

OPERATING SYSTEMS

GEORGE SALMAN

Remark. These notes are based on Operating systems lectures given by Associate Prof. Leoanid Raskin at the Technion. Please note this text may contain error.

CONTENTS

Introduction.	3
The OS is reactive.	3
Processes.	4
Signals.	6
Multiprocessing	10
File descriptor.	10
In Posix, “every thing is a file”.	10
Scheduling.	13
SJF.	15
Convoy effect.	16
Performance metrics of preemptive schedulers.	17
RR scheduling.	17
Price of preemption.	17
Batching vs. preemption.	17
SRTF (Shortest-Remaining-Time-First)	19
Selfish RR Algorithm.	20
Scheduling using priorities.	20
Negative feedback principle.	20
Multi-level priority queue.	21
The linux scheduler.	22
Epoch.	23
static/dynamic priorities	23
HZ, resolution, ticks	24
per-task scheduling info	24
task's nice (kernel vs. user)	24
task's counter	25
processor, need_resched, mm	26
Scheduler is comprised of 4 functions	26
Synchronization.	29
Context	29

Cache consistency – relates to different locations	36
Towards a solution: critical sections	37
Towards a solution: requirements	38
Solution: locks	39
Implementing locks	40
FINE-GRAINED SYNCHRONIZATION	40
Kernel spinlock issues	42
Spin or block? (applies to kernel & user)	43
Semaphore – concept	44
Semaphore – interface - (Contain two operations)	44
• Signal(semaphore) – (opposite behaviour of wait)	45
Semaphore vs. spinlock	45
Producer/consumer	45
Semaphore implementation.	48
Amdahl's law.	49
DeadLock.	50
Dining philosophers – naive solution	50
Resource allocation graph	52
Recall the formal definition of deadlock	55
Necessary conditions for deadlock	55
Prevention = violate 1 of the 4 conditions.	56
No “mutual exclusion” (#1)]	57
Deadlock detection	58
Recovery from deadlock after detection	59
Banker’s algorithm for deadlock avoidance.	59
Banker Algorithm.	60
Virtual memory	62
Virtual memory concepts.	62
Virtual memory (vmem) – motivation (Read them all to understand).	64
Virtual memory – terminology	64
Per-process virtual memory simplistic illustration.	65
Virtual memory – basic idea	65
Major page fault	67
Minor page fault	68
On-demand paging & readahead	70
Working set	71
Virtual memory: did we achieve our goals?	72
But how does it perform?	72
Reminder.	73
Page reclamation algorithms	75
64 BIT POWER-PC Architecture.	85
Intel vs POWER-PC ARCHITECTURE.	87

Introduction.

ABSTRACT. What is an operatin system? How can we run two application together in which each one try to access the same address in memory? How are we going to manage for running apps? How can we run same apps on different hardware? How operating system help manage? What is operatin system?

Answer: a piece of software acting as an intermediary between user apps and computer hardware.

Remark. If for each thing we access the operating system, our apps will be in trouble. hence, some of basic staff we can access in direct way such as memory, registers. I.e everything which require more than adding substracting registers we need operating system.

THE OS IS REACTIVE.

ABSTRACT. Most of the apps we use finish unlike operating systems. They run and do it's calculating then it finish work and never return back which the main finish. Operating system has no main and no calculate to for itself, and it never want to run, almost of the time operating system “chill” till some app ask for it help such as “user procces” like opening a file. In the moment operating system finish it go back for sleep.

	typical programs (= app = process) we've seen thus far	OS
what does it typically do?	get some input, do some processing, produce output, terminate	waits & reacts to “events”
structure	has a <code>main</code> function, which is (more or less) the entry point	<code>no main</code> ; multiple entry points, one per event
termination	end of <code>main</code>	always waits for next event
typical goal	terminate as soon as possible	handle events as quickly as possible => more time for apps to run

- Operating system give for each app a time to run.
- Each app can't access a file without help of operting system. The app call OS for help using systemcall then it handle it.

The OS performs recourse managment.

- It's important to know that multiple apps run concurrently on one core only.

The OS provide services .

