וה-thread קרס.
- ה-thread נתקע (למשל מחכה לקלט מהמשתמש ולא מקבל, נתקע בולאה אינסופית).

פתרון: הגבלת זמן הריצה של thread על משימה, אם אחרי הזמן הזה ה-thread לא סיים לבצע את המשימה הריצה תופסק והוא יוחזר לפול.

גודל הפול

בעיות ביצירת pool גדול:

- תקורה גדולה (בעיה קטנה יחסית).
- צריכת יתר של זיכרון ומשאים.
- תקורה מהרבה context switch.

בעיות ביצירת pool קטן:

- חוסר נגישות של כל המעבד.
 - ה-threads יתפסו ע"י משימות ארוכות ומשימות קצרות יჩכו לסיום, מה שגדיל את זמן ההמתנה הממוצע
- הפתרון:** יצירתpooloccupied בגודל האופטימי בהתאם למשימות התוכנית.

דוגמה כללית לפתרון:

נגדיר -

(ST) – הזמן שה-thread צריך לrhoץ על המעבד כדי לסיים את פעולתו.

(BT) – הזמן בו הוא חסום.

$$\text{הגודל שוכרח לפול הוא} \frac{\text{BT}}{\text{ST+1}}$$

Threads management

- הספריה pthread יוצרת threads ב-kernel.
- .pthread מתקבל פרמטר pthread และ מזהה thread ID דרך הארגומנט *thread.
 - .pthread_create זה פונקציהשה-thread יבצע.
 - .start_routine זה הארגומנט שיועבר לפונקציה שהועבירה ב-thread.
 - .arg זה הארגומנט arg.

```
int pthread_create (
    pthread_t *thread,
    const pthread_attr_t *attr=NULL,
    void *(*start_routine) (void *),
    void *arg);
```

סיכום threads

- קריאה מה-thread לפונקציה pthread_exit(void* status) של הספריה pthread שמסיימת אותו.
- יכול לחת ערך החזרה ש יכול להיות מועבר לאחר ע"י שימוש בפונקציה pthread_join.
- .return ממבצעת start_routine.
- .pthread_cancel Thread לאחר קורא לפונקציה ומ לבטל את הריצה של ה-thread.
- התוכנית עצמה הסתיימה:
- אחד threads קורא ל-.exit.
- ה-thread הראשי הסתיים.

דוגמה:

```
#define NUM_THREADS 5
void *PrintHello(void *arg) {
    int *index = arg;
    printf("\nThread %d: Hello World!\n", *index);
    pthread_exit(NULL);
}
int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    int res, t;
    for(t=0;t<NUM_THREADS;t++){
        printf("Creating thread %d\n", t);
        res = pthread_create(&threads[t], NULL,
PrintHello, (void *)&t);
        if (res<0){
            printf("ERROR\n");
            exit(-1);
        }
    }
}
```

Joining threads

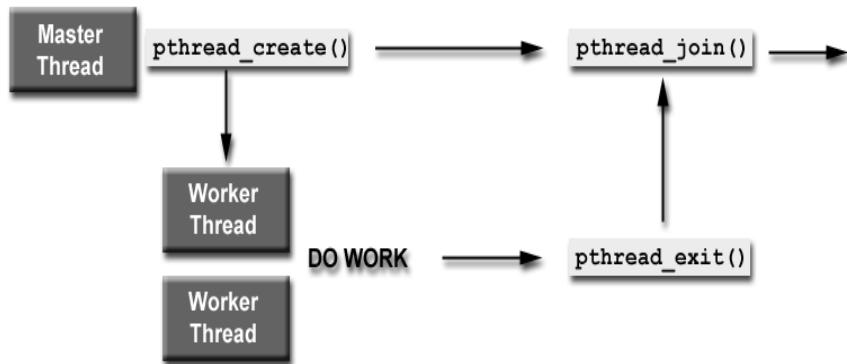
נרצה לבצע דברים אחרי שהובטח לנו ש-thread מסויים סיים, והפונקציה `pthread_join` מאפשרת זאת.

- thread ה-`pthread_t`
- `value_ptr`

```
int pthread_join(pthread_t thread, void **value_ptr);
```

אם ה-thread סיים הפונקציה לא עשו כלום. אם לא, ה-thread שקרא לפונקציה הופך להיות blocked ווזור ל-ready רק אחרי שה-thread אליו חיבה יסתיים.

נשתמש בה למשל אם יש איזשהו thread שכדי להמשיך צריך לחכות ל-thread אחר שיבצע משימה:



از בקוד בדוגמה הקודמת בו ייצרנו מ-`main` thread, נctrיך להוסיף לולאה את הקטע הבא:

```
// main thread waits for the other threads
for(t=0;t<NUM_THREADS;t++) {
    res = pthread_join(threads[t], (void **)&status);
    if (res<0) {
        printf("ERROR \n");
        exit(-1);
    }
    printf("Completed join with thread %d status= %d\n",t, *status);
}
```

Mutex

דוגמה לקטע קוד קרייטי בו ה-counter משותף לשני ה-threads, יש שלוש אופציות ל-counter בסיום ההרצה – 6 / 5 . לא נרצה שקטע הקוד יבוצע בו זמנית ונשתמש באלגוריתמים למניעה הדדית.

Thread 1

```
int a = counter;
a++;
counter = a;
```

Thread 2

```
int b = counter;
b--;
counter = b;
```

Mutex work flow

mutex – משתנה כמו מנעול שיכל להיות במצב געול/פתוח. מובטח ע"י מערכת הפעלה שרק thread אחד יוכל לנעול אותו בכל רגע.

← יצירת mutex.

← מספר threads מנסים לנעול אותו ברגע נתון.

← אחד thread.

← אותו thread מבצע את המשימות שלו בקטע הקרייטי והשאר חסומים.

← משחרר את mutex.

← thread אחר מצליח לנעול אותו וחוזר חלילה.

• לבסוף mutex משוחרר (כי הוא משאב של מערכת הפעלה).

יצירה, נעליה והריסה של mutex

mutex הם משתנים עם טיפ של `t_mutex`.

• ניתן ליצור את המשתנה בצורה סטטית:

`pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;`

• או בצורה דינמית:

`pthread_mutex_init(mutex, attr)`

• הריסת mutex:

`pthread_mutex_destroy(mutex)`

• נעליה:

`pthread_mutex_lock(*mutex)`

אם המוטקס געול, ה-thread שקרה לפונקציה נחסם וישורור ורק בשחרומו ייפתח.

• פתיחה:

`pthread_mutex_unlock(*mutex)`

הפונקציה תחזיר שגיאה במקרים הבאים:

- ניסיון לפתיחת mutex פתוח.

- ניסיון לפתיחת mutex שנעל thread אחר.

mutexים נועדו רק למניעת הדדיות.

נבחן לדוגמה הקודמת, ונראה קטע קוד מותוקן עבורה (לא כולל בדיקת תקינות ערך החזירה):

Thread 1

```
pthread_mutex_lock (&mut_counter);
int a = counter;
a++;
counter = a;
pthread_mutex_unlock (& mut_counter);
```

Thread 2

```
pthread_mutex_lock (& mut_counter);
int b = counter;
b--;
counter = b;
pthread_mutex_unlock (& mut_counter);
```

Deadlock

הבעיה: בדוגמה הבאה יש שני משתנים שונים a ו-b, ושני מוטקסים ששמורם עליהם. יתכן מצב שבו ננעלו את המוטקס של a מ-1 thread, אחריו נריץ את 2 thread וונעלו את המוטקס של b, ואחריו נעבר לנושא לנעול את המוטקס של a וונחסום את 2 thread. נעברו ל-1 thread 2 thread 1 ויחסם גם הוא.

Thread 1

```
lock(a_mutex)
lock(b_mutex)
a=b+a;
b=b*a;
unlock(b_mutex);
unlock(a_mutex);
```

Thread 2

```
lock(b_mutex)
lock(a_mutex)
b=a+b
a=b*a;
unlock(a_mutex);
unlock(b_mutex);
```

Monitors

בעיות שמצריכות סינכרוןiza של threads

- **Bounded buffer** – תור בגודל סופי, consumer שימושם ו-producer שוחף דברים לתור. בשהתור ריק ה-consumer עצר, וה-producer מעיר אותו בשמכניס דברים לתור. בשהתור מתמלא ה-producer עצר וה-consumer מעיר אותו בשיש מקום.
- **Readers-Writers** – למשל רשיימה מקורתת שכלי threads ניגשים אליה, אבל לא נרצה שייגשו אליה בו זמינות (ובתבו, יקראו ויכתבו) אבל יתכן שנרצה כמה קוראים שיקראו ממנה.
- **Barrier** – במה threads מבצעים יחד פעולה בשלבים, ונרצה שכלי שלב יבוצע ע"י כל ה-threads, ומעבר לשלב הבא יהיה רק בשכולם יסיימו. איזשהו מחסום בין השלבים.

לא היינו מצליחים לפתרור את הבעיה עם מוטקס, נוצרת לולה

אינסופית:

Thread 1

```
While (true){
    mutex_lock (varMutex)
    if (canUseVar){
        //Use var;
        canUseVar = false;
        break;
    }
    mutex_unlock (varMutex)
}
```

Thread 2

```
mutex_lock(varMutex)
init(var);
canUseVar=true;
mutex_unlock (varMutex)
```

הפתרון: מוניטורים (conditional variables)

שתי פונקציות עיקריות:

- כדי ללחכות (חוסם את ה-thread ומשחרר את המוטקס):

wait(monitor , mutex)

- לשחרר (לעוזר ע"י סיגナル לנעול את המוטקס):

signal(monitor)

נרצה להשתמש בסיגナル באשר איזשהו תנאי מתקיים.

דוגמה:

Thread 1

```

mutex_lock (varMutex)
if (!canUseVar){
    cv_wait(varCV, varMutex)
}
//Use var;
mutex_unlock (varMutex)

```

Thread 2

```

mutex_lock(varMutex)
init(var);
canUseVar=true;
cv_signal (varCV)
mutex_unlock (varMutex)

```

ה-variable conditional variable נקרא varCV וקוראים לסיגナル כדי להעיר את מי שמחכה. Thread 1, במקרה שכוננו ל-if, מעביר את varCV ל-wait ונחסם. יתעורר רק בשיקראו לסיגナル. Wait מקבל איזשהו מוטקס במקרה בו הפעולה נעשית בקטע קוד קרייטי שבו המוטקס בעול. משחרר את המוטקס כדי שנוכן לעבור ל-thread 2.

פונקציות חשובות של pthread

- `pthread_cond_init(pthread_cond_t *cv, NULL);`
- `pthread_cond_destroy(pthread_cond_t *cv);`
- `pthread_cond_wait(pthread_cond_t *cv, pthread_mutex_t *mutex);`
- `pthread_cond_signal(pthread_cond_t *cv);`
- `pthread_cond_broadcast(pthread_cond_t *cv)`

bounded buffer using CVData structures

1. `Queue<T> q`
2. `pthread_mutex_t qM,`
3. `pthread_cond_t qCV`

Reader flow

1. `pthread_mutex_lock (qM)`
2. If (`q.empty()`)
 1. `Wait (qCV, qM); //waiting for an element to be written`
 3. `Element e = q.pop(); //here we have both a lock and element`
4. `Pthread_mutex_unlock (qM)`
5.

Barrier example

```

#define NTHREADS 5
pthread_mutex_t *lock;
pthread_cond_t *cv;
int ndone;
typedef struct {
    int id;
} TStruct;
main() { //Checking of return values omitted for brevity
    TStruct ts[NTHREADS];
    pthread_t tids[NTHREADS];
    int i;
    void *retval;
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cv, NULL);
    ndone = 0;
    for (i = 0; i < NTHREADS; i++) {
        ts[i].id = i;
    }
    for (i = 0; i < NTHREADS; i++)
        pthread_create(tids+i, NULL, barrier, ts+i);
    for (i = 0; i < NTHREADS; i++)
        pthread_join(tids[i], &retval);
    printf("done\n");
}
void *barrier(void *arg) { //Checking of return values omitted for brevity
    TStruct *ts;
    int i;
    ts = (TStruct *) arg;
    printf("Thread %d -- waiting for barrier\n", ts->id);
    pthread_mutex_lock(lock);
    ndone = ndone + 1;
    if (ndone < NTHREADS) {
        pthread_cond_wait(cv, lock);
    }
    else {
        pthread_cond_broadcast(cv);
    }
    pthread_mutex_unlock(lock);
    printf("Thread %d -- after barrier\n", ts->id);
}

```

ב-main יש משתנה גלובלי שמספר כמה הגיעו למחסום ומערך של ה-threads.
 וצרים את ה-threads ע"י קרייה ל-thread_create שבל אחד יירץ את הפונקציה barrier.
 אחרי שה-threads יסתינו ב�行ה עליהם join בollowה.
 הפונקציה barrier מדפסה thread לפני המחסום ואחריו. בתנאי if, כל עוד thread-if הוא לא החמישי שהגיע
 למחסום הוא יচכה. אם הוא בן החמישי, עושים broadcast, מערימים את כל ה-threads.threads.לבסוף יודפס:

```

Thread 0 -- waiting for barrier
Thread 1 -- waiting for barrier
Thread 2 -- waiting for barrier
Thread 3 -- waiting for barrier
Thread 4 -- waiting for barrier
Thread 4 -- after barrier
Thread 0 -- after barrier
Thread 1 -- after barrier
Thread 2 -- after barrier
Thread 3 -- after barrier
done

```

semaphores

הם במו מוטקסים שמאפשרים ריצה של כמה threads בקטע קוד הקרייטי, נרצה mutual exclusion רחב וגייש יותר.

- את ה-semaphores ניתן לאותל לכל ערך שהוא unsigned int בפונקציה init:

```

sem_t semaphore;
int sem_init(sem_t *sem, int pshared, unsigned int value);

```

- הרישה בפונקציית :destroy

```
int sem_destroy(sme_t *sem);
```

- פונקציית wait שמורידה ב-1 את הערך של ה-semaphore

```
int sem_wait(sem_t *sem);
```

- אם הערך הוא 0, ה-thread שניסה לעשות את זה ייחסם, אך ש-thread אחר יקרה לפונקציה post שתעלה את הערך של ה-semaphore ב-1:

```
int sem_post(sem_t *sem);
```

ה-threads שניسو להורד ל-0 יכולם להתעורר ולהעלות ל-1. אין הבטחה של סדר.

תרגול 5 – Concurrency

The critical section problem

Concurrency – חילוק עבודה בין threads שונים, חילוק בעיה לתחתי בעיות.

בערך הרעיון טוב אבל מסובך:

- בsharp;ץירק לגשת למבני נתונים משותפים ויש צורך בסינכרון.

- סבנה של deadlock.

- thread Starvation שמחבה להיכנס לקטע הקרייטי ולא נכנס.

回忆录

- **החלק הקרייטי:** קטע קוד רגיש שיש בו גישה למשאים משותפים.
- **אלגוריתמים של מניעה הדדית** נועדים למניעת ביצוע בו זמנית של החלק הקרייטי.

דוגמה – int problem

שני threads שמבצעים בו זמנית את הפקודה $50 = X + A$ כאשר X הוא משתנה משותף. הפקודה מinterpretת עצם לשולש פקודות אסמבלי:

```
A= lw(&X)      // get from memory to register
B= add(A,50)    // add two registers
sw(B, &X)      // store the first register to the memory.
```

הקטע קוד בו הפקודה מבוצעת נחשב לחלק קרייטי.

בעיה שיכולה להיווצר למשל היא בשמבצעים אחרי context switch השורה השנייה Thread 1 לא מספיק לשמר את הערך בזיכרון ולא יוכל הגיעו לערך 100

Thread 1

```
A= lw(&X);    // A=0
B= add(A,50); // B=50

sw(B, &X);   // X=50
```

Thread 2

```
A= lw(&X);    // A=0
B= add(A,50); // B=50

sw (B, &X ); // X=50
```

dogma – linked list problem

insert after(current, new)

```
new->next=current->next
current->next=new
```

1. פונקציה insert_after שמקבלת current וアイבר חדש שנרצה להוסיף לאחרי new.

2. פונקציית remove_next שמוחקת את האיבר אחרי ה-current.

remove next(current)

```
tmp=current->next;
current->next=current->next->next;
free(tmp);
```

```
[thread 0]
new->next = cur->next

[thread 1]
remove_next (cur):

[thread 0]
cur->next = new
```

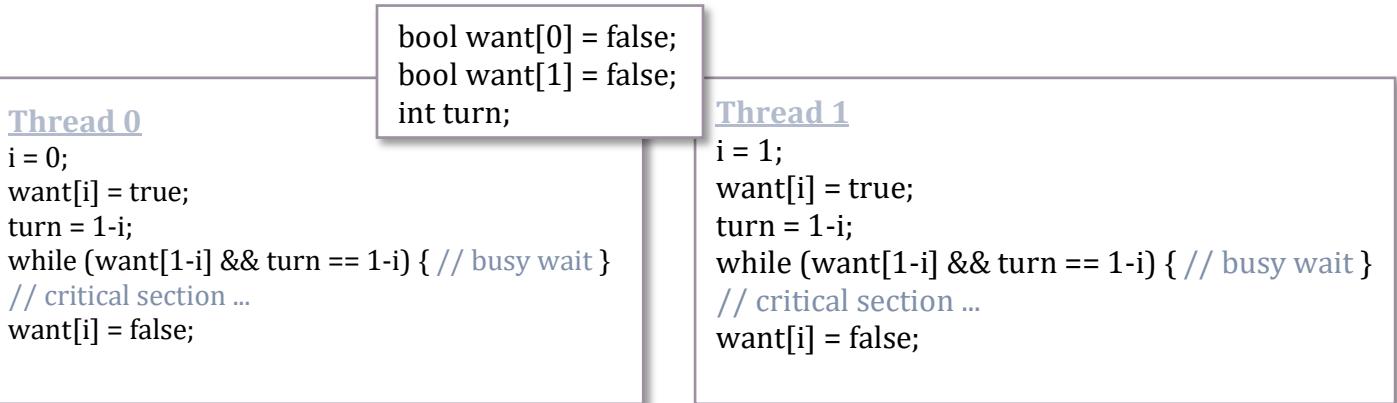
בעיה לדוגמה להיווצר כאשר מוסיפים ומחסירים את אותו איבר:

The critical section problem

יש n תהליכים ואין הנחה מקדימה על הזמנים שלהם, רוצים למנוע את הסינכרון. נראה מימושים של אלגוריתמים להתחזות עם הבעיה. קיימים חמישה תנאים שברגע שהם מתקייםים באלגוריתם אין בעיה בחלוקת הקרייטי:

1. **Mutual exclusion** – רק תהליך אחד יהיה בחלוקת הקרייטי בזמן נתון.
2. **Progress** – אם יש הרבה תהליכים שוחצים להיכנס בחלוקת הקרייטי, אחד מהם ייכנס.
3. **Starvation free** – אם תהליך מסוים מחכה להיכנס, בסופו של דבר הוא ייכנס.
4. **Generality** – האלגוריתם יעבד לכל n תהליכים/מעבדים/threads.
5. **No blocking in the remainder** – תהליך שלא נמצא בחלוקת הקרייטי לא יחסום תהליך אחר.

אלגוריתם פטרסון



want הוא משתנה בוליאני שקיים לכל thread רוצה להיכנס לקטע הקרייטי או לא. Turn הוא תור מי להיכנס. הרעיון הוא שכל thread נוטן לאחר להיכנס לפניו. כל התנאים שהזכרנו מוקדם מתקיימים מלבד בלילהות.