- Each service invloves two aspects.
 - Abstraction: we don't want to write code of hello world 300 times on each architicture. So it hides Hard ware datails, making interaction with hard ware easier.
 - Isolation: If we run two different apps, we want OS run them with out letting one harm the other. Other isolation, we want OS run proccesses

the most faster. (No app see the data of others, in case there is no data sharing between the two apps).

Different ways to view an OS:

- (1) By its programming interface
 - (a) its system calls.
- (2) According to the services it provides/
 - (a) Time slicing (how multiprogramming is implemented as explained earlier), file systems and their operations.
- (3) According to its internals, algorithms, data structures.

OS components can be studied in isolations;

- Process handling
 - processes are the agents of processing; the OS creates them, schedules them, and coordinates their interactions.
- Memory management
 - memory is allocated to process as needed, but there might be enough for all, so paging is used.
- File system
 - Files are an abstraction providing named data repositories based on disks.

Some HW mechanisms to support the OS.

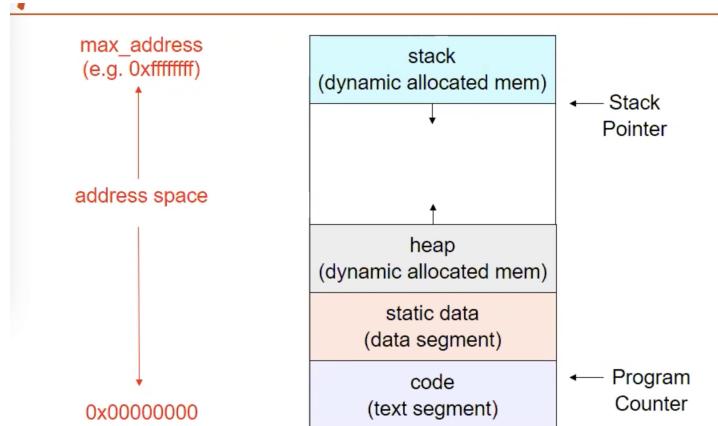
- Interrupts
 - Such as HW clock, network, disk..
- Protected/unprotected modes & privileged instructions
 - in x86L ring 0 vs. ring 3
 - * App run in ring 3 can't for example access a disk. In order to allow app use a disk we call OS so it can access it.
- System calls
- Atomic synchronization operations
- Memory virtualization & protection
- I/O
 - DMA (=direct memory access)
 - MMIO (=memory-mapped I/O)
 - IOMMU (=I/O memory management unit)
- ...

Processes.

- (1) What is a process?
- (2) Data structures to manage process
- (3) context switch.
- (4) Manage process using operating system.

What each process has?

- Address space
- program code
- data
- program counter
- registers
- process id

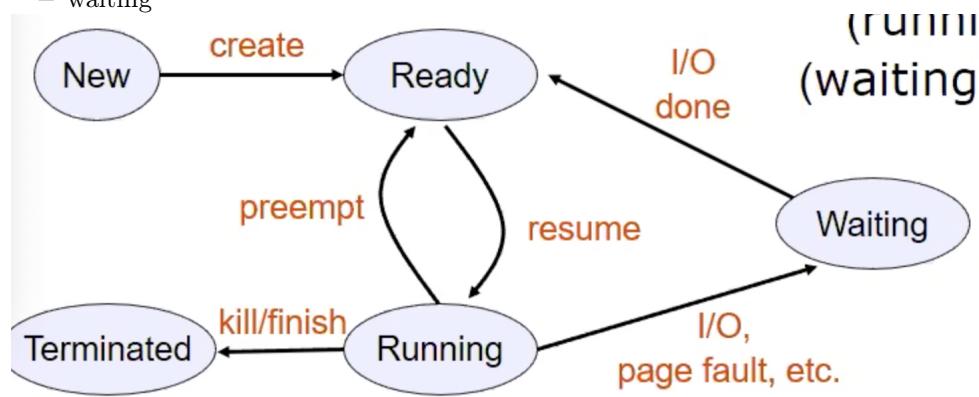


Address space of process.

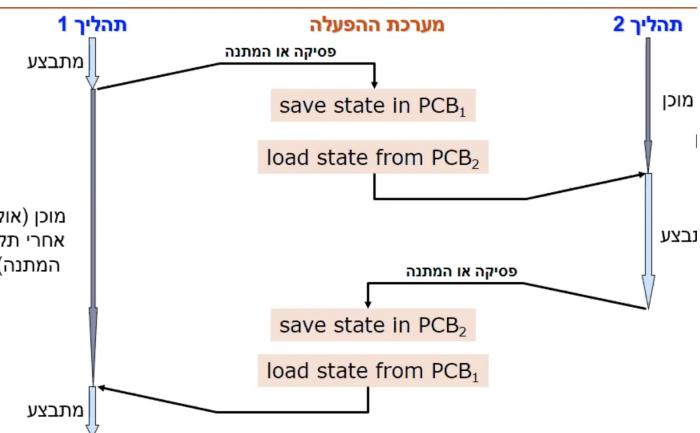
OS (234123) - intro

35

- each process could be in one of the following cases:
 - ready
 - running
 - waiting



-



Context switching:

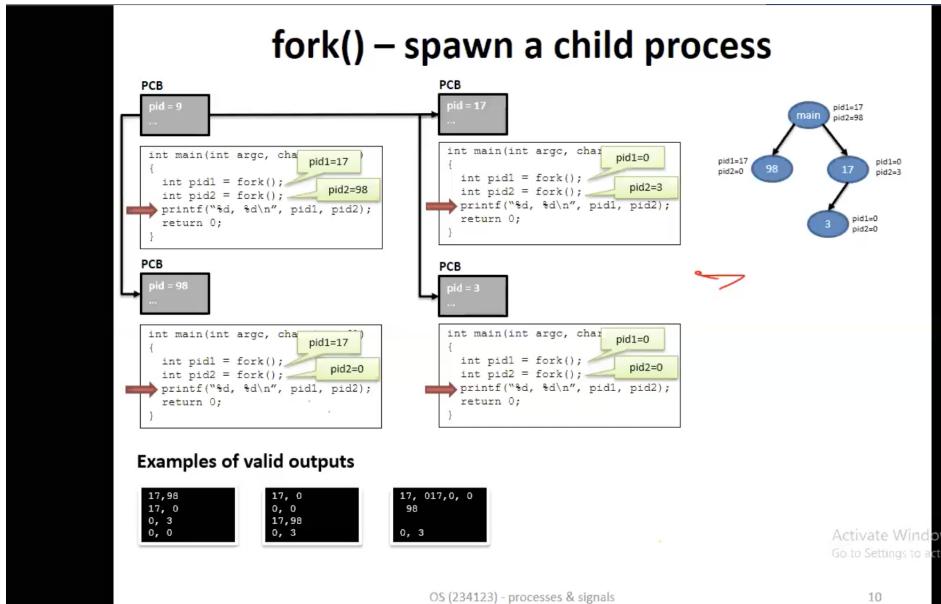
OS (234123) - intro

Creating process.

- Proces (the father) can creatr other process the chikd.
- In general, the father define and inherite it's feautures to it's childs/
 - In linux, the child inherit the fiefs user, and etc..
- The father can wait for it child to finish, or continue running simultaneous

Creating proccess in UNIX: fork():

- create abd initalize PCB.
- Create new address space and initalize it with copy of address space of it's father.
- Initlize kernel resources using it's father resources.
- Put the PCB in the ready queue.
- Now we have two procceses, in which one exists at the same point of executing the same program.
- Both proccesses return from the fork.
 - The childs with value 0,
 - The fatherm with the pid of it's childs



Signals.

`exec*()` - replace current process image.

- To start an entirely new program unlike his father.
 - use the `exec*()` syscall family; for example.
 - `int execv(const char* programPath, char *const argv[]);`
- Semantics
 - stops the execution of the invoking process
 - load the executable 'programPath'
 - Starts 'programPath', with 'argv' as its argv
 - never returns (unless fails)
 - replaces the new process; doesn't create new processess.

* In particular, PID and PPID are the same before/after `exec*()`

Simplistic UNIX shell loop example.

What shell do?

shell read the next command for execution and execute it. E.g cd, ls..