פעולות אוטומיות

- פעולות שמתבצעת בפקודה אחת, באמצעות הביצוע לא יהיה context switch.
 - דרוש תמיינה מהמעבד.
- אם נחזור לדוגמה של ה-list linked, נרצהשה insert ו-ho remove insert ו-ho remove יהי אוטומיות וכן הרשימה תישאר תקינה.

TSL

הfonקציה TestAndSet משמשת היא אוטומית, ונitin לבנות בעזרתה אלגוריתם למניעה הדדי. הfonקציה מקבלת פרמטר lockPtr, משנה את הערך שלו, ומחזירה את הערך הקודם.

```
#define LOCKED 1
int TestAndSet (int* lockPtr) {
int oldValue= *lockPtr;
*lockPtr = LOCKED;
return oldValue;
```

אפשרות למשה כדי לבדוק מנעה הדדיות: כל thread ינסה לבצע TestAndSet ל-lock שמאוחתל ל-0. ובכל עוד הfonקציה תחזיר 1, ה-thread יחכה בollowah. ברגע ש-thread כלשהו קיבל 0 הוא ייכנס לקטע הקרייטי. - יתכן שלא יתקיים starvation.

```

int lock=0; //shared variable
....
while (TestAndSet(&lock))
    \\busy wait
critical section
lock = 0

```

```

down(): // also called wait()
down(S) {
    while(S <= 0)
        // busy wait
    S--;
}
up(): // also called signal()
up(S) {
    S++;
}

```

דוגמה - semaphore

כמו counter ששמור במה threads רשות להיכנס לאזור קרייטי בלבדו. ברגע ש-thread מסוים רוצה להיכנס הוא קורא לפונקציה down שהיא אוטומטית. אם הערך של semaphore קטן מ-0 אז אין מקום לעוד thread להיכנס לקטע הקרייטי והוא ייכה עד שהערך יידל. ביציאה מהקטע הקרייטי thread יעלה את הערך של semaphore.

Classical problems of synchronization

Bounded buffer problem

יש איזשהו ב퍼 שיבול להחזיק מספר קבוע של רכיבים. יש כמה producer – חלקם שמוסיפים מידע לבפר וחלקם consumer שמשתמשים במידע שלו. רוצים לוודא ש-producer לא יוסיף לבפר מידע כשהוא מלא, וה-consumer לא יקח מידע כשהוא ריק. כדי לפתור את זה משתמשים ב-semaphore.

סוגי semaphore שישמשו לפתור הבעיה:

- Mutex:** סמפור שמאתחל ל-1. רק thread אחד מקבל את הערך 1 ויכול להיכנס. יגנ על הבפר – רק אחד ייכנס אליו.
- fillCount:** סמפור שמאתחל ל-0 ושומר כמה איברים יש בבר sha-consumer יכולם לצרוך אותם.
- emptyCount:** סמפור שמאתחל ל-n (גודל הבפר) ושומר כמה מקומות פנוי יש בו.

Producer:

```

while (true) {
    produce an item
    down (emptyCount);
    down (mutex);
    add the item to the
    buffer
    up (mutex);
    up (fillCount);
}

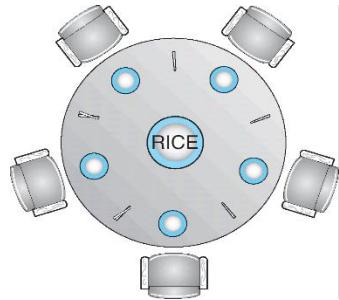
```

Consumer:

```

while (true) {
    down (fillCount);
    down (mutex);
    remove an item from
    buffer
    up (mutex);
    up (emptyCount);
    consume the item
}

```



Dining philosophers problem

פילוסופים יושבים סביב שולחן ורוצים לאכול. כדי לאכול כל פילוסוף צריך שני צ'ופסטיקים שנמצאים אחד משמאלו ואחד מימין. הבעיה היא שיש רק חמישה צ'ופסטיקים, ורוצים שבסופה של דבר בולם יאכלו.

- מנסים להימנע מ-starvation ו-*deadlock*.
- המידע המשותף – קורת אוזן (data set) וסמכורים (צ'ופסטיקים)
- מה חשוב הוא שהפתרון לא יהיה סימטרי.

פתרון 1:

הבעיה: deadlock בין שני ה-*chowdow*

```
While (true) {
    down ( chopstick[i] );
    down ( chopstick[ (i + 1) % 5] );
    eat
    up ( chopstick[i] );
    up ( chopstick[ (i + 1) % 5] );
    think
}
```

פתרונות:

- יהיה "מלצר" – מוטקס, כל אחד שרוצה להשתמש בצ'ופסטיקים צריך לבקש רשות.
- א-סימטריה: הפilosופים הזוגיים יעשו משהו אחר מהilosופים במספרים האזוגים.

אלגוריתם רנדומלי – lehmann-rabin :

כל פילוסוף מטיל מטבע, אם המטבע הוא 0 הוא רוצה לחת את הצ'ופסטיק מצד שמאל ואם 1 מצד ימין. מכך עד שהצ'ופסטיק שרוצה לחת יהיה פנוי. אוח"ב הוא בודק אם הצ'ופסטיק השני פנוי. אם כן תופס אותו. אם לא, מיותר על הראשון ומחיל הכל מחדש. אם קיבל את שני הצ'ופסטיקים הוא יצא מהלולאה ומשחרר.

```
repeat
    if coinflip() == 0 then          // randomly decide on a first chopstick
        first = left
    else
        first = right
    end if
    wait until chopstick[first] == false
    chopstick[first] = true          // wait until it is available
    if chopstick[~first] == false then
        chopstick[~first] = true      // if second chopstick is available
        break                         // take it
    else
        chopstick[first] = false     // otherwise drop first chopstick
    end if
end repeat
eat
```

Readers-writers problem

- יש מבנה נתונים משותף, ויש שתי קבוצות של threads:
1. **Readers** – קוראים מהדатаה ולא יכולים לשנות אותה.
 2. **Writers** – קוראים מהדатаה ומשנים אותה.
- יתכנו כמה threads שקוראים מהדטה באותו זמן.
- רק writer אחד יכול לשנות את הקובץ. לא ניתן לקרוא בזמן שהוא כותב בדטה.
 -

פתרון 1

שלושה סטפורים:

1. **readCount** – מאותחל ל-0 וסופר את ה-reads.
2. **readCount_mutex** – מאותחל ל-1 ומגן על ה-readCount.
3. **Write** – מאותחל ל-1 ומבודד שה-writer לא משתמש באותו זמן בדטה עם שאר ה-readers.

writer

```
while (true) {
    down (write);
    writing is
    performed
    up (write);
}
```

reader

```
while (true) {
    down (readCount_mutex);
    readCount++;
    if (readCount == 1)
        down (write); // lock from writers
    up (readCount_mutex)
    reading is performed // CS
    down (readCount_mutex);
    readCount--;
    if (readCount == 0)
        up (write);
    up (readCount_mutex);
}
```

הבעיה: לא מתקדים ל-writer starvation

פתרון 2: writer priority

- נוסיף את הסטפורים הבאים כך שתהייה ל-writer עדיפות על ה-readers:
1. **Read** – מוטקס שמאפשר לבקש לקרוא מהקובץ.
 2. **writeCount** – סופר את ה-writers שמחכים לקרוא מהקובץ.
 3. **writeCount_mutex** – מגן על ה-writeCount.
 4. **Queue**

writer

```
while (true) {  
    down (writeCount_mutex)  
    writeCount++; //counts number of waiting writers  
    if (writeCount ==1)  
        down (read)  
    up(writeCount_mutex)  
  
    down (write);  
    // writing is performed – one writer at a time  
    up (write);  
  
    down (writeCount_mutex)  
    writeCount--;  
    if (writeCount ==0)  
        up (read)  
    up (writeCount_mutex)  
}
```

reader

```
while (true) {  
    down (queue)  
    down (read)  
    down (readCount_mutex);  
    readCount ++;  
    if (readCount == 1)  
        down (write);  
    up (readCount_mutex)  
    up (read)  
    up (queue)  
    reading is performed  
    down (readCount_mutex);  
    readCount - -;  
    if (readCount == 0)  
        up (write);  
    up (readCount_mutex)  
}
```

תרגול 5 – Deadlock and caching

Deadlock

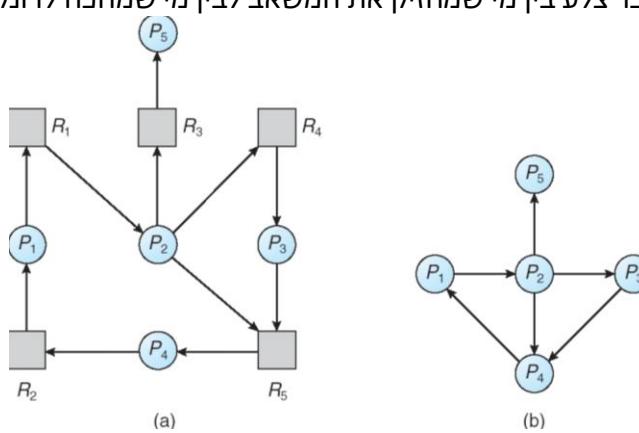
Deadlock הוא מצב שבו כל התהיליכים מחכים ובעצם נתקעים.

תנאים ל-deadlock

1. **mutual exclusion** – קיים משאב ייחודי שרק תהליך אחד יוכל להשתמש בו כל פעם ושאר התהיליכים יוכלו.
 2. **Hold and wait** – מצב שבו תהליך המחזיק משאב (סمفורה/מוטוקס) יכול לחכות למשאב נוסף ולהיות חוסם.
 3. **No preemption** – התהליך שמחזיק במשאב זה שמשחרר אותו לפי רצונו.
 4. **Circular wait** – מעגל.
- ניתן ליצור גרף המקשר בין התהיליכים ומשאיים, ואם אין בו מעגל אז אין deadlock.

פתרונות לטיפול ב-deadlocks

1. **התעלמות** – ostrich approach
 - אם deadlocks קוררים לעיתים רוחקות.
 - מניעת deadlocks יכולה להיות יקרה לבוא על חשבון דברים אחרים.
 - windows ו-linux משתמשים בכך. במקרה זה עושים reset למחשב.
 2. **Detection and recovery**
 - יש איזשהו גרף של בקשות (העיגולים הם התהיליכים והרביעועים הם המשאיים ונוצר צורך לבדוק אם יש בו מעגל. אם כן, נדרש לפתח אותו.
 - מחיר זמן ריצה גבוה.
 - recovery :
- סיום כל התהיליכים.
 - סיום תהליך אחד. עליה השאלה איזה תהליך נרצה לסיים?
- בגרף לדוגמה מ-**a** העיגולים הם התהיליכים והרביעועים הם המשאיים. כדי לזהות מעגל נוריד את המשאיים ונחבר צלע בין מי שמחזיק את המשאב לבין מי שמחכה לו ונקבל את גרף **b**.



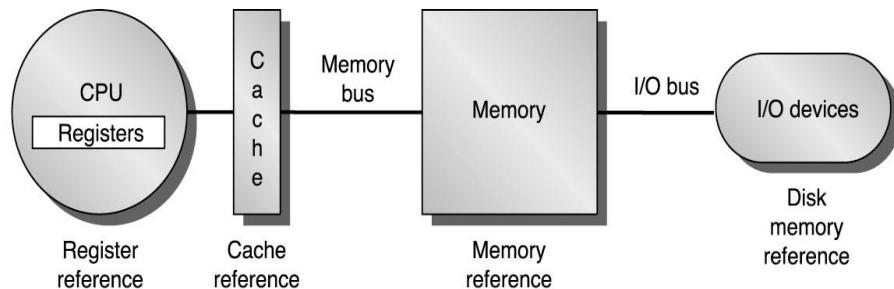
3. **Avoidance** – ננסח להתחמק מ-deadlock.
 - בכל תהליך מצהיר מראש כמה משאיים יצטרכו וכל משאב מוקצה מראש.
 - מערכת הפעלה מנהלת אлогוריתם בשם אלגוריתם הבנקאי ובבודק מראש אם יש אפשרות ל-deadlock.
4. **מניעה** – לא אפשר שהתנאים ל-deadlock יתקיימו.

מפנייה

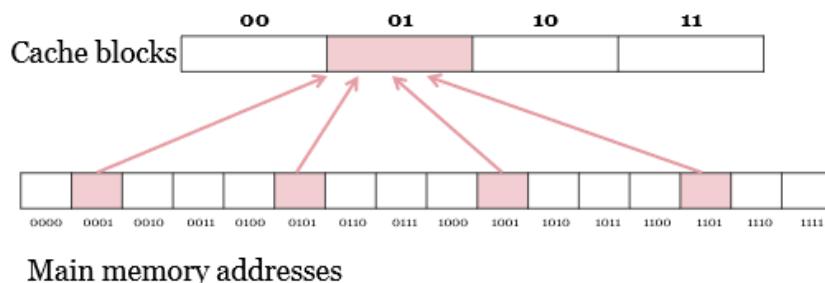
1. **Mutual exclusion** – יש מקרים שבהם אין צורך ב-mutual exclusion ואז זה בסדר, אבל יש מקרים שבהם אין מצב לבטול ונוצרך לנסוט למנוע תנאים אחרים.
 2. **No preemption** – כאשר מערכת הפעלה מכינה תהליך-block היא משחררת את כל המשאים שהוא נעל. הפתרון בעיתי כי יש מקרים שבהם אי אפשר להפקי שליתה למשאב מסוים, ולפעמים המשאים לא יודעים מראש.
 3. **Hold and wait** – נודא שכאשר תהליך מבקש גישה לאיושה משאב, הוא לא מחזיק במשאב אחר. בעצם כל תהליך מקבל את כל המשאים בביטחון אחד לפני שהוא מתחילה לרטז.
- הבעיות:
- לא תמיד התהיליך יודע את המשאים שייצטרך מראש.
 - יכולה להיות הרעבה. יתכן שהטהיליך לא יוכל ל תפוס את כל המשאים.
 - ? Livelock
- חותמת זמן: עדיפות לתהליכי המבוגרים יותר מל"כעירים" בתפישת משאים.
4. **Circular wait** – הימנעות ממגל. כל משאב מקבל מספר וכל התהליכי נעלים אותו לפי הסדר. לדוגמה בעיית הפילוסופים הסועדים.

Caching

Cache – זיכרון קטן שנמצא בין ה-CPU ל-RAM וכן נועד כדי להגביר את זמן הביצועים. מבוסס על ההנחה שכשניגשים למקום בזיכרון ניתן למצוא מקום לאזרור.

Direct mapping

מייפוי בין ה-cache לבין הזיכרון. כאשר ה-CPU מבקש גישה למקום בזיכרון, איך נדע אם המקום נמצא ב-cache? cache - ב-cache יש בלוקים שמכלים מיליה וכל כתובות בזיכרון מומפה לכתובת ב-cache.



ניתן ב-cache שהבלוק יכול יותר ממיליה אחת, והמקום בבלוק תלוי בבייטים-least significant.

דוגמה:

| | |
|-------------------------------------|---|
| 4MB Cache | - |
| 4GB RAM | - |
| מילה בגודל 4 בתים | - |
| בלוק בגודל 8 מילימ' | - |
| $\frac{2^{22}}{4} = 2^{20}$ - cache | |
| $\frac{2^{20}}{8} = 2^{17}$ - cache | |

אינדקס $\leftarrow \log_2(\#blocks\ in\ cache) = 17\ bits$

אופסט $\leftarrow \log_2(block\ size) = 3\ bits$

תג $\leftarrow \log_2(\#words\ in\ RAM) - index - offset = 30-17-3 = 10$

מייפוי המילה בכתובת 5221 -

000000000000 0000001010001100 101

בשה-CPU ירצה לגשת לכתובת 5221 בזיכרון הוא יבדוק קודם בבלוק 652 ב-cache את התג. אם הוא זהה, ה-CPU לא ייגש ל-RAM. אם לא, הוא ניגש ל-RAM ודורס את מה שיש בבלוק עם מה שכתבו ב-5221.

2^x -way associativity

מייפוי חד ערכי. מקום בזיכרון יכול להיות ממופה לכמה מקומות(cache).
ה-cache מחולק לסטים כך שבכל סט יש 2^x בלוקים והזיכרון יכול להיות ממופה לבlok בכל הסטים.

דוגמה:

| | |
|---------------------|---|
| 4MB Cache | - |
| 4GB RAM | - |
| מילה בגודל 4 בתים | - |
| בלוק בגודל 8 מילימ' | - |

نمפה לפי 16-way associativity:

$$\frac{2^{22}}{4} = 2^{20} - cache$$

$$\frac{2^{20}}{8} = 2^{17} - cache$$

$$\frac{2^{17}}{16} = 2^{13} - cache$$

אינדקס $\leftarrow \log_2(\#sets\ in\ cache) = 13\ bits$

אופסט $\leftarrow \log_2(block\ size) = 3\ bits$

תג $\leftarrow \log_2(\#words\ in\ RAM) - index - offset = 30-13-3 = 14$

מייפוי המילה בכתובת 5221 -

00000000000000 0001010001100 101

בשה-CPU ירצה לגשת לכתובת 5221 בזיכרון הוא יבדוק קודם ב-16 הבלוקים שנמצאים בסט 652 את התג ומשם ישמש במילה החמישית.

שאלה לדוגמה – 4way associativity

נשתמש בהנחה הקיימת:

A **word** is 4 bytes.
Main memory is of size 2^{20} words.
Cache is of size 2^8 words.
Each **block** is of size 2^3 words.
Cache is **4-way** set associative.

1. מה הכתובות האפשריות ב-cache אליהן הכתובת 209715 יכולה להיות ממופה?

00110011001100 110 011

בגלל גודל הבלוק, האפסט יהיה 3 ביטים, מספר הסט יהיה גם 3 ביטים וכל השאר יהיה התג.

הכתובת ב-cache תהיה 110XX001, כלומר כל הכתובות האפשריות הן:

195 (**11000011**)203 (**11001011**)211 (**11010011**)219 (**11011011**)

2. נניח שיש cache hit לכתובת 209715 בכתובת 195 (ב-cache). מה יהיה כתוב בכתובת ?194?

הערך השמור בכתובת 209714 (0011001100110010), כי המיפוי שלה יהיה לכתובת

110XX010 ומכיוון שהוא לנו היט בכתובת 195, כל הבלוק שמכיל את 195 – 11000 – 11000 – בעצם קיבל היט.

לכן המיפוי יהיה ל-11000010 (194).

Replacement algorithm

Victim – הבלוק ב-cache שנבחר כדי לדרס אותו ולשים מידע אחר מהזיכרון.

– היחס בין כמות ה-misses לבין כמות הפעמים שכיגשנו. **Fault rate**

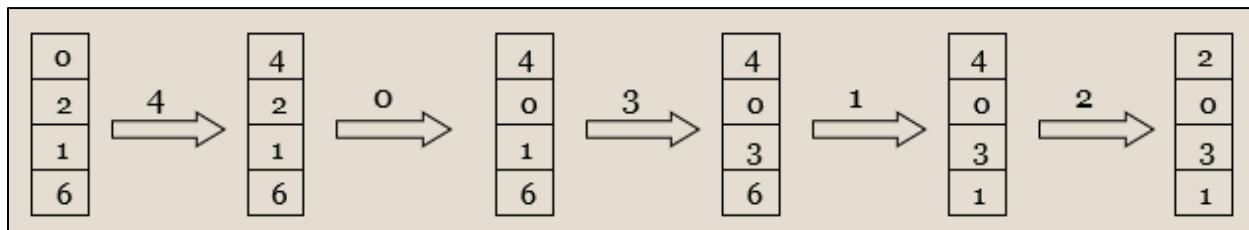
אין נדע לבחור אותו?