Simplistic UNIX shell loop example

```

int main(int argc, char *argv[])
{
    for(;;) {
        int stat;
        char **argv;
        char *c = readNextCom(&argv);
        int pid = fork();

        if( pid < 0 ) {
            perror("fork failed");
        }
        else if( pid==0 ) { // child
            execv(c, argv);
            perror("execv failed");
        }
        else { // parent
            if( wait(&stat) < 0 )
                perror("wait failed");
            else
                chkStatus(pid,stat);
        }
    }
    return 0;
}

void chkStatus(int pid, int stat)
{
    if( WIFEXITED(stat) ) {
        printf("%d exit code=%d\n",
               pid, WEXITSTATUS(stat));
    }
    else if( WIFSIGNALED(stat) ) {
        // the topic we're going
        // to learn next
        printf("%d died on signal=%d\n",
               pid, WTERMSIG(stat));
    }
    else if
        // a few more options...
}

```

Activate Window
Go to Settings to activ

OS (234123) - processes & signals

12

Code explain: Here we want to create sort of shell. First we check $pid < 0$ i.e it check whether creating the child process succeeded, if not then we print fork failed. Otherwise, the fork indeed succeeded and in this case we load the new code, if not succeeded we print execv failed. Otherwise, the child success then the childs start run new code and after this the child process never return to the same point.

Remark. In windows unlike UNIX we create process as following:

```

CreateProcess{
    fork()
    if(res==0){execv}
}

```

Remark. We pass stat into wait which will contain the data about how child finished.

Silly example for signal:

Simplistic UNIX shell loop example

```

int main(int argc, char *argv[])
{
    for(;;) {
        int stat;
        char **argv;
        char *c = readNextCom(&argv);
        int pid = fork();

        if( pid < 0 ) {
            perror("fork failed");
        }
        else if( pid==0 ) { // child
            execv(c, argv);
            perror("execv failed");
        }
        else { // parent
            if( wait(&stat) < 0 )
                perror("wait failed");
            else
                chkStatus(pid,stat);
        }
    }
    return 0;
}

void chkStatus(int pid, int stat)
{
    if( WIFEXITED(stat) ) {
        printf("%d exit code=%d\n",
               pid, WEXITSTATUS(stat));
    }
    else if( WIFSIGNALED(stat) ) {
        // the topic we're going
        // to learn next
        printf("%d died on signal=%d\n",
               pid, WTERMSIG(stat));
    }
    else if
        // a few more options...
}

```

OS (234123) - processes & signals

Activate Window

Go to Settings to activate

12

Remark. We can see signal which is very confusing. Signal doesn't send signal, it just change handler. I.e we asked operating system when in signal we have SIFGFPE (define as 8) we apply sigfpe_handler. (Handler which care of floating point exception, handler is always in user mode, and not in kernel mode, we get signal by the kernel mode and we as user decide how to take care of it. Each signal has a default handler which OS has it, and we as users send other handler for that signal. We can also use the OS handler but it's not common).

Another silly example

```

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

void sigint_handler(int signum) {
    printf("I'm ignoring your ctrl-c!\n");
}

int main() {
    // when pressing ctrl-c in the shell
    // => SIGINT is delivered to foreground process
    signal(SIGINT,sigint_handler);
    for(;;) { /*endless loop*/ }
    return 0;
}

```

Activate Windows
Go to Settings to activate

When we press on *cntrl-c* we will have keyboard interrupt. Then shell send signal in his process. So operating system doesn't interfere, it just tell shell what the user pressed right now.

Example – ask a running daemon how much work it did thus far

```
int g_count=0;

void do_work() { for(int i=0; i<10000000; i++); }

void sigusr_handler(int signum) {
    printf("Work done so far: %d\n", g_count);
}

int main() {
    signal(SIGUSR1,sigusr_handler);
    for(;;) { do_work(); g_count++; }
    return 0;
}
```

Activate W
Go to Settings

OS (234123) - interrupts & signals

21

Signal vs interrupts. We should examine that who create interrupts is hardware or user by systemcalls (like the user want to retrieve informations from disk) unlike signals which is create by other process or by OS the most. Interrupts pass to OS unlike signal which pass to process for treatment by user.

Signal system calls.

- int kill(pid_t pid, int sig)
 - Allows process to send a signal to another process.
- int sigprocmask(int how, const sigset *set, sigset_t *oldest)
 - it allow OS to ignore some signals with using any handler.
- int sigaction (int signum, const struct sigaction *act, struct sigaction *oldact)
 - The following is replace of the signal function but it's better.
 - In general signal when it get handler afterward it get back to the default handler unlike sigaction which will keep the handler

Signal interact with other system calls.

- ssize_t read(int fd, void *buf, size_t count); (The f
 - what happens if getting signal while reading.
 - The read system call return -1, and it sets the global variable 'errno' to hold EINTR.

Multiprocessing

Definition. Using a several CPU cores (=processors) for running a single job to solve single “problem”.

Motivation: Finish the job faster.

Example:

- Web server which get requests and return applies.
- Replies can be
 - Static (existing file).
 - Dynamic (computed on the fly).

Problem. Assuming we want two multiple two big matrices? In the following we can use one process which use one core and it would take lot of time. Assuming we have 4 cores we can create 4 processes in which each process contain matrix A and B and calculate $\frac{1}{4}$ of the wanted matrix each on its core. But how we bring all what we calculated to only one matrix? it's possible but little complicated for now.

Problem solving: we have one process and we split it into 4 processes and they have all on common the same code, heap..but they have different stack.

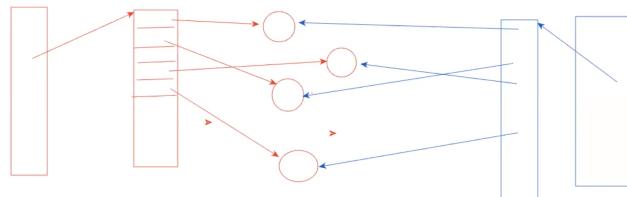
File descriptor.

Thread support is often implemented natively by the programming language runtimes.

- java
- C++

Reminder. We saw that process doesn't share anything with each other unlike threads which share almost everything except registers and stack. Each thread has its own stack. There is no problem the thread 1 access thread 2 stack.

In Posix, “every thing is a file”. Each process has its own process descriptor which points to an array called file descriptor table (FDT). It's an array with 255 entries. Each entry points to a file object. When we open a file, the OS finds the first free entry and creates a file object. For a new process created by fork, it gets a new FDT table the same as its father. I.e. if entry 3 points to hw.txt, then its child FDT entry 3 also points to hw.txt. Furthermore, if parent reads from file, then child reads from where its parent read.



Pipe.

Definition. Pipe is a communication line between child and parent process (vice-versa), or between two different threads. Pipe has two descriptors: one for write and one for read.

A pipe call should return two values reading descriptors and write descriptors (it return 1 if pipe created and -1 otherwise). It get array with two entries, entry 0 denote reading descriptor and entry 1 writing descriptors.

```
#define DO_SYS( syscall ) do { \
    /* safely invoke a system call */ \
    if( (syscall) == -1 ) { \
        perror( #syscall ); \
        exit(1); \
    } \
} while(0)
```

Code Explain: DO_SYS appear in many places, we can do DO_SYS(execv), DO_SYS(pipe) every system call. Then it check if success otherwise it return error and exit. Examine that we can do the same without using do and while but it won't work. The context of using while and do is urgent in this case. Since, using if statement would cause problem, assuming we want to stop the code from running at some point the compiler could consider the else internal neither external if other, i.e it's not full.

```
char g_msg[N]; /* "g" stands for "global" */ \
enum {RD=0, WT=1}; \
int main() { // send g_msg from parent to child \
    int fd[2]; \
    DO_SYS( pipe(fd) ); // establish communication channel \
    if( fork() != 0 ) { // parent writes, so \
        DO_SYS( close(fd[RD]) ); // close read side \
        fill_g_msg(); // don't care how \
        DO_SYS( write(fd[WT], g_msg, N) ); \
        DO_SYS( wait( NULL ) ); // for child to end \
    } \
    else { // child reads, so \
        DO_SYS( close(fd[WT]) ); // close write side \
        DO_SYS( read( fd[RD], g_msg, N ) ); \
    } \
    return 0; \
}
```

Activate Win
Go to Settings to

Explain: In the presented code we first define the parent to be the process who write to child and the child process is the one who read. We can see that synchronizing error is merely possible since a child can read while parent didn't stop writing. Other point is that we close non used descriptor of parent and child. E.g in this case the parent close the reading descriptior and child close writing descriptior.