- בקורס אקרואית, הבעיה – أولי נפנה בлок שנצחך.
 - אלגוריתמים להחלפה:

FIFO – First in first out

הבלוק הבי ותיק ב-cache יפנה את מקומו לטובת בלוק חדש. קל למיושן.

דוגמא: נבנис לפי הסדר הבא

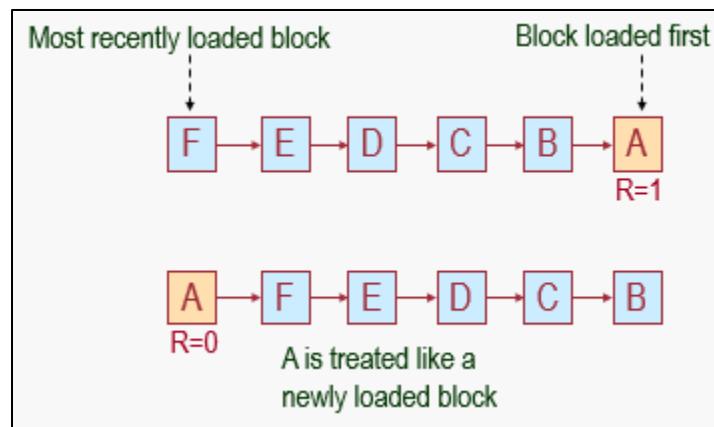


השיטה לא כל כך יעילה. אם למשל נרצה להכנס בлок אחריו שכבר הודנו אותו זה יהיה בΖבוז. יכול להיות גם שגעיף בлок שאחננו משתמשים בו די הרבה.

Second chance FIFO

כל בלוק יש רפנס בית, באשר ניגשים אליו מדליקים את הבית.

ההחלפה נעשית כמו FIFO רק שאם ניגשנו לבлок שהביט שלו הוא 1, נכבה את הביט ונעביר אותו לסוף קר שהוא עכשווי הבלוק הקי חדש.



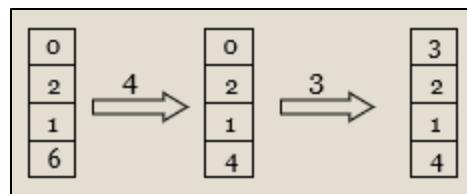
Optimal block replacement

נרצה להעיף בлок שאנו יודעים שלא נשתמש בו יותר.

השיטה בלתי ניתנת למימוש אבל יכולה לשמש להשוואה עם שיטות אחרות.

לעומת

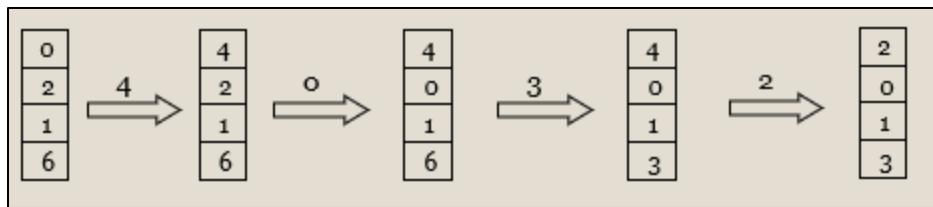
כמו מוקדם נכנס לפיקוד הסדר 1, 2, 3, 0, 1, 0, 4, 6, 2, 0.



LRU – Least recently used

נוריד את הבלוק שהשתמשנו בו עד כה הכי פחות מבין שאר הבלוקים ב-cache.

דוגמה:



שיטת אבל בעיינית למימוש, דרושה תמייה של החומרה (עם מונימ או איזושאי מחסנית) ובסביל זה יש **LRU-pseudo**.

ניהול זיכרון

ה-CPU יודע לעבד רק עם כתובות שנמצאות בזיכרון הראשי. יש ³³ 2^{32} כתובות בזיכרון, כל כתובה 33 ביטים.

Virtual address space

בשלהיל רץ במחשב מבחינתו כל הזיכרון שייך לו. מרחב הכתובות הווירטואלי זה מרחב כל הכתובות שתהילן חושב ששיכוכות לו.

במערכת הפעלה של 32 ביט כל תהילך חושב שכמות הזיכרון שיש לו היא ³² 2, למרות שבפועל זה לא נכון, ויש רכיב חומרה שיודע לסנן בין כל הזיכרונות הווירטואליים.

MM

הרכיב במעבד שיודע למפות כתובות לוגית (וירטואלית) לכתובות אמיתי. יש שתי גישות למיפוי:

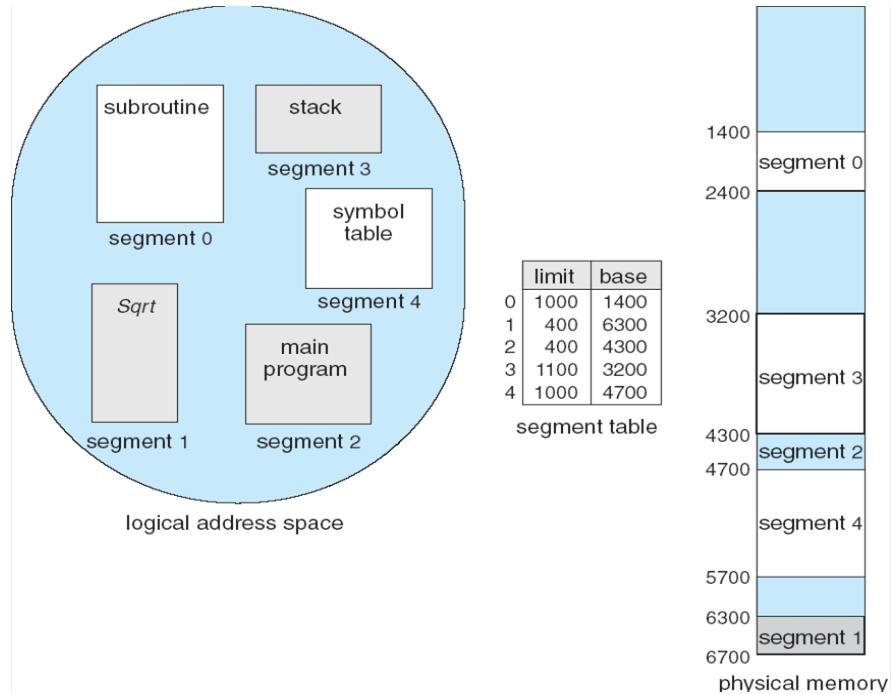
1. **Segmentation** – חלוקת הזיכרון של תהילך מסוים לכמה יחידות קטנות שהן אוסף של כתובות בזיכרון.

– segment (פונקציה, אובייקט, סטאק, מערכיים וכו'). ה-MM יודע למפות את הסגמנטים למקום בזיכרון האמיתי.

הגישה לסגמנט מסוים נעשית ע"י חלוקת הכתובת הלוגית לשני חלקים: החלק הראשון הוא מספר הסגמנט אליו רוצים לגשת, החלק השני הוא offset.

מערכת הפעלה מנהלת טבלת סגמנטים. כל שורה בטבלה מייצגת מספר סגמנט ומכליה שתי עמודות:

- Base
- limit



.1 – נדבר עליה בהמשך. **Paging**

תרגול 8 – Paging

במה מספרים וחישובים בסיסיים:

- Byte = 8 bits
- KB = 2^{10}
- MB = 2^{20}
- GB = 2^{30}

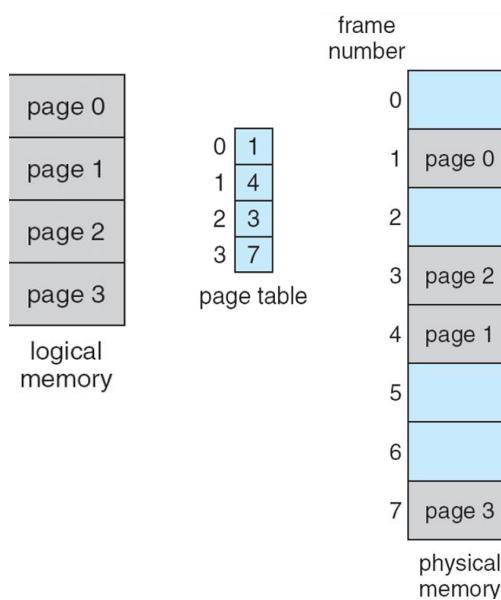
$$\begin{aligned}
 & 4\text{GB} \text{ (memory size) / } 8\text{KB} \text{ (block size)} = 2^{32} / 2^{13} = 2^{19} \\
 & \quad \text{(number of blocks)} \\
 & \quad \text{ניתן ליצא } 2^8 \text{ מספרים עם 8 ביטים} \\
 & \quad \text{חישוב הערך הדצימלי של 1101:} \\
 & \quad 1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3 = 13
 \end{aligned}$$

מרחב הכתובות הווירטואלי והפיזי

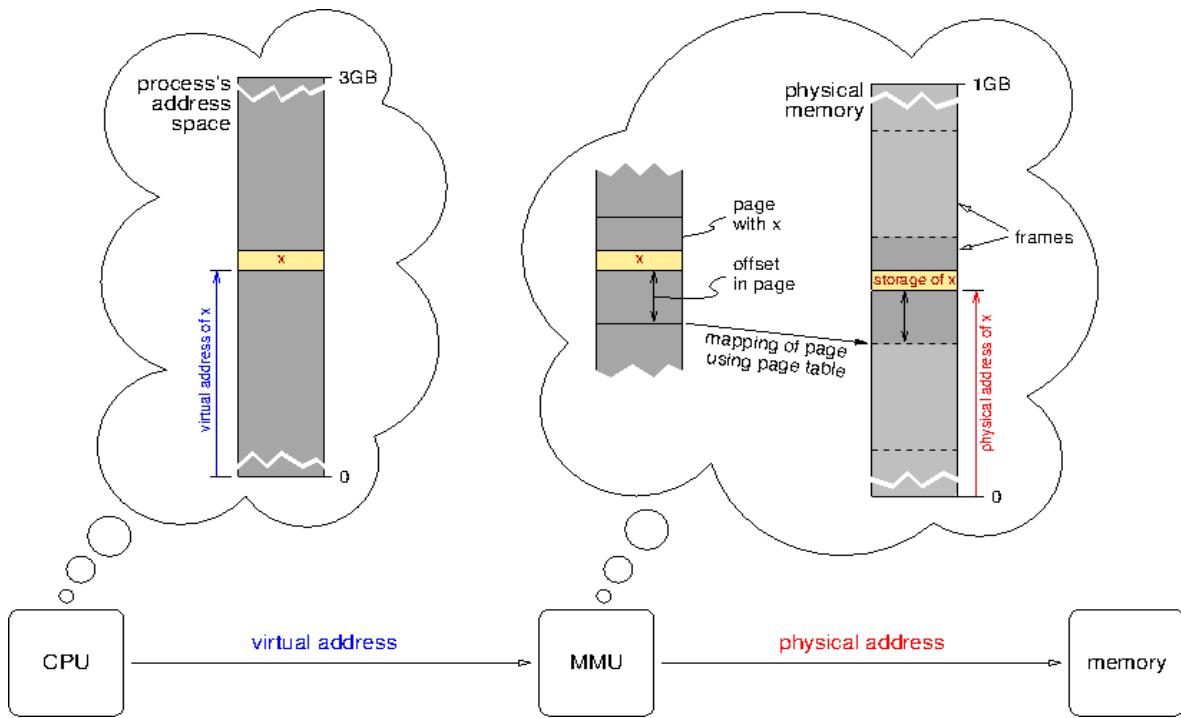
הזיכרון הוא דינامي ולכן תהליכי לא ידעו מראש היכן הדטה שלהם תשמר בזיכרון הפיזי ולכן יש את הזיכרון הולוגי שעובר מיפוי ע"י מערכת הפעלה לזיכרון הפיזי. איך נעשה המיפוי?

- **סגמנטציה** – כל יחידת זיכרון בתהיליך (stack, heap) וכו' נקראת סגמנט, וכל סגמנט מקבל הקצהה רציפה בזיכרון הפיזי. כדי לבצע את המעבר מכתובת וירטואלית לכתובת פיזית יש טבלה שmapsדרה את ה-base (המקום בו הסגמנט שמור בזיכרון הפיזי) ואת ה-limit (offset) של כל סגמנט. צריך לבדוק שה-base, offset המיקום אליו רוצים להגיע בסגמנט, לא חורג מהגבול.
- בעיה: **external fragmentation**, זיכרון פנוי ולא בשימוש בין סגמנטים של תהליכי.
- **Paging** – התמודדות עם בעיית הפרוגמאנטציה החיצונית. חלוקת מרחב הכתובות הווירטואלי לחלקים שוויים – pages (גודל 8 KB). הדף ממופה למסגרת בזיכרון הפיזי ויש טבלה שעוקבת אחר המיפוי של כל דף.

יכול להיות מצב של **internal fragmentation**, בזבוז מקום פנימי ב-pages, אבל היא פחות גורעה מהפרוגמאנטציה החיצונית.



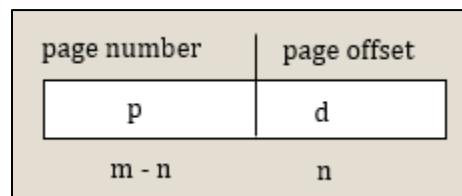
תרשים מיפוי ע"י-MMU:



תרגום כתובות

התרגום נעשה ע"י חילוק הכתובת לשני חלקים:

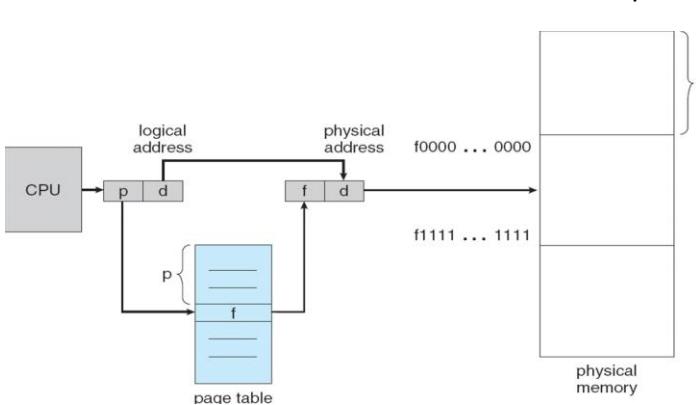
1. מספר ה-page (האינדקס בטבלה בו תישמר הכתובת הבסיסית של הדף בזיכרון הפיזי)
 2. ה-offset (יחד עם כתובות הבסיס יציין את המיקום בזיכרון הפיזי)
- עבור זיכרון לוגי בגודל 2^m ודף בגודל 2^n החלוקה תתבצע באופן הבא:



בנוסף יש שני רגיסטרים במעבד עבורם ה-CPU שומר עבור התהילה הנוכחי:

- (PTBR) **Page-table base register** פוינטער לטבלה.

• **Page-table length register** גודל הטבלה.



מידע נוסף שנשמר ב-table page

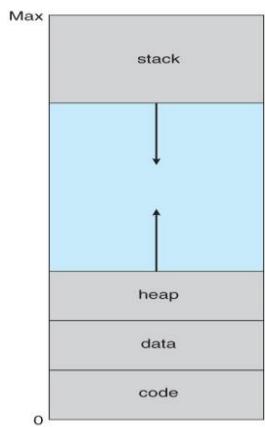
- – ביט שמצוין האם אותו דף נמצא בזיכרון הפיזי (0 אם לא).
- – ביט שמצוין האם הדף השתנה.
- – מתי הייתה הגישה האחרונה לדף.
- – האם הדף הוא read-only, read-write וכו'.

 פיקו page

במקרה שבו מנסים למפות דף למסגרת שלא פנוי קורה page fault. זה בעצם אומר שהדף לא ממופה וה-bit valid יהיה שווה ל-0. מערכת הפעלה תחפש מסגרת פנוי, ואם אין אחת בזו מערכת הפעלה תוציא page קיים ותשימם במקום אחד חדש.

מבנה טבלת הדפים

גם הטבלה ממופה לזכרון ותופסת בו מקום משמעותי. קיימות שתי שיטות שמקטינות את הזיכרון שהטבלה תופסת:



- – נזכר על השיטה בתרגול הבא.
 - – **Inverted page table** – ה-stack ו-heap בזיכרון הלוגי מוקצים אחד לקראת השני כמו בשרטוט:
- לפעמים האזר שביביהם לא בשימוש ואין סיבה להקצתו לו מקום, אין סיבה להחזיק טבלה שבה חלק מהדפים לא יהיו בשימוש.
- נפרק את הטבלה לכמה רמות, כל שורה ברמה מסוימת מצביעה לרמה שמعلיה.

תרגול 9 – 9 memory management and file systems

Paging

Inverted page table

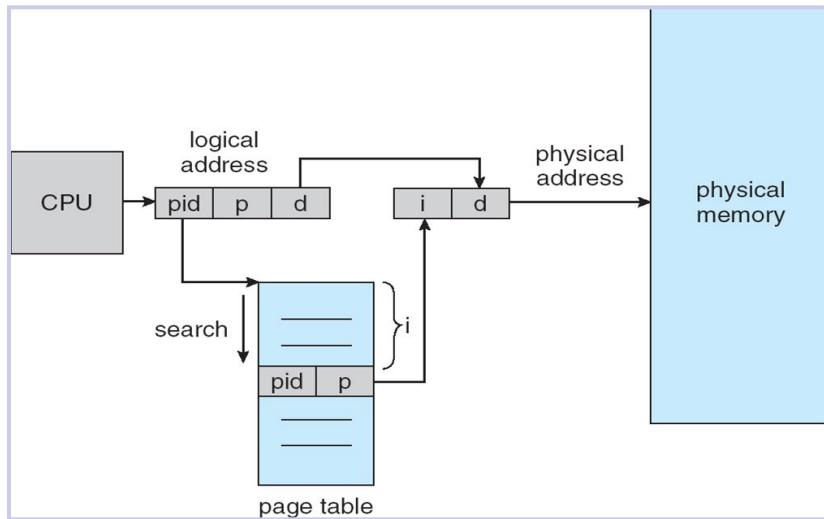
בכל שטוף התהילכים גדול, בך גם ה-pages ומרחב הכתובות הוירוטואליות וכן גם טבלת הדפים שתופסת הרבה מקום בזיכרון עם שורות ריקות ולא ממוקמות. אחד הפתרונות לכך (מעבר לטבלה היררכית) הוא טבלה הפוכה.

אם בגישה הקודמת לבל תהיליך הייתה טבלת דפים משלו וכן תהיליך אחד לא ניתן לזכור של תהיליך אחר, אז יש טבלה אחרת המשותפת לכל התהילכים וממנה מסגרת לדף.

- מספר המסגרות בזיכרון הוא מספר השורות בטבלה. אין שורות מיוחסות בטבלה במקרה שמנצחים את כל ה-RAM. הטבלה לא גדלה, גם בשמגדלים את התהילכים.

בשנרצה לגשת לאייזה דף, נחלק את הכתובת שלו ל- **pid + p + pid**. כל שורה בטבלה מכילה את **p** (מספר ה-page) ואת **pid** (הזהה של התהיליך, מכיוון שהטבלה שייכת לכל התהילכים).

לא נדע איפה הדף בטבלה הוא נמצא, ונctrar לעבור על כל השורות ולחפש את השורה בה כתוב **p**. אם באותו **p** כתוב ה-**pid** של התהיליך, ניתן לזכור הפיזי לפי השורה שמצאנו את **p** בטבלה (**i**), ונראה שהאפסט **d** נמצא.