Remark. By closing descriptior thread we affect other different threads since it's our DFT is identical for all the threads unlike different process.

```

// Assume that filling g_msg requires a lot of
// computational work.
// So we want to use 2 threads, one thread to fill the
// first half of g_msg, and another to fill the second half
void fill_g_msg( void )
{
    pthread_t t1, t2;

    // launch the two threads
    pthread_create(&t1, NULL, thread_fill, "first");
    pthread_create(&t2, NULL, thread_fill, "second");

    // wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
}

```

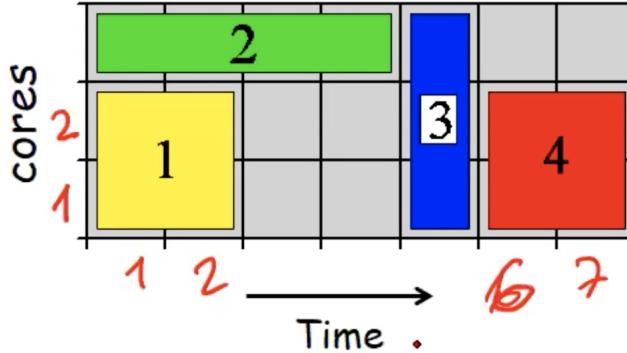
Activate Win
Go to Settings to

How fill_g_msg work? First defining two threads t_1, t_2 then calling `pthread_create`, it get 4 parameters. $\&t_1$ which denote the address in which it going to write it in it the id of the thread. Afterward, there is pointer to stack, `NULL` denote that `pthread` library create stack by itselv (it improve performance often). `thread_fill` is pointer to function and “first” is a `void*` i.e which parameter to pass to a function. So thread t_1 will run function `thread_fill` with parameter “first” and t_2 will run function `thread_fill` with parameter “second”.

Cost of context switch is inherently relative. The OS switch from procces running now with other procces going to run and it has cost.

Problem: CPU faster then main memory. According to Moores law at some point the growth percentage of CPU performance will be slower. Moreover, the gap between DRAM and CPU performance is about 50% the result stem from memory being so slow and one of the reason is that the physical memory is far from CPU so solution is using cashes. And CPU inside it there is a lot of caches and the more the cashe more close to ALU the CPU get faster. We have different type of cashe lvl-3, lvl-2, lvl-1.

Scheduling. Each procces should tell us whats it's size i.e how many cores it need and for much time. So what is scheduling?



As we see in the picture process in yellow will run in 1–2 and will use cores 1-2 in the time. OS can't prevent process from running than less time than process decided. (Some OS don't).

Metrics to evaluate performance. Avg. wait vs .response time - the connection

- Claim:
 - In our context we assume that job runtimes (and thus their average) are a give; they stay the same regardless of the scheduler.
 - Thus, for batch schedulers, the difference between average wait time and average response time of a given schedule is a constant.
 - The constant is the average runtimes of all jobs.

Proof. for each job $i = 1, \dots, n$

□

- let w_i be the wait time of job i .
- let r_i be the runtime of job i .
- let T_i be the response time of job i .

So

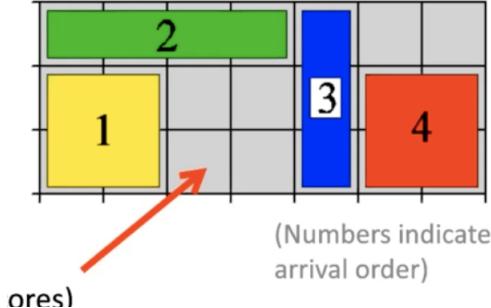
$$\frac{1}{n} \sum_{i=1}^n T_i = \frac{1}{n} \sum_{i=1}^n (r_i + w_i) = \underbrace{\frac{1}{n} \sum_{i=1}^n w_i}_{\text{OS can control}} + \frac{1}{n} \sum_{i=1}^n r_i$$

Scheduling.

FCFS (first come- first served) scheduling.