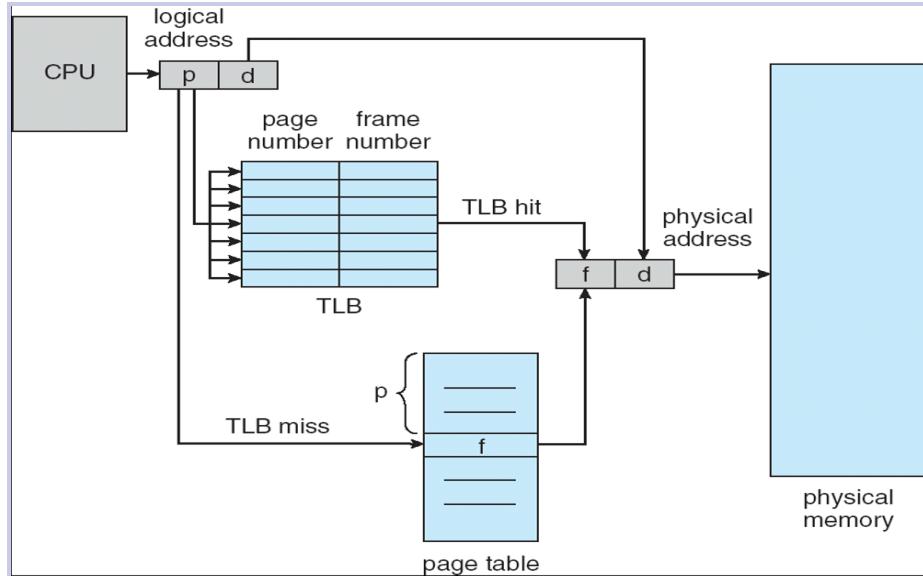
TLB

נזכר שה-cache משמש כדי לחסוך גישות ל-RAM. דיברנו על המיפוי בין ה-cache ל-RAM. בשממפים דפים באמצעות הטבלה יש כמה גישות לזכרון זה מגדיל את התקורת. למשל בטבלה שלבים ויהיו יש שתי גישות – גישה לטבלה עצמה וגישה לדאטנה. בד"כ בטלאות הדפים הן בכמה שלבים ויהיו אפילו יותר מושתת.

בשביל זה קיימים cache לטבלת המיפוי בשם **translation look-aside buffer** (TLB – בד"כ fully associative) עם 64 שורות ולמן לא יהיה index אלא רק tag ואפסט) שם יישמרו המיפויים האחרונים שהשתמשנו בהם לאחרונה/לעתים קרובות. אם יש מיפוי של page למסגרת מדגים על הגישות לטלאות הדפים.

גם ה-cache הזה משותף לכל התהילכים וכך לאפשר הגנה נחזק בכל שורה ערך **address-space identifiers (ASIDs)** (שייחודי לכל תהיליך).

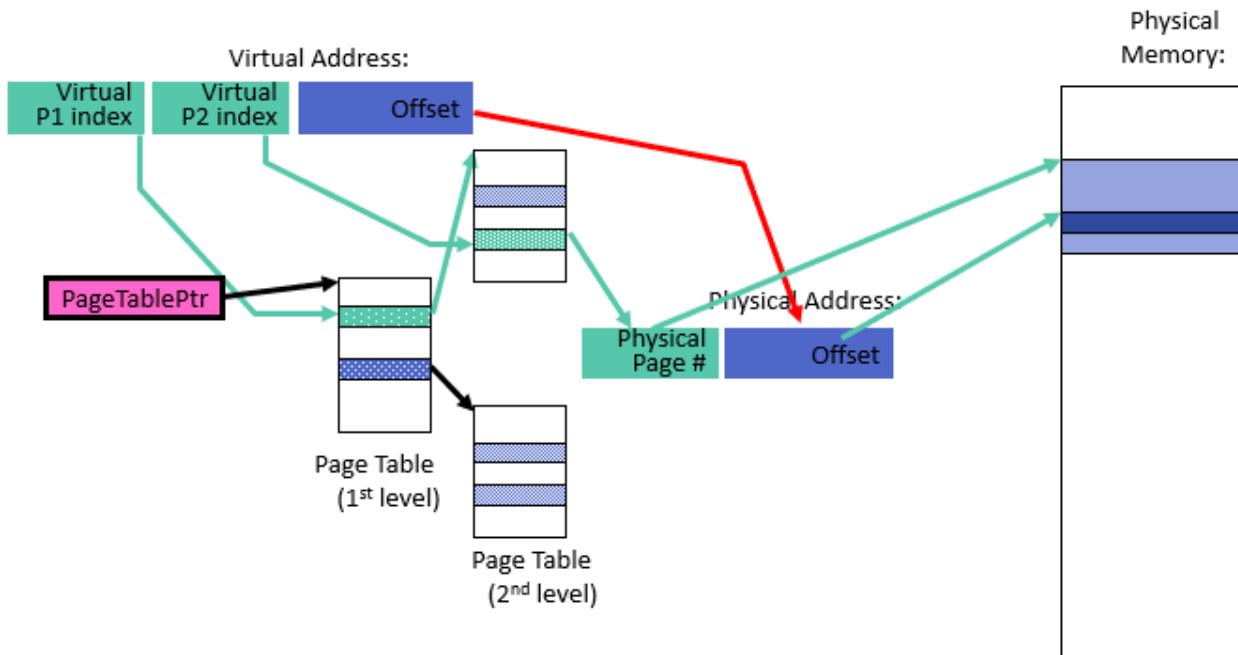
כאשר יש את הכתובת הפיזית (בין אם נמצאה ב-TLB או לא) ניתן לחשב את המיפוי ב-cache ולנסות לחסוך את הגישה ל-RAM.



נסתכל עבשו על שילוב של כל מה שלמדנו –
במיינט בתובת וירטואלית של page נחלק אותה באופן הבא:



1^ק מציבע לנו על מספר השורה בטבלת הדפים ברמה הרכינה (first level) ← שם יש את הטבלה הדרישה בرمה שנייה ← שם ניגשים לשורה 2^ק ← שם יש את הכתובת בזיכרון הפיזי, מספר המסגרת ← באותה המסגרת נסתכל על ה-offset שם יהיה מה שנוצר.



בנוספּ ניגש ל-TLB לשורה k ואם זה שם נקלט מייפוי של pid למסגרת, וכן חסכנו שני גישות ל-RAM. בעת בשנרצה לגשת ל-RAM נבדוק קודם cache לפי המייפוי. אם לא, אז נשמר את המייפוי בעת ב-TLB וב-cache.
(המחשות בשקופיות 11 ו-12)

File systems

קובץ

- **ADT** – יחידה לוגית, אבסטרקטית של נתונים.
- בעל שם וデータ (תוכן).
- בנוספּ בעל **Meta-data**, אטראיבוטים (תהליך יצירה, גודל וכו').
- ניתן לעשות עליו פעולות (פתיחה, קרייה, כתיבה וכו')
- **non-volatile**, לא נמחק שמאכבים את המחשב, בשתוכנית מסתימת.

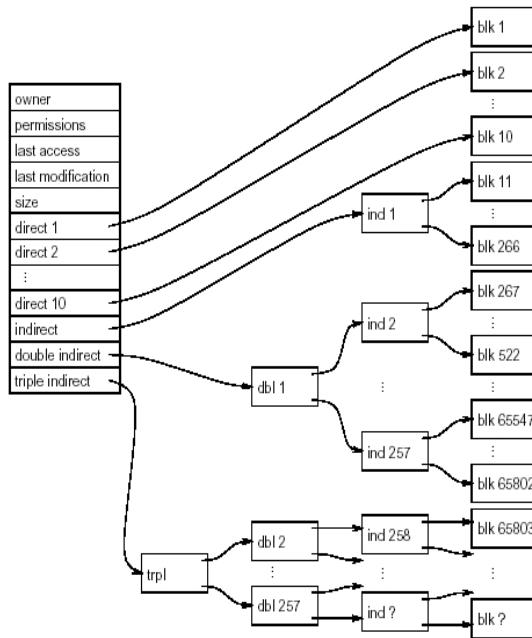
File meta-data

- **Size**
- **Owner**
- **Permissions**
 - Readable? Writable? Executable?
- **Timestamps**
 - Creation time
 - Last time content\metadata where modified\accessed
- **Location**
 - On disk (recall that a disk is a “block device”)
 - Where do the file’s blocks reside on disk?
- **Type**
 - E.g., binary vs. text, or regular file vs. directory

נשים לב שם הקובץ הוא לא חלק מהmeta-דטה כי בערךון לקבצים אין ממש שם, לכל מערכת הפעלה יש דרך מסוימת ליזג קובץ. השם הוא סוג של אבסטרקציה.

inodes

- ביונייקס קבצים מיוצגים ע"י **inodes** – איזשהו מבנה נתונים מוחולק לשדות.
- ה-inodes מכילים את המטה-דטה ובנוספּ מצביעים לבלוקים בדיסק שם שמור הדטה הרלוונטי:
- 12 פוינטורים ישירמים (לבולוקים אחד אחרי השני, לא חייבים להשתמש בכלום), מספיק לקבצים קטנים שדורשים רק 12 בלוקים.
 - אם יש צורך ביותר, יהיה פוינטר בשם **single indirect** שמצבייע לבלוק בדיסק שבו יש פוינטרים לבלוקים אחרים. מספיק לקבצים בינוניים.
 - אם יש צורך ביותר, יהיה פוינטר בשם **double indirect** שמצבייע לבלוק בו יש פוינטרים שמצבייעים לבלוקים בדיסק שמצבייעים לבלוקים עם הדטה. מתאים לקבצים גדולים.
 - לקבצים ממש ענקיים, יש **triple indirect**.

**דוגמיה:**

נתונה מערכת קבצים עם `nodes` עם גודל 32 ביטים, כל מצביע בגודל 4 בתים, כל בלוק בדיסק הוא בגודל 4096 בתים.

- 12 פוינטרים ישירים יאפשרו לנו קבצים בגודל 48 KB.
- Indirect מכיל בלוק עם 1024 פוינטרים לבlokים ואפשר לנו קובץ בגודל 4 MB.
- Double indirect מכיל בלוק עם 1024 פוינטרים שמצוינים על indirect 1024 פוינטרים,סה"כ 4GB.
- טריפל אפשר 4TB.

Directories

תיקיות הן בעצם קבצים וגם להן יש `inodes`. התיקיות בעצם מאחסנות את השמות של הקבצים (כל קובץ לא מחזיק את השם שלו).

הפקודה `ls` נותנת עבור תיקייה מסוימת את ה-`inodes` של הקבצים בתיקייה.
הפקודה `ls` – `ls` נותנת גם חלק מהמטה-דטה.

הקצאת בלוקים ו-inodes

Superblock – בלוק שמאוחתך באשר מפרמטרים את הדיסק. מכיל את גודל הדיסק, רשימה של בלוקים פנויים ושל `nodes` פנויים. גם ה-`nodes` נקבעים בפרמטרים.

- לא כל ה-`inodes` הפנויים מוחזקים בסופר-בלוק, ולכן מערכת הפעלה צריכה לחפש `inodes` פנויים.

באשר יוצרים קובץ מקדים לו `inodes`.

אחרי שייצרנו אותו, באשר נרצה לכתוב לתוכו נצטרך להקצתו לו בלוקים.

באשר אנחנו קוראים/כותבים משתמשים בבלוק שלם. אם רצים לשנות חלק ממנו צריך לקרוא את כלו, לשנות את החלק שאנו רוצים ואז לשנות את כל הבלוק לדאטה המעודכן.

לפעמים אפשר לעשות את הפעולות האלה ב-RAM.

פעולות על קבצים

פתחת קובץ

כאשר פותחים קובץ מקבלים איזשהו **fd** – **file descriptor**, מספר שמייצג את הקובץ מול מערכת הפעלה ובאשר נרצה לבצע על הקובץ פעולות נשלח את המספר זהה.

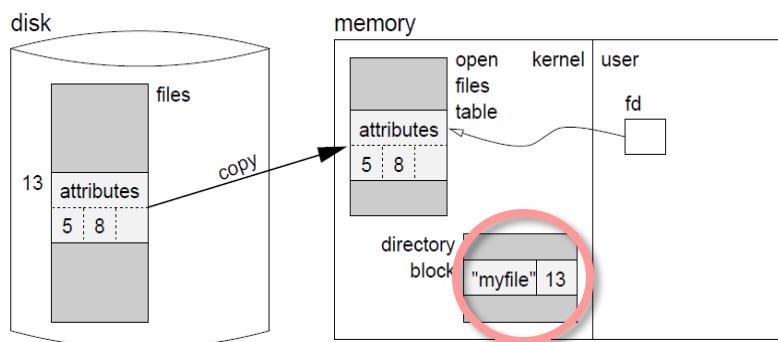
```
fd=open("myfile",R)
```

- ל-threads שלאות תהליך יש טבלה של fd.
 - לא ניתן לשנות את הצבעה של ה-fd, תמיד יתאר את אותו הקובץ.
- בלינוקס ל-hinput, stdin, stderr ו-stdout יש איזשהו fd, וביתן לעשות read direction שבאמת כל הפעולות האלה עושים על איזשהו קובץ. ה-fd שלהם קבוע מראש.

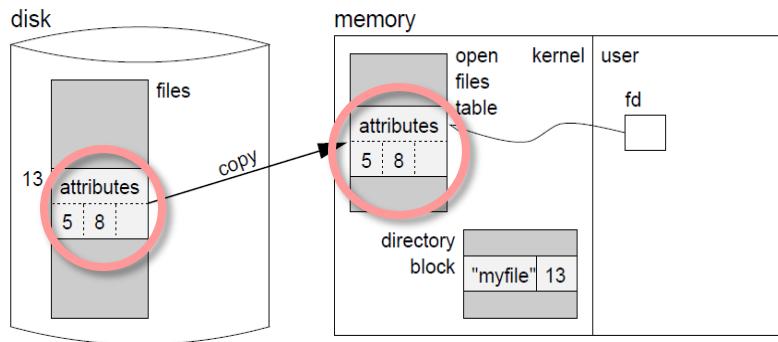
| File | File Descriptor | POSIX Symbolic Constant |
|-----------------|-----------------|-------------------------|
| Standard Input | 0 | STDIN_FILENO |
| Standard Output | 1 | STDOUT_FILENO |
| Standard Error | 2 | STDERR_FILENO |

לקבצים אחרים, בעת הפקודה **open** מקבלים fd שמצוין על טבלה בשם open files table. התהילך קורה בדרך:

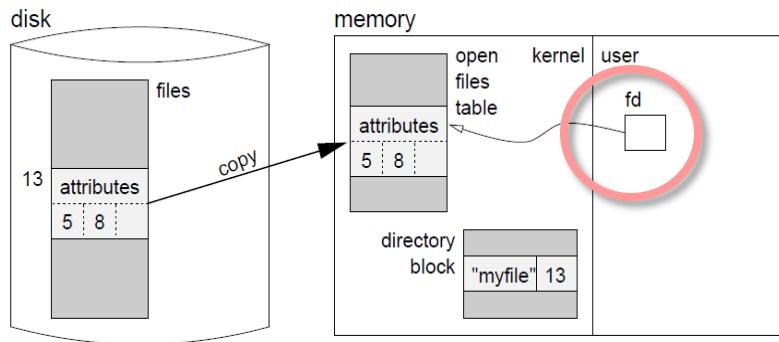
1. בעת הקריאה ל-**open**, ניגשים לתקינה הנוכחית ושם נמצא inode של הקובץ.



2. נחפש בדיסק את inode המצוין (13 בדוגמה) ונעתיק את תוכן שלו לטבלה open files, כדי לא לפגש כל פעם ליסק בשנctror לבצע פעולות.



3. ה-fd שיווזר יקבע על המיקום בטבלה בו שומר המטה-דאטא של הקובץ.



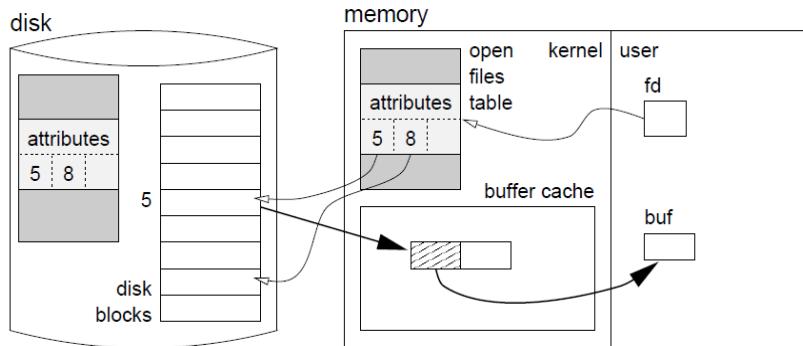
ה-open files table בעצם שומרת את ה-inodes ומאפשרת לבדוק הרשות ואחסן את הסמן של הקובץ.

- קובץ אחד יכול להופיע במא פעים בטבלה.

קריאה מקובץ

read (fd, buf, 100)

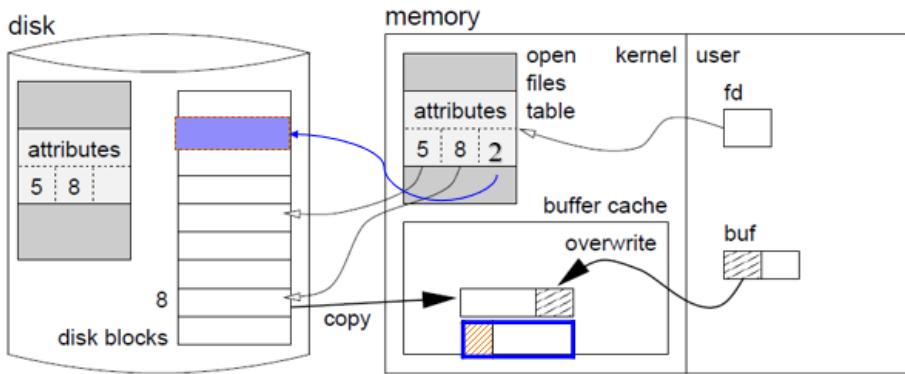
1. הפקודה read מקבלת את ה-fd, ה-buffer buf ומספר הבתים שרוצים לקרוא. דרך ה-fd מוצאים בטבלה את הבלוק הדיכרונו אליו נרצה לגשת.
2. ניגשים לאותו בלוק בזיכרון ומעתיקים אותו כל התוכן שלו ל-buffer cache.
3. התוכן הרצוי מועתק לתוך ה-buffer buf שנთנו.



כתיבה לקובץ

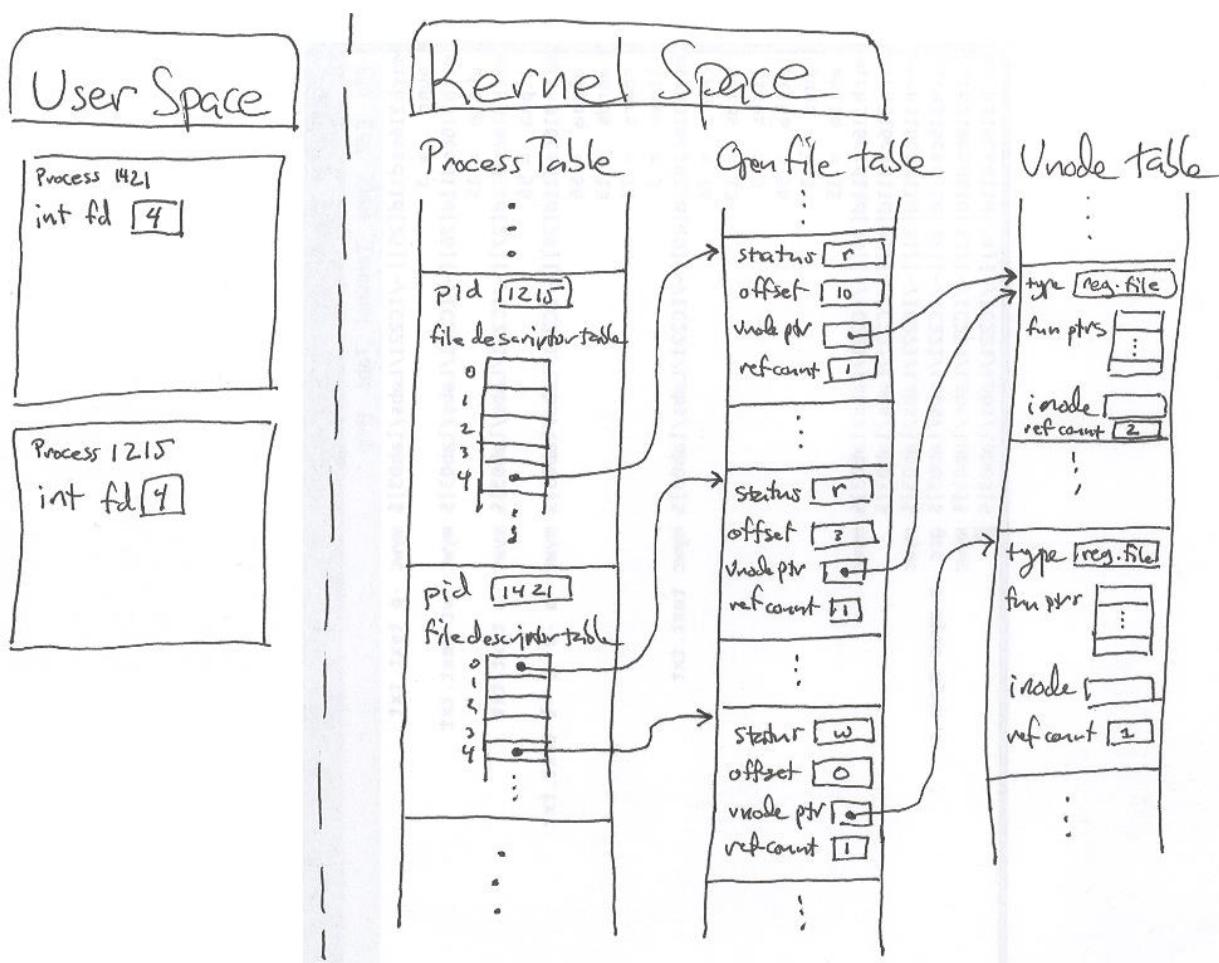
לקובץ מוקצים מספר בלוקים ובאשר הם מלאים ונשאר תוכן לבטיה נצטרך להקצות בלוק חדש כדי לבטוב לתוכו. האופטט, הסמן שלו, נמצא באיזשהו מקום לקובץ. נניח שהתוכן שאנחנו רוצים לכתוב גדול יותר מהה שנשאר לנו לכתוב בבלוק, למשל רצים לכתוב 100 בתים לקובץ עם בלוקים בגודל 1024 בתים, והסמן נמצא במקום ה-2000, נשארו לנו 48 בתים בבלוק מס' 2 שנרצה לכתוב ונצטרך להקצות בלוק שלישי.

1. נקצת בלוק חדש מחרוז פול של בלוקים פנויים.
2. נקצת בלוק חדש ב-buffer cache ונכתבו לתוכו את התוכן. נקצת בלוק חדש ב-buffer cache ונכתבו לתוכו את התוכן.
3. הבלוקים ששינו יועתקו אל הדיסק.



טבלאות הקבצים של מערכת הפעלה

1. **The inodes table** – כל הקבצים שבדיסק לפי ה-inodes שלהם.
2. **The open files table** – כל הקבצים שפתחנו. בכל שורה בטבלה יש פוינטר לטבלת-h-inodes ומיקום בקובץ (הסמן). ניתן לפתוח את אותו קובץ כמה פעמים ולבן בטבלה יכולות להיות שורות שונות שמחזיאות על אותו inode.
3. **The file descriptor table** – טבלה המוקנית לביליהר, כל שורה בה מצביעת לשורה ב-table.open files. מספר השורה הוא בעצם ה-fd של אותו קובץ.



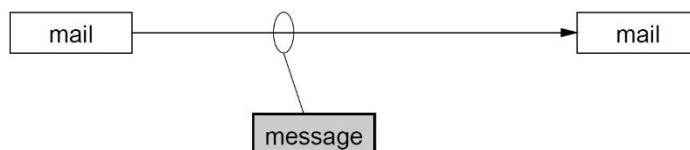
תרגול 10 – sockets

Networking protocols – motivation

פרוטוקול תקשורת

- פרוטוקול זה מוסכמתה בין שני צדדים לכלי דיבור, כללים על האופן בו שלוחים הודעות, כדי להיות מתאימים לגבי השיחה – סינטакс, סמנטיקה וסינכרוניזציה.
- פרוטוקול תקשורת בין מחשבים זה כללים דיגיטליים לגבי חילופת נתונים בין המחשבים.
- הכללים חייבים להיות מוגדרים היטב.
- מימוש פרוטוקול תקשורת בין מחשבים נקרא **network stack** או **protocol stack**.

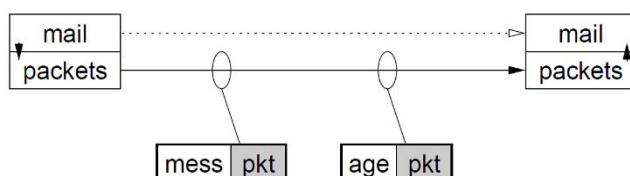
דוגמה – שליחת מייל



המידע במייל הוא בתובת הנמען והתוכן. יכולות להיות בעיות בשילוח הודעות ארוכות שייגרמו מהתווסףות רעש להודעה (חלוף ביטים) ומפרק כל ההודעה לצורק לפח.

הפתרון: חלוקת ההודעה להודעות קטנות. לא יעיל.

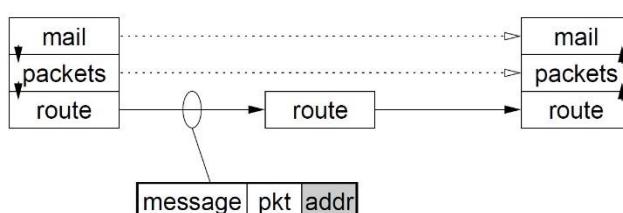
פתרון אחר: הוספת תוכנה בשני הצדדים שפרקת את ההודעה לחטיבות קטנות – packets. לכל חטיבה נוספים אינדקס כדי לידע את סדר ההודעות.



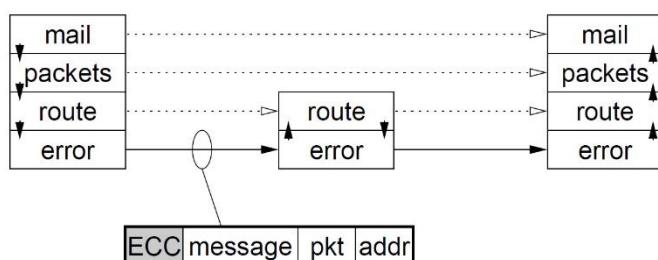
הבעיה: איבוד packets בדרך, הגעה בסדר לא נכון, וכו'.

הפתרון: הוספת מגנון שידאג לבקרה על הזרימה – **routing**.

לכל הודעה, בנוסף לתוכן ול-pkt (packet-h-l-pkt) נוסיף מידע נוסף, את בתובת הנמען.



אבל, יכול להיות מצב של שגיאות ונרצה להתמודד עם חלקן. נוסף קוד לתיקון שגיאות בכל צד.



כל השימוש של מבנה שמצויר לעליה זו בעצם ה-protocol stack. הצד השולח מוסיף headers, מידע נוסף שלא שייך להודעה המקורי. הצד מקבל משתמש כל פעם ב-headers (במו בחציהם).
כל שכבה בין כל צד עובדת בתיאום.

הפרוטוקול של האינטרנט
מורכב מרבע שכבות שכל שכבה נוערת בשירותים של השכבה מתחתיה:

| Layer name | Description (Layer's goal) | Protocols |
|---------------------------|---|-----------------------|
| Application | process-to-process communications | HTTP/S, SSH, FTP, DNS |
| Transport | End-to-end communication services for applications | TCP, UDP |
| Network / Internet | Transport datagrams (packets) from the originating host across network boundaries, if necessary, to the destination host specified by a network address | IP |
| Link / Physical | Communications protocols that only operate on the link that a host is physically connected to. | 802.11 WiFi, Ethernet |

נתמך בשכבה השנייה.

TCP and UDP – transport protocols

שכבת הטרנספורט

שכבה ה-network מתחת לשכבה ה-transport אחראית על העברת הודעות מצד אחד לשני, דואגת לניטוב אבל לא לאمنיות המידע.
ה-transport מוסיפה שכבה של אמינות וספקת שירותים לשכבה האפליקציה שיאפשרו להודעות להגיע בוצרה יותר טובות.
שני פרוטוקולים: TCP ו-UDP (פחות אמינה).

User Datagram Protocol – UDP

שכבה דקה שמוסיפה מעט מידע להודעה המקורי.
 • Source and destination ports. Ports מושרים(packet) לאפליקציה.
 • אורך הודעה
 • Checksum – בדיקת אמינות מינימלית למידע פגום, בגודל הבדיקה היא ע"י xor למידע בך שנמדד אם מספר הביטים זוגי או אי-זוגי. הערך יהיה ב-field checksum. והצד מקבל יריץ גם את אותה בדיקה ויצפה לקבל את אותו ערך.
 עדין השכבה הזאת לא אמינה מספיק – היא לא בודקת דופליקציות, איבוד מידע או סדר לא נכון.
 נזכיר גם connectionless, אין תיאום בין הצדדים לגבי הגעת המידע.

Transmission Control Protocol – TCP

פרוטוקול שמאפשר העברת אמינה של מידע. שני הצדדים מתואימים ובודקים שהפרטים זרים. מחזיקים buffer ומזהדים שאין שליחה של יותר מדי מידע.
בקירה על זרימת המידע, על איבוד פקודות, על סדר נבון, דופליקיציות וכו'.
הבעיה: תקורה גבוהה וזמן איטי יותר.

| Property | UDP | TCP |
|-----------------|------------------|-------------------------------------|
| Reliable | no | yes |
| Connection type | Connectionless | Connection oriented |
| Flow control | No | Yes |
| Latency | Low | High |
| Applications | VOIP, Most games | HTTP, HTTPs, FTP, SMTP, Telnet, SSH |

Sockets

Socket הוא אינטפייס שקיים אצל המארח host שמנוהל ע"י מערכת הפעלה ומאפשר את התקשרות עם הצד השני. דרכו שלוחים ומקבלים מידע.

- ה-socket יכול להשתמש בתקשורת אמינה/לא אמינה.
- ונוצר בmphorsh ע"י האפליקציה שמשתמשת בו.

Socket-programming using TCP

התוכנה יוצרת איזשהו socket שמנוהל ע"י מערכת הפעלה. שני צדדים – לקוח ושרת. כשלוקוח רוצה לתחבר לשרת, בנקודה זו השירות בבר יצר socket דרכו הלקוח יוכל לתחבר. אוח"ב הלקוח יוצר TCP socket משלו דרכו הוא מתחבר ל-socket של השירות ע"י ציון בתובת IP והפורט אליו הוא רוצה לתחבר.

השירות מקבל את החיבור ויאפשר socket חדש עבור אותו לקוח דרכו מתנהלת התקשרות.

- רץ של תווים שעובר בין שני תהליכיים.
- **Input stream** – מידע שמתקבל.
- **Output stream** – מידע שיוציא.

דוגמה פשוטה: תהליך שרעץ אצל הלקוח ומקבל קלט מהמקלדת (שולחת input stream) ואוטו התהילך שלות את ה-socket בהתאם output stream ל-socket של השירות.

Client/server socket interaction: TCP

- השירות:** יוצר socket שנועד לכל הלקוחות, מגדר מספר pocket ולביקשות כניסה welcomeSocket=serverSocket()
- השירות:** מחייב לבקש התחברות.
- הלקוח:** יוצר socket ונסה לתחבר לשרת לפי IP ולפי הפורט אליו רוצה להגיע. ברגע שנוצר חיבור יש "handshake".
- הלקוח:** שולח בקשה מסוימת לשרת דרך ה-socket.
- השירות:** מקבל את הבקשתה.
- השירות:** בותב תשובה ושולח ללקוח.
- הלקוח:** מקבל את התשובה.

5. גם הלקוח וגם השירות סגורים את ה-socket וחוזר חלילה.

Socket programming using UDP

כאן אין את שלב ה-“handshake”, השולח בותב את ה-IP ושולח את המידע בלי לוודא שהצד השני קיבל את המידע ואין דרך לוודא סדר פקודות וכו'.

1. **השרת:** יוצר socket, מגדר מספר pocket ולביקשות נוכנשות ()

הלקות: יוצר socket ע"י ()

2. **הלקוח:** שולח דאטה דרך ה-socket שלו.

השרת: מקבל איזשהו דאטה מהלקוח.

3. השרת יכול לשולח מידע והלקוח יוכל לקבל מידע.

• ה-socket נשאר זהה, אפילו בין ללקוחות שונים.

Technical material

Socket's address

Struct sockaddr

```
struct sockaddr {
    unsigned short sa_family;
    char sa_data[14];
};
```

סטרuktur ששמור בתובות שאליה אפשר להתחבר.
sa_family מציין את סוג הכתובת ששומרה בסטרuktur (למשל כתובות אינטרנט).
sa_data כל המידע של ה-socket אליו מתחברים.

Struct sockaddr_in

```
struct sockaddr_in {
    short      sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};
struct in_addr {
    uint32_t s_addr;
};
```

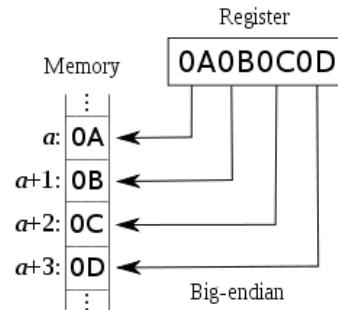
הดาטה מפורק לבמה שדות כדי שתהייה גישה ספציפית למזה שמעוניין.

- ניתן לבצע קאסטיניג בין שני הסטרוקטים.
- memset() מואפס עם sin_zero

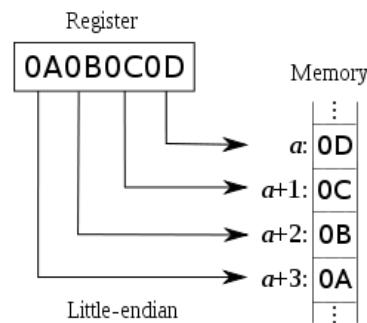
הפורט והכנתובת נשמרים ב-byte order של הרשא.

יש שתי גישות לסדרה:

- **Big endian** .1



- (כותבים הפור) **Little endian** .2



פונקציות המרה:

| |
|--|
| htons() - "Host to Network Short" |
| htonl() - "Host to Network Long" |
| ntohs() - "Network to Host Short" |
| ntohl() - "Network to Host Long" |

דוגמה לשימוש:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
struct sockaddr_in my_addr;
my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(3490);
inet_aton("10.12.110.57",&(my_addr.sin_addr));
memset(&(my_addr.sin_zero), '\0', 8);
```

הfonקציה **inet_aton** מקבלת כקלטning כתובת IP וממיר אותה למספר בינארי שייצג את הכתובת. מוחזירה 0 אם לא הצלחה.

:getpeername()

```
int getpeername(int sockfd,struct sockaddr *addr,int *addrlen);
```

מקבלת בתובות של הצד השני אליו ה-socket מחובר (ה-socket מיוצג ע"י fd)

DNS – Domain Name Serving (mapping names to IP)

פרוטוקול שהמטרה שלו היא לתרגם דומיינים (URL) לכתובות IP.

הפרוטוקול ניגש לאייזהו מבנה נתונים שומרם על מיפויים בין דומיינים לבין כתובות IP.
הfonקציה `(hostname, gethostbyname)` מוחזירה את שם ה-host וניתן להשתמש בה לפונקציה `(hostnet, gethostbyname)`.
שוחזירה פוינטר לSTRUCT hostnet שמכיל את המידע על המארח.

```
#include <netdb.h>
struct hostent*
gethostbyname(const char *name);
```

:hostnet struct

```
struct hostent {
    //Official name of the host
    char *h_name;
    //Alternate names
    char **h_aliases;
    //usually AF_INET
    int h_addrtype;
    //length of each address
    int h_length;
    //network addresses for the host in N.B.O
    char **h_addr_list;
};

#define h_addr h_addr_list[0]
```

דוגמאות לקוד לקליטת IP:

```

int main(int argc, char *argv[]) {
    struct hostent *h;
    if (argc != 2) {
        fprintf(stderr, "usage: getip address\n");
        exit(1);
    }
    if ((h=gethostbyname(argv[1])) == NULL) {
        fprintf(stderr, "gethostbyname ");
        exit(1);
    }
    printf("Host name : %s\n", h->h_name);
    printf("IP Address : %s\n",
    inet_ntoa(*((struct in_addr *)h->h_addr)));
    return 0;
}

```

Socket programming

ה-server:

1. יצירת סוקט:

```
int socket(int domain, int type, int protocol);
```

סוגי הדרטה (type) בסוקט, סוג החיבור:

SOCK_STREAM indicates that data will come across the socket as a stream of characters.
SOCK_DGRAM indicates that data will come in bunches (called *datagrams*).
SOCK_RAW allows bypassing the layers and writing/reading all bytes in the packet.

הדומין הוא בהתאם למשפחת הכתובות. בד"כ מתאפשר רק פרוטוקול אחד. אפשר להשתמש בערך 0 כדי לבחור את הפרוטוקול הדיפולטיבי.

2. קישור הסוקט לכתובת (system call):

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

הפרמטר הראשון הוא ה-*fd* של הסוקט (כמו בקובץ)

3. הקשבה:

```
int listen(int sockfd, int backlog);
```

הגדרת מספר החיבורים לסקט.

דוגמה לשימוש בשלושת השלבים:

```
int establish(unsigned short portnum) {
    char myname[MAXHOSTNAME+1];
    int s;
    struct sockaddr_in sa;
    struct hostent *hp;
    //hostnet initialization
    gethostname(myname, MAXHOSTNAME);
    hp = gethostbyname(myname);
    if (hp == NULL)
        return(-1);
    //sockaddr_in initization
    memset(&sa, 0, sizeof(struct sockaddr_in));
    sa.sin_family = hp->h_addrtype;
    /* this is our host address */
    memcpy(&sa.sin_addr, hp->h_addr, hp->h_length);
    /* this is our port number */
    sa.sin_port= htons(portnum);

    /* create socket */
    if ((s= socket(AF_INET, SOCK_STREAM, 0)) < 0)
        return(-1);
    if (bind(s , (struct sockaddr *)&sa , sizeof(struct
    sockaddr_in)) < 0) {
        close(s);
        return(-1);
    }

    listen(s, 3); /* max # of queued connects */
    return(s);
}
```

.4. המתנה לקריאות:

```
int accept(int sockfd, struct sockaddr *cli_addr, socklen_t *cli_addrlen)
```

דוגמה למימוש:

```
int get_connection(int s) {
    int t; /* socket of connection */
    if ((t = accept(s,NULL,NULL)) < 0)
        return -1;
    return t;
}
```

ה-client

צריך ליצור סוקט ולהשתמש בפונקציה `.connect`

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

```
int call_socket(char *hostname, unsigned short portnum) {
    struct sockaddr_in sa;
    struct hostent *hp;
    int s;

    if ((hp= gethostbyname (hostname)) == NULL) {
        return(-1);
    }

    memset(&sa,0,sizeof(sa));
    memcpy((char *)&sa.sin_addr , hp->h_addr , hp->h_length);
    sa.sin_family = hp->h_addrtype;
    sa.sin_port = htons((u_short)portnum);
    if ((s = socket(hp->h_addrtype, SOCK_STREAM,0)) < 0) {
        return(-1);
    }
    if (connect(s, (struct sockaddr *)&sa , sizeof(sa)) < 0) {
        close(s);
        return(-1);
    }

    return(s);
}
```

~ דוגמאות והסבירים נוספים במצגת ~

הפונקציה select

הזכרנו שסוקט מיוצג כמו קובץ, ע"י fd. אבל לsocket אחד של השרת יכולים להיות מספר fd וצריך להתייחס לכך. אופציה אחת היא ע"י יצירת של thread לכל סוקט, אופציה נוספת היא להשתמש בפונקציה select. הפונקציה מקבלת קבוצות של fd וממתינה שהאחד השני יהיה מוכן.