- Jobs are scheduled by their arrival times.
 - If there are enough cores a newly arriving job starts to run immediately.
 - Otherwise, it waits, sorted by arrival time, until enough cores are freed.
- Pros:
 - Easy to implement (FIFO wait queue).
 - Typically perceived as most fair.
- Cons:
 - Creates fragmentation (unutilized cores).
 - Small/short jobs might wait for a long, long while.

Mechanism. Assuming we have 3 process and 3 cores. As described in the photo.



First we run both process since they start together, first one take 2 cores and second one take 1 core, then both are finished and process 3 (blue process) run with two cores then the fourth process take two cores.

Remark. Each task take number of cores as it asked before running, e.g process 1 asked for two cores also it's running time.

(Backfilling - scheduling).

- The “backfilling” optimization.
 - A short wait job can jump over the head of the wait queue.
 - Provided it doesn’t delay the job @ head of the wait queue.
- Easy algorithm: Whenever jobs start or terminates:
 - Try to start the job @ head of wait queue (FCFS).
 - Then, iterate over the rest waiting jobs (In FCFS order) and try to backfill them.

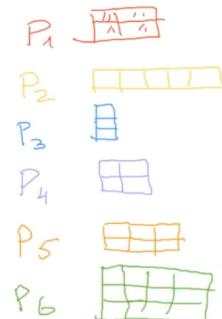
Backfill methods.

- (1) Easy.
- (2) Shortest jobs backfill first - SJBF
- (3) longest jobs backfill first - LJBF
- (4) largest expansion factor - LXF

Remark. In each methods we order process in FIFO order then we look at the fragmentation created and try to exploit then for running other process, after in easy method we get fragmentation we look at all process after then the first once compatible we take it without considering if the first process cover exactly the fragmentation. In SJBF we look at the first processes after and take the shortest process inserted. In LJBF we look at the first processes after and take the longest process inserted. In LXF we take the job with the biggest expansion factor when expansion factor is $1 + \frac{w}{r}$ when w stand for the waiting time of process and r the running time of process i.e we have priority for more smaller process but if there longest one which is waiting a lot of time so LXF insert it.

Remark. The most used methods are easy and SJBP, but remember that any of those methods are optimal. Like if we take a look at the queue we can't change anything, but if we have fragmentation which we are able to cover without harming other processes so it's recommended to do it for efficiency reasons.

Example. We will take example in which easy, SJBF, LXF methods are not the one to use (not optimal). Look at the following processes



In Fifo what will happen is the following,

P ₁	P ₂	P ₃	P ₄	P ₅	P ₆						
0 2 4 6 8 10 12	1 3 5 7 9 11	2 4 6 8 10 12	3 5 7 9 11	4 6 8 10 12	5 7 9 11						
0 1 2 3 4 5 6 7 8 9 10 11	1 2 3 4 5 6 7 8 9 10 11	2 3 4 5 6 7 8 9 10 11	3 4 5 6 7 8 9 10 11	4 5 6 7 8 9 10 11	5 6 7 8 9 10 11						

After backfilling in all methods we will get the same result which is,

P ₁	P ₂	P ₃	P ₄	P ₅	P ₆						
0 2 4 6 8 10 12	1 3 5 7 9 11	2 4 6 8 10 12	3 5 7 9 11	4 6 8 10 12	5 7 9 11						
0 1 2 3 4 5 6 7 8 9 10 11	1 2 3 4 5 6 7 8 9 10 11	2 3 4 5 6 7 8 9 10 11	3 4 5 6 7 8 9 10 11	4 5 6 7 8 9 10 11	5 6 7 8 9 10 11						

The response time is

$$T = 2 + 5 + 6 + 4 + 9 + 13 = 39$$

Other good order is the following

↓											
1	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆					
2	0 2 4 6 8 10 12	1 3 5 7 9 11	2 4 6 8 10 12	3 5 7 9 11	4 6 8 10 12	5 7 9 11					
3	0 1 2 3 4 5 6 7 8 9 10 11	1 2 3 4 5 6 7 8 9 10 11	2 3 4 5 6 7 8 9 10 11	3 4 5 6 7 8 9 10 11	4 5 6 7 8 9 10 11	5 6 7 8 9 10 11					

With

$$T = 2 + 5 + 6 + 8 + 5 + 12 = 38$$

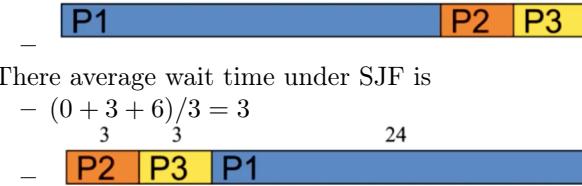
Remark. We are not going to discuss cores optimization, all the optimization we are going to discuss is about time.