```
int select (int nfds, fd_set *read-fds, fd_set *write-fds, fd_set *except-fds, struct timeval *timeout);
```

~ כל מיני דוגמאות ופרטים נוספים במצגת ~

תרגול 11 – 11

CPU Scheduling

באשר אנחנו עושים scheduling נרצה לבחור את האלגוריתם הטוב ביותר, ויש לנו כמה קритריונים לשם כך:

- ניצול של המעבד, נרצה לנצל אותו במתא שיווירה. **CPU utilization (max)**. 1
- תפקוד, כמה משימות פריחות זמן מסוימים. **Throughput (max)**. 2
- זמן שבוי תחילת נמצאת ב-ready queue, במצב המתנה. **Waiting time (min)**. 3
- (waiting time + execution time) (min). 4

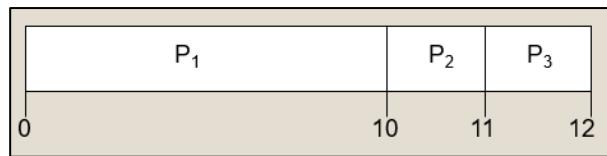
נראה אלגוריתמים שונים ל-scheduling שעונים על קритריונים מסוימים, אבל יתכן שלא יהיו אופטימליים בקריטריונים אחרים.

- זמן ריצה של תחילת ללא ביצוע פעולות פלט קלט. לא מחשבים את הזמן שבו התחלת חסום. **Burst Time**

First Come First Served (FCFS)

FIFO. יש תור של מהלכים וה-scheduler בוחר את המשימה שבראש התור ומבצע אותה עד הסוף.
בדוגמה הריצה של התהיליכים תהיה בדרך הבאה:

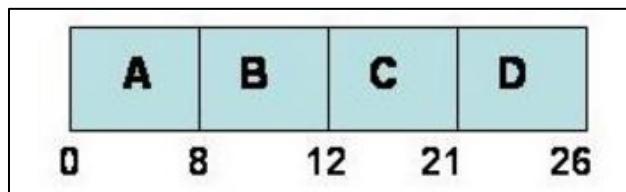
| Process | Burst Time |
|---------|------------|
| P1 | 10 |
| P2 | 1 |
| P3 | 1 |



$$\text{זמן המתנה: } (0 + 10 + 11 + 3\text{cs}) / 3 = 7 + \text{cs}$$

דוגמה נוספת:

| Process | Burst Time |
|---------|------------|
| A | 8 |
| B | 4 |
| C | 9 |
| D | 5 |



ניתן לראות שזמן המתנה לא אופטימי!

| Metric | FCFS |
|------------------|---|
| CPU Utilization | $26 / (26 + 3\text{cs})$ |
| Turn around time | $((8) + (12 + \text{cs}) + (21 + 2\text{cs}) + (26 + 3\text{cs})) / 4 = 16.75 + 1.5\text{cs}$ |
| Waiting | $((0) + (8 + \text{cs}) + (12 + 2\text{cs}) + (21 + 3\text{cs})) / 4 = 10.25 + 1.5\text{cs}$ |
| Throughput | $4 / (26 + 3\text{cs})$ |

- האלגוריתם מבצע זמן הוגן, אין הרעבה.
- זמן המתנה הממוצע יכול להיות גבוה. משימה קצרה יכולה להמתין הרבה זמן אם היא מגיעה אחרி משימה ארוכה, ואם הינו מרים אותה לפני זמן המתנה היה יותר.
- Non-preemptive

Shortest Job First

סדר המשימות מזמן הריצה burst time הנמוך לגבוה. אופטימי מבחינת waiting time הממוצע.

- Non preemptive

הבעיות:

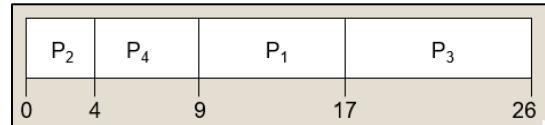
1. מנחים שודעים מראש את זמן הריצה של המשימות.

2. לא הוגן. יתכן שימושה ארוכה תמתין בטור, אבל כל הזמן יגיעו משימות קצרות שיירצו לפניה והוא לעולם לא תרוץ.

האלגוריתם יהיה אופטימלי בבעיית offline - אם יודעים מראש את זמני הריצה של כל המשימות.

דוגמה:

| Process | Burst Time |
|---------|------------|
| P1 | 8 |
| P2 | 4 |
| P3 | 9 |
| P4 | 5 |

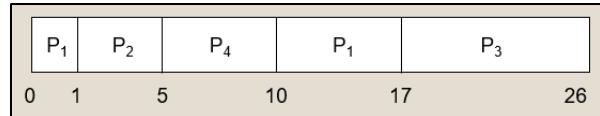


- Average waiting time:** $(4 + 9 + 17 + 0) / 4 = 7.5$
- Using FCFS scheme:** $(0 + 6 + 14 + 21) / 4 = 10.25$

Shortest Remaining Time

תינן עדיפות למשימהשה-remaining time שלה הנמוך ביותר.

- Preemptive
- **לפי הדוגמה הקודמת:**



- Average waiting time:** $((10-1) + (1-1) + (17-2) + (5-3)) / 4 = 6.5$

Priority Scheduling

כל תהליך יש איזושהי עדיפות כך שהם ימצאו בתור לפי העדיפויות שלהם, התהיליכים עם העדיפויות הגבוהה יותר יבוצעו קודם (מספר נמוך - עדיפות גבוהה). SJF הוא כמו מקרה פרטי של תור עדיפויות כאשר עדיפות הגבוהה ניתנת ליגוף עם זמן הריצה הנמוך יותר.

העדיפות נקבעת לפי:

1. (Internal) לפי קבועה של מערכת הפעלה.

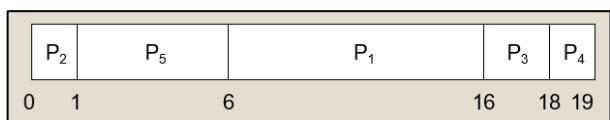
2. (External) לפי מדיניות שמערכת הפעלה מקבלת.

האלגוריתם יכול להיות preemptive - אם מגיעה משימה חשובה יותר אז התהליך שרגע בעת יפסיק ומערכת הפעלה תרים את המשימה החדשה. יכול להיות גם non-preemptive.

הבעיה: יכולה להיות הרעבה. פתרון אפשרי: aging, בכל שהתהליך נמצא יותר זמן ב-queue ready העדיפות שלו עולה בקצב יתבצע.

דוגמה:

| Process | Burst Time | Priority |
|---------|------------|----------|
| P1 | 10 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 4 |
| P4 | 1 | 5 |
| P5 | 5 | 2 |



- Average waiting time:** $(6 + 0 + 16 + 18 + 1) / 5 = 8.2$

Round-Robin

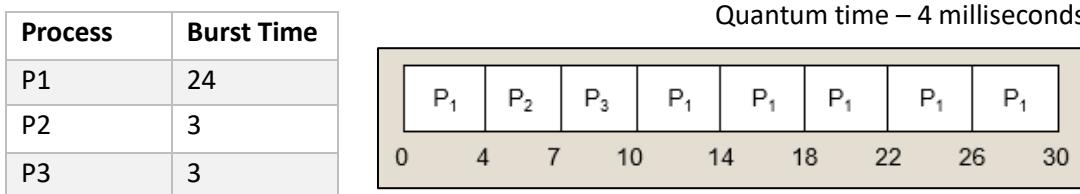
Time-sharing, מחלקים למשתמשים בתור quantum time שזו יחידת זמן שנקבעת מראש. ה-CPU עבר על התור שהוא מעגל ומנוהל ב-FIFO, ומקרה כל תהליך ישחרר את ה-CPU.

יתכנו שתי אפשרויות:

- .1 אם ה-first CPU של אותו תהליך פחות מיחידת הקוונטום שהוקצתה, התהליך ישחרר את ה-CPU.
- .2 אם ה-first CPUburst יותר מיחידת הקוונטום, יבוצע context switch.

preemptive •

דוגמה:



- Average waiting time: $(6 + 4 + 7) / 3 = 5.66$

~ דוגמאות נוספות במצגת ~

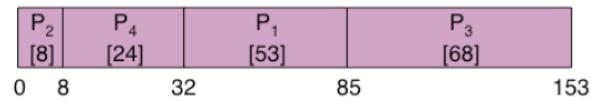
Comparing FCFS and RR

| Job # | FCFS CT | RR CT |
|-------|---------|-------|
| 1 | 100 | 991 |
| 2 | 200 | 992 |
| ... | ... | ... |
| 9 | 900 | 999 |
| 10 | 1000 | 1000 |

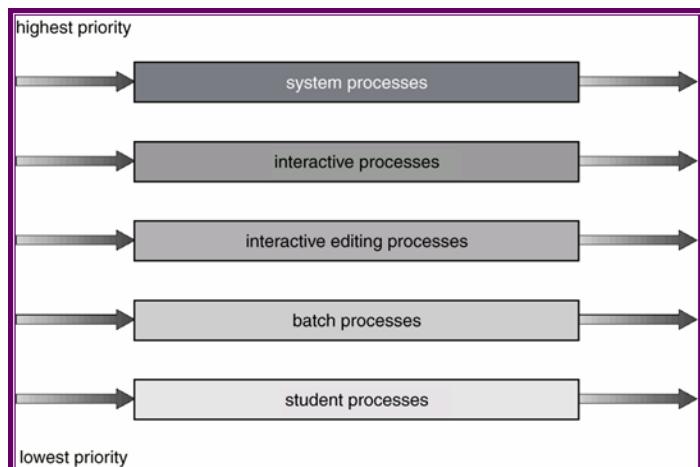
נסתכל על הדוגמה הבאה בה FCFS עדיף על RR: 10 תהליכיים שכבל אחד מהם רץ 100 מיל-שניות, ודומן קוונטום של שנייה אחת. שני האלגוריתמים יסימנו באותו זמן, אבל זמן התגובה של RR הוא יותר איטי, גם אם הה-context switch לא עולה. במקרה זה RR לא עדיף. הוא בעצם עדיף ב"חישום האמתיתים" כאשר יש משימות ארוכות. אבל במקרה קצה שבו מביצעים הרבה המשימות באותו אורך, אין סיבה לבצע RR:

Context switches •

- בשימושים תדריכים ב-FCFS ה-cache misses יועלה כתוצאה מארוך turnaround time.
- SMBADZIM זמן ביצוע - turnarround waiting time מהודע.



| Wait Time | Quantum | P ₁ | P ₂ | P ₃ | P ₄ | Average |
|-----------------|-----------|----------------|----------------|----------------|----------------|---------|
| Best FCFS | 32 | 0 | 85 | 8 | 31½ | |
| Q = 1 | 84 | 22 | 85 | 57 | 62 | |
| Q = 5 | 82 | 20 | 85 | 58 | 61½ | |
| Q = 8 | 80 | 8 | 85 | 56 | 57½ | |
| Q = 10 | 82 | 10 | 85 | 68 | 61½ | |
| Q = 20 | 72 | 20 | 85 | 88 | 66½ | |
| Worst FCFS | 68 | 145 | 0 | 121 | 83½ | |
| Completion Time | Quantum | P ₁ | P ₂ | P ₃ | P ₄ | Average |
| | Best FCFS | 85 | 8 | 153 | 32 | 69½ |
| | Q = 1 | 137 | 30 | 153 | 81 | 100½ |
| | Q = 5 | 135 | 28 | 153 | 82 | 99½ |
| | Q = 8 | 133 | 16 | 153 | 80 | 95½ |
| | Q = 10 | 135 | 18 | 153 | 92 | 99½ |
| | Q = 20 | 125 | 28 | 153 | 112 | 104½ |
| Worst FCFS | 121 | 153 | 68 | 145 | 121½ | |

Multi-level Queue Scheduling

ישנם כמה תורים (למשל תור של משימות רקע, תור של משימות אינטראקטיביות וכו') שעלה כל אחד מהם מופעל אלגוריתם תקין אחר, ויש הזמן בין התורים עצם.

- כדי למנוע הרעה במקורה שההתזמון הוא preemptive יהיה time-slicing בין התורים.

Multilevel feedback queue

תהליכיים יכולים לעבור בין תורים, כאשר בכל מעבר תחשב איזושהי פונקציה של פידבק על התהיליך שלפיה מחליטים על התור הבא בו התהיליך ייכבה.

למשל:

1. אם התהיליך השתמש יותר מדי במעבד אפשר להעביר אותו לתור עם עדיפות נמוכה יותר.
2. אם התהיליך חיכה יותר מדי זמן בטור, "הזדקן", נעביר אותו לתור עם עדיפות גבוהה יותר (מניעת הרעבה).

אלגוריתמים שונים מאופינים ע"י:

- מספר התורות
- אלגוריתמי הזמן של כל תור
- קביעת התור לכל תהיליך חדש שmagiu
- מתי לשדרוג תהיליך או להפר

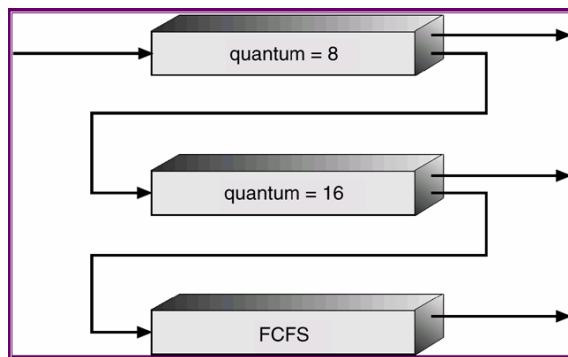
דוגמה:

Three queues:

- Q_0 – time quantum 8 milliseconds
- Q_1 – time quantum 16 milliseconds
- Q_2 – FCFS

האלגוריתם יפעל באופן הבא:

- ג'וב חדש יכנס ל- Q_0 , שפועל כ-FCFS. כשהוא מקבל את ה-CPU מוקצים לו 8 מיליאני-שניות. אם לא סיימ אתון, הג'וב עברו ל- Q_1 .
- ב- Q_1 האלגוריתם המופעל יהיה שוב FCFS והג'וב יקבל 16 מיליאני-שניות נוספות. אם עדין לא סיימ הוא יעבור ל- Q_2 .

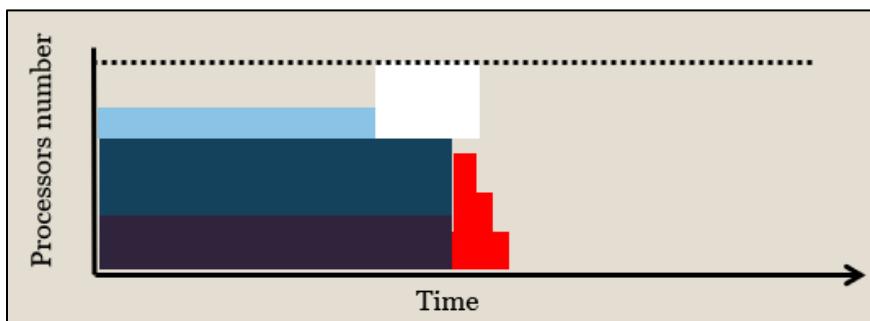
**Parallel System Scheduling**

מחשבים על מריצים הרבה משימות במקביל ולכל מחשב יש מערכת הפעלה ומתחם משלו. בשרתים שעובדים על כמה מחשבים כאלה יש מתחם שקבוע אליו מחשב מטפל באיזו בעיה וגם לכך יש אלגוריתמי זמן שונים.

- אין preemption!

FCFS

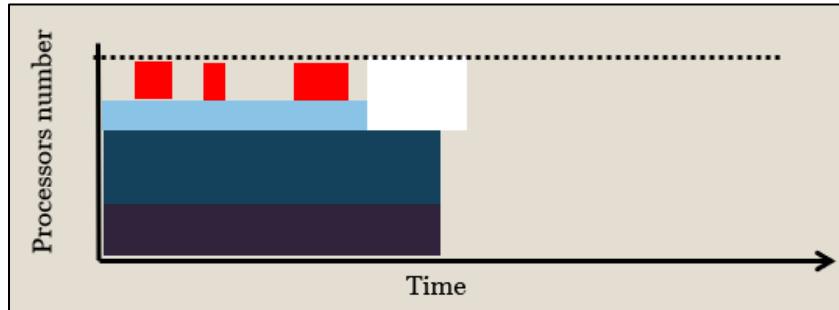
הראשון שנכנס הוא הראשון שմבוצע. ציר ה- y אלה המעבדים שצורך למשימה, ציר ה- x הוא זמן. כל משימה היא מלבד



וניתן לראות שימושות שונות משתמשות במספר אחד וлокציות זמן שונות.
במספר שונה של מעבדים וлокציות זמן שונה.
האלגוריתם לא אופטימלי אבל הוגן. (רצינו
לשבע את הג'וביים האדומים ואת הג'וב הלבן,
ובגרף רואים את המקום שנמצא להם בזמן
(מסויים))

Backfilling

זמןן בצורה ייעלה. לא הוגן. (בגרף היג'וב הלבן הגיע לפני האדומים, אבל שיביצנו אותם לפניו כי להם אין היה מקום)

The Easy Scheduler

אלגוריתם שמשלב את השנאים הקודמים כך שהוא מבצע זמןן הוגן ויעיל.

האלגוריתם מחזיק רשיימה של תהליכיים שרצים בעת עם מידע על המעבדים בהם הם משתמשים וזמן סיום משוער, ורישמה של תהליכיים שמחכים בעת בתורים לפי סדר הגעה ומידע על מספר המעבדים שדרושים וכן ריצה משוערת.

האלגוריתם מבצע Backfilling על בסיס זמן הגעה:

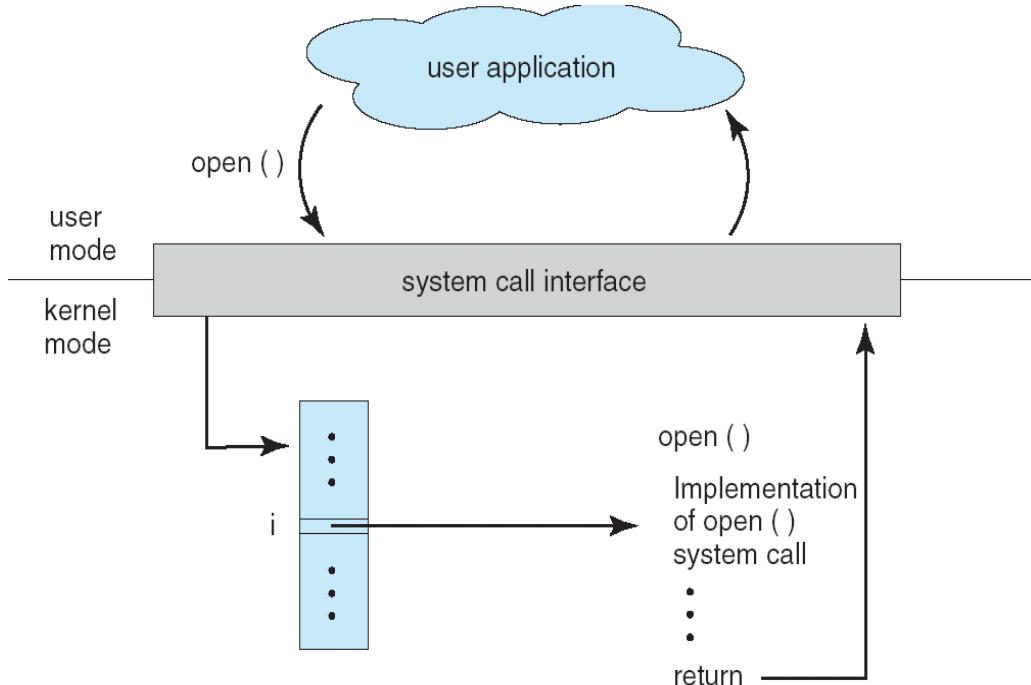
1. משימה מגיעה, ננסה למצאו לה מקום.
2. אם יש - נכניס אותה ונבצע.
3. אם אין נעשה לה "הזמןה" - נמצאת המקום הראשון שיוכל להפעיל אותה, ונשמר לה את אותו מקום.
4. backfilling מתחשב גם בהזמנות עתידיות, באשר הוא ישבץ משימה אחרת הוא יבדוק שאין התנגשות עם הזמןה. הוא ישבץ את המשימה באותו מעבד אם זמן הסיום שלה לא מתנגש עם תחילת הזמןה:
אם כן, ישבץ את המשימה למעבד אחר שם לא תהיה התנגשות עם הזמןה:
Runtime Estimate - כשיוצרים מגישים יוב הם מצינום את מספר המעבדים שדרושים להרצתה וזמן ריצה משוער שנועד כדי לבוא מתי מעבד יהיה פניו להזמנות ולוודא ש-job יסתהים לפני הזמןה. אם תהיה חריגה מהזמן יתכן שה-scheduler ישמיד את התהילין.
~ המחשה גרפית במצגת ~

תרגול 12 – POSIX API Standard

System calls type and API in general

System Calls

תכורת: system calls הן שירותים שניתנים ע"י מערכת הפעלה כדי לבצע פעולות שרק היא יכולה לבצע. היוזר קורא לפונקציה שקוראת ל-call system. ה-call ממוספרות, כך שכאשר הפונקציה נקראת היא כותבת בזיכרון את מספר ה-call ומבצעת trap.



מערכת הפעלה מספקת אינטראפיס גם לKERNEL לביצוע system calls וגם ספרייה לYOU של שימוש באותון קריאות. פונקציות הספרייה נקראות בריגיל והן דואגות לביצוע ה-trap כדי שהKERNEL יטפל בבקשתה ויקרא ל-handler של הקראה הרלוונטי.

| Higher language interface – a part of a system library | Kernel part |
|---|--|
| <ul style="list-style-type: none"> • Executes in user mode • Implemented to accept a standard procedure call • Traps to the Part 2 | <ul style="list-style-type: none"> • Executes in kernel mode • Implements the required system service • May cause blocking the caller (forcing it to wait) • After completion returns back to Part 1 (may report the success or failure of the call) |

מימוש האינטראפיס

מכניסים למחסנית את הארגומנטים ← ביצוע הקראה ← קראה לפונקציית ספרייה שתכתוב את הקוד של ה-call system ← קראה לKERNEL ← ביצוע הפעולה שצורך דרך dispatcher ← החזרת תוצאות לפונקציית הספרייה ← החזרת התוצאות לתהליין.

במצגת יש דוגמאות: .printf(), fread(fd, buffer, nbytes)

System API Standard

- לפונקציות שלא רצוט בקורס יש API סטנדרטי, XPOS, שמשותף בין רוב מערכות הפעלה כדי שייה קל להעיבר תוכניות בין מערכת הפעלה אחת לאחורה.
 - ל-windows יש סטנדרט אחר בשם win64 / win32.
 - תוכניות שרצות ב-JVM (Java Virtual Machine) משתמשות בסטנדרט API Java.

סובי system calls

- Process control and IPC (Inter-Process Communication)
 - Memory management
 - allocating and freeing memory space on request ✓
 - Files access and management
 - Device management

Process Control Calls

כל תהליך יש שיצר אותו, ומשתתף הראשי שרצה ראשון ב-mode user ויצרת את האחרים. כל תהליך יכול ליצור תהליכי אחרים ואוטם תהליכי ירשו את הסביבה שלו, בעצם יהיו שכפול כמעט זהה שלו, עם הזמן ישתנו לפי הצורך. כל תהליך שנוצר מכיל שלושה "קבצים" פתוחים: `fd`, `stdin`, `stdout`, `stderr`, יש להם כבר

pid

הfonקציות הבאות נועדו כדי לקבל את ה-pick של אותו תהילה:

```
pid_t getpid(void) // returns the process ID of the calling process
```

`pid_t getppid(void) // returns the process ID of the parent of the calling process`

fork

הfonctionality יוצרת תהליכי חדש. התהליכי החדש יהיה העתק כמעט מוחדר של התהליכי האבא, מלבד ה-*pick* ועוד כל מינו זרמים בקבוק וביוב זבגה.

ערר הקבוצה של הפקידיה מצינו האם מהלך הרץ עראשיו הוא מהלך האר או הרכז:

1. אם ערך ההזורה הוא 0 התחילה שacz בעת הוא התחילה החדש שנוצר.
 2. אם ערך ההזורה גדול מ-0 התחילה שacz בעת הוא האז' והעריך הוא בעצם ה-pm של התחילה שנוצר.

exit

הפונקציה מסימנת תחיליך רץ, סוגרת את כל הקבצים הפתוחים של אותו תהליך ומשחררת זיכרון ומושאים. אם אישחה מידע נכתב לקובץ הוא נשמר בברפ"ר, ולאחר הסגירה של התחליך מערכת הפעלה כותבת את אותו מידע בדיסק. היא מקבלת ערך status שנعاد בשביב תחיליך האב שיעד עיר תחיליך הבן הסטיטים, כל ערך שהוא לא 0 זו סגיהה. לא מחזירה כלום (למשל בטרמינל שמריץ תוכניות ביתין לרדואות את ערך הייצאה).

```
void exit(int status)
```

wait

פונקציה בטליהר האב שמקבלת כפרמטר פינטער -*to* ומחייב שאייזהו תטליהר מאותם תליליכים שהאב יצר ישטיים כך שעריך החזורה של *to* יכנס לאוטו פרמטר.

```
pid = wait(int *stat_loc)
```

הfonקציה `waitpid` ממחכה לתהילך ספציפי שיסתייעם.

```
pid = waitpid(pid_t pid, int *stat_loc, int options)
```

הפרמטר `options` מאפשר גמישות, אפשר למשל לחובות לSIG_NCL של התהילך הבן (במקום לסיומו).

exec
הfonקציות מקבלות path (נתיב לקובץ executable) ומרגע שהן נקראות התוכנית שבקובץ היא אותה תוכנית שבקובץ.

```
execl( "/bin/ls", "ls", "-l", NULL);
```

```
char *args[] = { "/bin/ls", "-l", NULL};  
execv(args[0], args);
```

דוגמאות:

```
int main(void)  
{  
    pid_t pid;  
  
    pid = fork();  
  
    if (pid == -1) {  
        /*  
         * When fork() returns -1, an error happened.  
         */  
        perror("fork failed");  
        exit(EXIT_FAILURE);  
    }  
    else if (pid == 0) {  
        /*  
         * When fork() returns 0, we are in the child process.  
         */  
        printf("Hello from the child process!\n");  
        _exit(EXIT_SUCCESS); /* exit() is unreliable here, so _exit must be used */  
    }  
    else {  
        /*  
         * When fork() returns a positive number, we are in the parent process  
         * and the return value is the PID of the newly created child process.  
         */  
        int status;  
        (void)waitpid(pid, &status, 0);  
    }  
    return EXIT_SUCCESS;  
}
```

```

pid_t my_pid=getpid(), parent_pid=getppid() , child_pid;
int status;
printf("\n Parent: my pid is %d\n\n", my_pid);
printf("Parent: my parent's pid is %d\n\n", parent_pid);
if((child_pid = fork()) < 0){ perror("fork failure"); exit(1);}
if(child_pid == 0){
    printf("\nChild: I am a new-born process!\n\n");
    my_pid = getpid(); parent_pid = getppid();
    printf("Child: my pid is: %d\n\n", my_pid);
    printf("Child: my parent's pid is: %d\n\n", parent_pid);
    sleep(3);
    execl("/bin/date", "date", 0, 0);
    perror("execl() failure!\n\n");
    _exit(1);
} else{
    printf("\nParent: I created a child process.\n\n");
    printf("Parent: my child's pid is: %d\n\n", child_pid);
    system("ps -acef | grep ercal"); printf("\n \n");
    wait(&status); /* can use wait(NULL) since exit status
                     from child is not used. */
    printf("\n Parent: my child is dead. I gonna leave. \n ");
}
return 0;

```

שאלה מבחן:

נניח כי תהליך עם $\text{pid} = 1$ מרים את הקוד הבא:

```

int a = fork();
int b = fork();
printf("a: %d, b: %d", a, b);

```

במו כן הינו כि ה-pid של התהליכים שנוצרים לאחר מכן מוכן גדים ב-1 (כלומר ה-pid של התהליך הבא שיופיע יהיה 2 וכך). רשםו פלט אפשרי אחד של התוכנית.

פתרון:

נניח שתהליך 1 יוצר את תהליך 2 ואת תהליך 3.

← מנוקדת המבט של תהליך 1 הוא יוצר את תהליך 2 בפורק ומכניס את ערך החזרה של הפונקציה ל-a, ולכן $a = 1$.

← מנוקדת המבט של תהליך 2, מכיוון ש-1 יצר אותו $a = 0$.

← לאחר מכן 1 יוצר את 3 ומוכניס את הערך ל-b $b = 1$.

← דיברנו על כך שתהליכי שנוצר הוא שכפול מדויק של התהליך שיצר אותו ולכן $a = 2$, ומכוון שהוא חוזר לפורמטר b אז $b = 0$.

← אחר כר תהליך 2 ממשיר לווז ויבצע את השורה השנייה, יוצר תהליך חדש בפורק ואת ערך החזרה ישים ב-b. לכן $b = 0$ באותו תהליך.

| 1 | 2 | 3 | 4 |
|----------------|----------------|----------------|----------------|
| $a = 2, b = 3$ | $a = 0, b = 4$ | $a = 2, b = 0$ | $a = 0, b = 0$ |

סדר הייצירות התהליכים יכול להיות שונה, וכך הכל יש 48 אופציות שונות להדפסה.

Memory Management Calls & File Access CallsMalloc and Free

הfonקציה `malloc` מבקשת זיכרון פנוי בגודל הfrmטר שהוא מקבלת ומחזירה פוינטר למקום ממנו הזיכרון מתחילה.

```
void *malloc(size_t size)
```

הfonקציה `free` מבקשת פוינטר למקום בזיכרון שהוקצתה ומשחררת אותו כדי שניתן יהיה להשתמש בו שוב. אם הפוינטר שהוכנס לה בfrmטר לא הוקצה ע"י `malloc`, או אם אותו מקום בזיכרון כבר שוחרר, הfonקציה מתנהגת בצורה לא מוגדרת.

```
void free(void *ptr)
```

File Access Calls

סוגי קבצים אפשריים בלינוקס (גישה אליהם תעשה באותה דרך באמצעות קבצים וגלים):

- Regular file
- Symbolic link
- Directory
- Character devices
- Block devices
- Pipes
- FIFO special file
- Sockets
- Random number generators

open

פתיחת קובץ. הfonקציה מוחזירה את ה-`fd` של הקובץ שזה המספר האי שלילי היכי נמור שלא בשימוש ע"י קבצים אחרים. הfrmטר `flag` משפיע על הגישה לקובץ ויכולם להישלח דרכו:

- `ReadOnly, WriteOnly, ReadWrite`
- `Create, Append, Exclusive`
- `O_direct` כתיבה ישירה לזכרון
- `O_sync` כתיבה שתתבצע באופן מיידי

```
fd = open(const char *path, int flag, ...);
```

close

סגירת קובץ וחרורו ה-`fd` שלו, כך שפעם הבאה שנפתח קובץ חדש יוכל להשתמש ב-`fd`. סגירה מוחלטת של הקובץ תתבצע רק אם כל ה-`fd` שלו ייסגרו.

```
err = close(int fd);
```

read and write

read קוראת `bytes` בתים מהקובץ שמצויה עם ה-`fd` לערך הבפר אליו מצביע הפוינטר `buf`. הfonקציה מוחזירה את מספר הבטים שהצליחה לקרוא.

```
b_read = read(int fd, void *buf, int nbytes);
```

write כתבת bytes בתים מסוימים בפרק אליהם מציין הפעלתה `buf` בתוך הקובץ המצוין ע"י `fd`. הפקציה מחזירה את מספר הבתים שהצליחה לכתב.

```
b_written = write(int fd, void *buf, int nbyte);
```

lseek

אפשרות להזין את האופט, את הסמן של הקובץ. הפקציה מקבלת את ה-`fd` ואופט.

הפרמטר `whence` יכול לקבל:

- SEEK_SET – הסמן יהיה בתחילת הקובץ.
- SEEK_CUR – הסמן יהיה בנקודה + הפרמטר אופט.
- SEEK_END – הסמן יהיה בסוף. אם הסמן חיובי מרכיבת הפעלה תגדיל את הקובץ בגודל אופט. אם שלילי נסתכל על הקובץ במיקום האופט ממסוף.

הפקציה מחזירה את המיקום החדש של הסמן.

```
where = lseek(int fd, off_t offset, int whence);
```

כדי לדעת את המיקום הנוכחי ניתן לכתוב:

```
where = lseek(fd, 0, SEEK_CUR);
```

dup

MSCPFLT קובץ פתוח, כלומר לקובץ אחד יהיה שני `fd`. מקבלת `fd` שמצוין לאיזשהו קובץ ומגדירה לו אותו `fd` חדש (האו-שלילי הראשון שפנוי) כך שגם החדש וגם הישן מצוינים באותו קובץ.

```
fd_new = dup(int fd);
```

הפקציה `2dup` מקבלת `oldfd` של הקובץ ו-`newfd` חדש ומשנה את ההצבעה של ה-`fd` החדש לאותו הקובץ אליו מציביע ה-`fd` החדש. אם ה-`fd` החדש מציבע לקובץ פתוח הפקציה תסגור את הקובץ הקודם.

```
int dup2(int oldfd, int newfd);
```

הפקציה `stat` מחדירה מידע על הקובץ בתוך struct `stat` שהוא מקבלת.

```
err = stat(const char path, struct stat *buf);
```

chmod

משנה הרשות של קובץ. היא מקבלת `path` של קובץ ומספר `mode` שמצוין את הרשות (הסביר בתרגילים קודמים).

```
err = chmod(const char *path, mode_t mode);
```

Inter-Process Communication

הפקציה `pipe` יוצרת עורך תקשורת בין תהליכים. הפקציה מקבלת מערך של `fd` בגודל 2 ומגדירה צינור בין שניהם כך שהם יכולים לתקשר, המקום ה-0 קורא והמקום ה-1 כותב.

err = pipe(int fd[2]);

דוגמה:

- ← יוצרים סודיק שמקבל שני קבצים.
- ← אחר כך יוצרים ע"י fork תהליך חדש.
- ← אם התהיליך שרע בעת הוא החדש, הוא יקרה ל-dup על המערך שקיבל ה-pipe במקומות ה-1, ויחליף את ה-stdout באותו קובץ. אותו תהליך יכתוב את המידע.
- ← התהיליך סוגר את ה-fd של ה-pipe ומבצע exec כך שהפלט יכתב ב-[1].pipefd2[1].
- ← יוצרים תילך חדש שסגור את ה-chin, stdin, עושה אותו דבר כמו התהיליך הקודם. הקלט של הפונקציה יגיע מה-pipe, יהיה מה שנכתב ע"י התהיליך הקודם.
- כלומר התוצאות של הפקודה זו יכתבו לאותו fd שניתן ב-execl במקום ב-stdout.

```
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
int main() {
    int pipefd2[2];
    pipe(pipefd2);
    if (fork() == 0) {
        dup2(pipefd2[1], STDOUT_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/bin/ls", "ls", NULL);
        exit(EXIT_FAILURE);
    }
    if (fork() == 0) {
        dup2(pipefd2[0], STDIN_FILENO);
        close(pipefd2[0]);
        close(pipefd2[1]);
        execl("/usr/bin/file", "file", "-f-", NULL);
        exit(EXIT_FAILURE);
    }
    close(pipefd2[0]);
    close(pipefd2[1]);
    wait(NULL);
    wait(NULL);
    return 0;
}
```

File & Directory Management Callsdirectory

mkdir יוצרת תיקיה חדשה. הparameter mode מגדיר את הרשאות של התיקיה:

```
err = mkdir(const char *path, mode_t mode);
```

rmdir מוחקת תיקיה קיימת אם היא ריקה:

```
err = rmdir(const char *path);
```

chdir משנה את התיקיה הנוכחי של התהילך ל-path שהוא מקבלת:

```
err = chdir(const char *path);
```

link & unlink

דיברנו על כך שכל קובץ מכיל את התוכן שלו ואת המטה-דאטה (inode), אבל לא את השם ואת ה-path. יש אפשרות לחתך אחד שני שמות כך שהוא ימצא בשני מקומות שונים. כל ילין למיקום זהה יקרא **hard link**. קובץ יהיה מספר רפרנסים – מס' ה-path links שמצוירים אליו. כאשר מס' זה הוא ניתן למחוק את הקובץ מהdisk. הפונקציה **link** יוצרת link חדש לקובץ ומגדילה את מונה הרפרנסים. היא מקשרת קובץ שנמצא ב-oldpath לשם חדש ב-newpath.

```
err = int link(const char *oldpath, const char *newpath);
```

לעומת זאת הפונקציה **unlink** מוחקת את ה-linkhard ובערך מקטינה את מונה הרפרנסים, ואם הוא שווה ל-0 הפונקציה תמחק את הקובץ (אם נעשה לקובץ open לפני הקובץ לא ימחק פיזית מהdisk, אלא רק בקריאה ל-close).

```
err = unlink(const char *path);
```

Device Management Calls

גם devices נחישבים בקבצים במערכת ההפעלה ונitin לתקשר איתם באמצעות הפונקציה ioctl. הפונקציה מקבלת fd שמייצג את אותו device ו-request, מה שנרצה שיבצע.

```
int ioctl(int fd, int request, ... /* arg */)
```

Other CallsKill

הfonktsia שלוחת סיגナル לתהילך אחר, אמצעי לתקשורת בין תהילכים.

```
err = kill(pid_t pid, int sig);
```

Signal

הfonktsia מבקשת פונקציה שמתקבלת בתור handler לSIGALRM.

```
void (*signal(int sig, void *func)(int)))
```

fd set
 זה struct fd_set שמתאר קבוצה של fd שיכולים להישלח לפונקיות מסוימות. מכיל מספר בינארי שכמות הביטים בו היא בכמות ה-fd הקיימת.

FD_ZERO מרוקנת את הקבוצה ובעצם מאפסת את הביטים:

FD_ZERO(fd_set *fdset)

יכולה למחוק fd ספציפי ע"י מהיקת הביט המתאים (מספר fd):

FD_CLR(int fd, fd_set *fdset)

מכניסה קובץ חדש לקבוצה ע"י הדלקה של הביט המתאים:

FD_SET(int fd, fd_set *fdset);

.fdset – פונקציה בוליאנית שמחזירה האם הביט ה-fd מותאם ל-fdset.

FD_ISSET(int fd, fd_set *fdset);

select

פרמטרים:

- **שלוש קבוצות fd** – קריאה, כתיבה ו-exceptions (אפשר לשЛОח NULL אם לא רצים להתייחס לכך).
- **timeout** – כמה זמן נרצה לחכות עד שנסיים לבצע את הפעולות על הקבצים.
- **nfds** – מספר ה-fd האגדול ביותר ששלהנו מבין כל הקבוצות + 1.

הפונקציה מתחילה עד שהקבצים המוחכים לקריאה יהיו מוכנים, וכך'ל לגבי הקבצים המוחכים לקריאה. הפונקציה בעצם משנה את הקבוצות האלה ומשאירת את הקבצים שבאמת מוכנים, כך שנוכל להיות בטוחים שכשנקרא/נכתב לא תהיה שגיאה.

int select (int nfds, fd_set *read-fds, fd_set *write-fds, fd_set *except-fds, struct timeval *timeout)

```
fd_set rfd;
struct timeval tv;
int retval;

/* Watch stdin (fd 0) to see when it has input. */
FD_ZERO(&rfd);
FD_SET(0, &rfd);
/* Wait up to five seconds. */
tv.tv_sec = 5;
tv.tv_usec = 0;
retval = select(1, &rfd, NULL, NULL, &tv);
/* Don't rely on the value of tv now! */
if (retval == -1)
    perror("select()");
else if (retval)
    printf("Data is available now.\n");
    /* FD_ISSET(0, &rfd) will be true. */
else
    printf("No data within five seconds.\n");
```

דוגמיה:

השוואה בין POSIX לבין Win32 (העשרה)

| POSIX | Win32 | Description |
|--------------|----------------------|---|
| fork | CreateProcess | Create a new process |
| wait | WaitForSingleObject | The parent process may wait for the child to finish |
| execve | -- | CreateProcess = fork + execve |
| exit | ExitProcess | Terminate process |
| open | CreateFile | Create a new file or open an existing file |
| close | CloseHandle | Close a file |
| read | ReadFile | Read data from an open file |
| write | WriteFile | Write data into an open file |
| lseek | SetFilePointer | Move read/write offset in a file (file pointer) |
| stat | GetFileAttributesExt | Get information on a file |
| mkdir | CreateDirectory | Create a file directory |
| rmdir | RemoveDirectory | Remove a file directory |
| link | -- | Win32 does not support “links” in the file system |
| unlink | DeleteFile | Delete an existing file |
| chdir | SetCurrentDirectory | Change working directory |

תרגול 13 – SUMMARY



Interrupts

Interrupts – סיגנל ממוקבetta הפעלה שמודיע לה על כך שיש צורך לטפל במשהו.

שני סוגי interrupts:

1. **External Interrupts** – interrupts שנגרמים ע"י רכיב חומרה (DISK סיום, נלחץ בפטור בעכבר).
 2. **Internal Interrupts**
- ניתנים לחלוקת גם ע"י:
- interrupts – **Maskable**
 - או אפשר להתעלם מהם.
 - או לא אפשר להתעלם. **Non-maskable**

Internal Interrupts

interrupts שגורים במהלך ריצת תוכנית ב-CPU (שגיאות, system calls). לעיתים תהיליך צריך עזרה של מערכת ההפעלה כדי לבצע איזושהי פעולה.

דוגמאות:

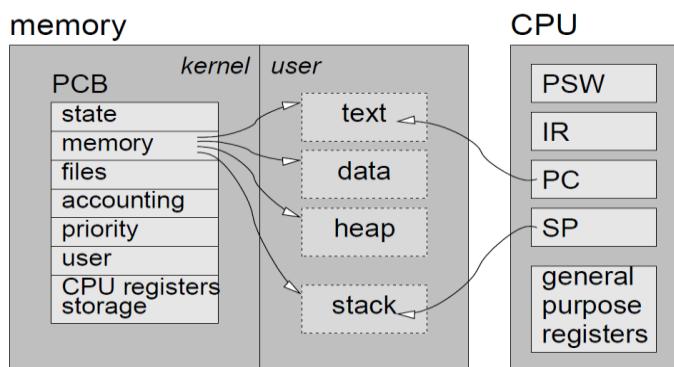
- Division by zero
 - Segmentation fault
 - Privileged instruction
 - Invalid instruction
 - Page fault – למשל בשאייזהו page לא נמצא בזיכרון נצטרך ליבא אותו וזה קורה ע"י שhiba'a לשילטה של מערכת ההפעלה על הריצה וטעינת ה-page לזכרון. התהיליך לא מודע לכך.
 - .system call – trap – במקרה מבצע call self.
- ברגע שה- handling של האינטראפט הסטיים חוזרים לאיופה שעיצרנו.

התמודדות עם interrupts פנימיים

1. שימרת מצבו הנוכחי
2. העברת שליטה למערכת ההפעלה וטיפול בבקשתה
3. החזרה למצב הקודם
4. החזרת השילטה לתהיליך

Signals

בניגוד ל-interrupts שנשלחים למערכת ההפעלה, הסיגנלים נשלחים לתהיליכים כדי להודיע להם על "אירועים" חשובים. התהיליך יצטרך להפסיק את כל הפעולות ולטפל בהפרעה. התהיליך יכול לטפל בהם ע"י פונקציית handle דיפולטיבית שספקה ע"י מערכת ההפעלה או להציג handler מסויל עצמוני.



Process Control Block (PCB)

בלוק בזיכרון ששומר לכל זיכרון ובו שומר כל המידע ששzier לתהיליך:

- State
- Memory – מצביעים לטבלאות הדפים
- Files – טבלת ה-fd
- Accounting – סטטיסטיות של התהיליך, כמה זמן רץ וכו'.
- User
- CPU registers

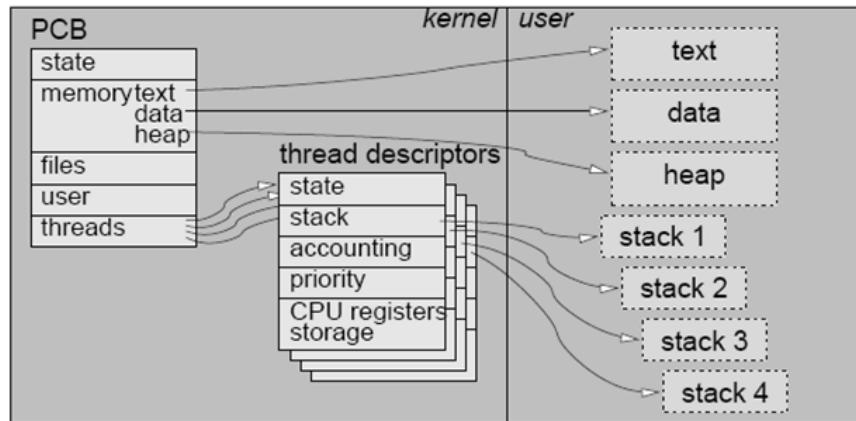
ולוק בזיכרון ששומר לכל זיכרון ובו שומר כל המידע ששzier לתהיליך:

- State
- Memory – מצביעים לטבלאות הדפים
- Files – טבלת ה-fd
- Accounting – סטטיסטיות של התהיליך, כמה זמן רץ וכו'.
- User
- CPU registers

Threads

ה-threads נועד לחלק משימה אחת שתהיליך מרץ לתתי משימות כדי שתתבצע בצורה אופטימלית יותר. Threads של אותו תהיליך חולקים ביניהם heap, קוד ודאטא. הרבה פחות תקורה מאשר תקשורת בין תהיליכים.

Kernel level threads Threads שמערכת הפעלה מנהלת.



User level threads

- היוור מנהל את ה-scheduling ומחזיק את טבלאות threads, מקצה stack stack לכל thread ב-heap של התהיליך. ה-threads לא מודיע לThreads, מבינתו יש תהיליך אחד.
- יתרונות: context switch מהיר יותר, שליטה של היוור.
 - חסרונות: thread אחד משפיע על כל התהיליך. אם thread נפל התהיליך נפל, אם מצב קרייה למשבץ כל התהיליךמושעה.

CriticalSection Problem

הבעיה: ח תהיליכים רצים במקביל, אין הנחות מקידימות על הריצה שלהם. יש משאב משותף (הקטע הקרייטי) לכל התהיליכים ולא נרצה שייגשו אל אותו משאב במקביל. נרצה סנכרון בגישה של התהיליכים למשאב.

קריטרונים להצלחה בפתרון הבעיה

1. **Mutual exclusion** – רק תהיליך אחד יכול להיכנס לקטע הקרייטי בכל רגע נתון.
2. **Progress** – אם יש תהיליכים שרוצים להיכנס לקטע הקרייטי בסופו של דבר מישו יוכנסו.
3. **Starvation free** – אם יש תהיליך שרוצה להיכנס הוא מתישו יצילח, לא יחכה לניצח.
4. **Generality** – עובר לכל מספר תהיליכים שרוצים.
5. **No blocking in the reminder** – אף תהיליך שנמצא מחוץ לקטע הקרייטי לא ייחסם תהיליך שכן נמצא.

ראיינו במה סוג פתרונות:

- פתרונות hardware (פתרונות שהחומרה מספקת, למשל set and test, למשת).
- פתרונות של מערכת הפעלה (מוסיקסים וסمفוריים)
- פתרונות אלגורitmיים שתהיליכים יכולים למשוך

TSL (test and set lock)

רכיב חומרתי שמקבל פינטער למפתח, מכניס לו ערך חדש ומוחזיר את הערך הישן שלו. הפעולה היא פעולה אוטומטית ולא תהיה הפרעה לתהיליך ברגע שייעשה זאת.

קוד לשם האינטואיציה:

```
# define LOCKED 1
int TestAndSet (int* lockPtr) {
    int oldValue = *lockPtr;
    *lockPtr = LOCKED;
    return oldValue;
}
```

וככל למש מוטקס עם שימוש ב-TSL:

```
int lock=0; // shared variable
.....
while (TestAndSet(&lock)) // busy wait
critical section
lock = 0
```

בכל התהליכים ינסו לנעול את TestAndSet. רק אחד מהם יוכל לנעול, עברו את הלולאה ולהגיע לقطع הקרייטי.
הבעיה: הרעה.

Semaphore

משתנה int שמאוחל בערך, והוא מייצג את כמות התהליכים שנרצה שיבנסו לقطع הקרייטי. משמש בעיקר לקריאה של כמה תהליכי ממשאב משותף. למשתנה יש שתי פעולות בסיסיות שמשנות את ערך הסמפור:

```
Down // also called wait()
down(S) {
    while(S <= 0)
        // busy wait
    S--;
}
```

```
Up // also called signal()
up(S) {
    S++;
}
```

Deadlock

קייפאון – מצב בו הרבה תהליכים רצים לגשת לממשאב משותף ובולם נחסמים.

יכול לקרות אם כל התנאים הבאים מתקיימים בו זמן:

1. רק תחיליך אחד יכול להיכנס לממשאב משותף.
2. כאשר תחיליך מחזיק מפתח לממשאב משותף הוא יכול לבקש מפתחות לממשאים נוספים.
3. לא יכול לקרות מצב שלוקחים מתחילה את המפתח שלהם.
4. שרשרת של המתנות.

Glossary for deadlocks

- **The ostrich approach** – הטעלות. נפוצה במערכות הפעלה לפי גישה בה אם זה קורה זה באשמה התהליך.
- **Detection and recovery** – ננסה לזהות את deadlock ולפתור אותו.
- **Avoidance** – ניסיין להימנע מה-deadlock.
- **Prevention** – ניסיין להפר אחד מרבעת התנאים.

Scheduling

חלוקת ה-CPU בין הרבה תהליכים שונים.

למה לא נרצה לתת לתהיליך אחד לרווח מההתחלת עד הסוף?

- זמן המתנה גבוה בגלל תהליכי אחרים.
- תהליכיים שמבצעים system call יעבדו תהליכי אחרים.
- אי אפשר לבצע משימות במקביל.

הפתרון: time-sharing, חלוקת המעבד בין כמה תהליכים.

האלגוריתמים

FCFS (First-Come, First-Served)

SJF (Shortest-Job-First) – מרכיבים את הזמן הכי קצר. הבעיה: אי אפשר לדעת מי הזמן הקצר ביותר מראש, יכול לגרום להרעה.

SRTF (Shortest Remaining Time First) – מרכיבים את הזמן שנותר לו הכי פחות זמן לרווח. גם יכולה להיות הרעה.

Priority Scheduling – הרצת תהליכים לפי תיעודו.
Round Robin

קריטריונים להערכת אלגוריתם תזמון

1. **CPU utilization (max)** – כמה זמן ה-CPU מרים הוראות "אמיות" (לא הוראות של מערכת הפעלה).

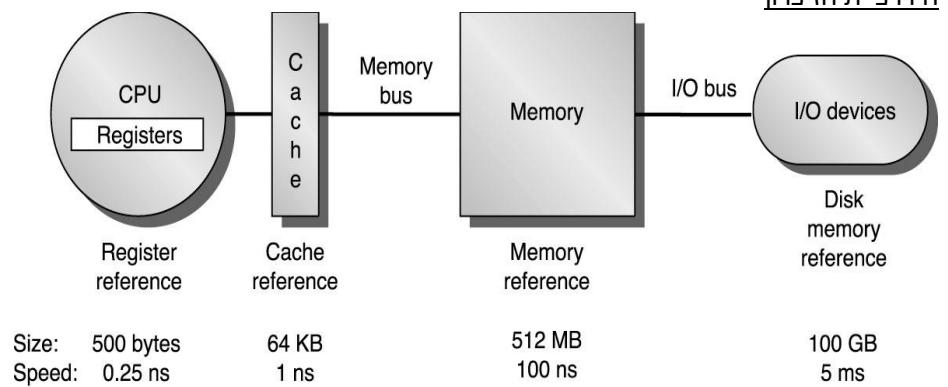
2. **Throughput (max)** – מספר תהליכיים ממשיים את הריצה שלהם ביחד בזמן.

3. **Waiting time (min)** – זמן המתנה המוצע של כל תהליך.

4. **Turnaround time (min)** – הזמן שבו תהליך קיים במערכת מהרגע שהוא עד הרגע שישים.

Caching & Memory Management

היררכיית הזיכרון



גודל ה-CPU מוגבל ולכן יש את הזיכרון הראשי, אבל הגישה אליו יותר איטית. לכן יש את ה-cache שמחולק לכמה רמות והגישה אליה מהירה יותר הדיסק שהוא לא נדי וגודלו שלו עצום.

אלגוריתמים לניהול ה-cache

Optimal – ideally – אם היינו יודעים בעתיד למה ניגש איזה דע מה לשומר ב-cache. לא ישים.

NRU (not recently used) – קשה לימוש. מי שלא השתמשו בו בזמן האחרון.

FIFO, second chance fifo

LRU (Least Recently Used)

LFU (Least Frequently Used)

Random

Memory management

דיברנו על כך שכל תהילך מכיר מרחב זיכרון לוגי ולא מודיע במצב הזיכרון האמיתי ומערכת הפעלה צריכה לתרגם בתובת וירטואלית לבתובת פיזית. יש בחומרה רכיב בשם MMU שאחראי על התרגום.

גישה לניהול זיכרון:

- Contiguous allocation •
- Segmentation •
- Page •

Page tables

חלוקת הזיכרון הוירטואלי ל-pages והזיכרון הפיזי ל-frames בך שכל frame יכול לאחסן page אחד.

גם טבלאות הדפים לוקחות מקום רב בזכרון ולכן יש שיטות שוחשות במקום זה:

- Inverted Page Table •
- Multi-level page table •

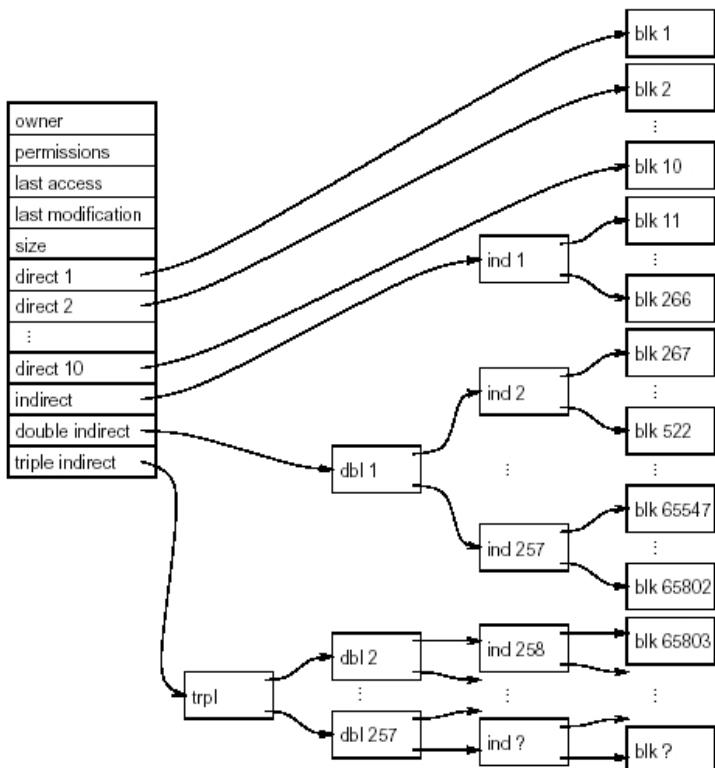
בנוסף יש את ה-TLB שהוא רכיב cache חומרתי שמאחסן מיפויים של דפים שונים אליו אליהם במדדיות גבוהה והגישה אליו מאוד מהירה.

תקשורות בין תהליכיים

- Shared Memory •
- Pipes •
- Sockets (including TCP and UDP) •

Filesystem

המטרה: להסוך בגישות לדיסק (שלוקחות הרבה זמן).
לכן מערכת הפעלה מחזיקה ב-buffer cache, כמו cache לדיסק, ששמור בזכרון הפיזי. גם לגישה אליו קיימים אלגורитמים שונים.



The Unix inode הוא מבנה נתונים שמחזיק את המטה-דטה של קובץ (או תיקייה).

בנוסף למטה-דטה, ב-inode יש פוינטורים (בערך 23) לתוכן הקובץ – **מצביעים ישרים**.

בד"כ גם יהיו ב-inodes **מצביעים עקיפים** שבוספו של דבר יציביעו על ישרים:

- One single *indirect* pointer - points to a whole block of additional direct pointers (Allows medium files)
- One *double* indirect pointer - points to a block of indirect pointers. (Allows large files)
- One *triple* indirect pointer - points to a block of double indirect pointers. (Allows huge files)

File tables

- – לכל תהליך. כל פעם שתהlixir יוצר קובץ מוסיפים שורה לטבלה שמכילה מצביע לטבלה השניה.
- – טבלת הקבצים הפתוחים, משותפת לכל התהליכים. כל קובץ שנפתח יוסיף לטבלה. אם נפתח כמה פעמים יהיו לו כמה שורות. בנוסף ישמר האפסט, הסמן של הקובץ באוטו התהlixir. בשוגרים קובץ תמחק השורה בטבלה. השורות בטבלה יצביעו לשורות בטבלה הבאה.
- – inode – ה inode של הקובץ. מספר המצביעים אליו מהטבלה הקודמת יהיה במספר הפעמים שהקובץ נפתח.