SJF.. The following algorithm we don't take the first process arrived as FIFO instead we take the shortest process we have. The main cons of this algo is that a long process "pathetic" should wait a lot since shortest process has higher priority. Which is so bad! But it's use for back filling.

Example. Assuming p_1, p_2, p_3 where their running times are 24, 3, 3. Assuming algorithm FCFS which orders them by their index, and SJF by their running time. Then,

- There average wait time under FCFS is

$$= (0 + 24 + 27)/3 = 17$$



We may examine that JSF is not optimal by luck, indeed this algorithm generate optimal time measurements under some restrictions.

Claim. Under the following conditions,

- Given 1 core system where all jobs are serial.
- If all processes arrive together and their runtimes are known.
- Then: the average wait times of SJF is equal to or smaller than average wait time of any other batch scheduling order S .

Proof. Assuming the scheduling order S is:

$$P(1), P(2), \dots, P(n)$$

If S isn't SJF there exists two processes $P(i), P(j = i + 1)$ s.t

$$R(i) = P(i).\text{runningtime} > P(i), P(j = i + 1).\text{runningtime} = R(j)$$

If we swap the scheduling order of $P(i), P(j)$ under S , then we have increased the wait time of $P(i)$ by $R(j)$ and we have decreased the wait time of $P(i)$ by $R(i)$.

Remark. Focus may, the other processes are not harmed by order.

And since $R(i) > R(j)$ the overall average is reduced, and we do above repeatedly until we reach SJF. \square

What happens if processes won't arrive together? More the once core exists? Is SJF still optimal?

Convoy effect. It's a situation in which a large process arrives then short processes arrive but they can't run since there is no enough cores. SJF tries to solve it but not solving it at most. E.g. assuming a scenario in which we got short process then we let them run after that come longest will let them run but afterward other shortest processes arrive. In algorithms with preemption we can do it.

Reminder. After we saw an algorithm which is batch we are moving to other type of algorithms which do preemption, we remember in first lecture we mentioned that process starts running, maybe it ends, or maybe OS decides to let it wait for cores after it exploits its limited time. And other process starts running, so this operation called preemption in which OS takes the cores from process A and provides it to other process. In general in this type of system, the preemption based on duration for each process we give duration and duration depends on some criterias such as priority. For this duration we call it quantum.

Performance metrics of preemptive schedulers. Respond time is the time in which process ready for running until it finished. And we also said that it's running time + waiting time when waiting time is defined by the time pass from the sec process arrive till it start run. Now for batch algorithm indeed we have equality i.e

$$\text{respondtime} = \text{runningtime} + \text{waitingtime}$$

But for preemption we don't have equality, since in preemption algorithms it's not true, since assuming a process arrive and it waits its waiting time and it starts running then OS tell them "you ran enough" and it changes it with other process and let it wait, so in policy which have preemption we have that

$$\text{respondtime} \geq \text{runningtime} + \text{waitingtime}$$

RR scheduling. This algorithm is a base for other algorithms which are preemption.

- Processes are arranged in a cyclic ready - queue
 - The head process runs, until its quantum is exhausted.
 - The head process is then preempted.
 - The scheduler resumes the next process in the circular list.
 - The time in which we run all processes is called "epoch".
- The epoch relies on number of processes existing, also quantum.

Price of preemption. As motivation of RR we can look at the following example, assuming we have 10 processes each want to run 100 seconds of *CPU* time. If we use FIFO then we will have first process run 100 seconds then next process run 100 seconds then the last process have to wait a lot of time. What will happen in RR? assuming each process has quantum of 1 sec i.e we run first epoch for 10 seconds then other 10 seconds and forth.

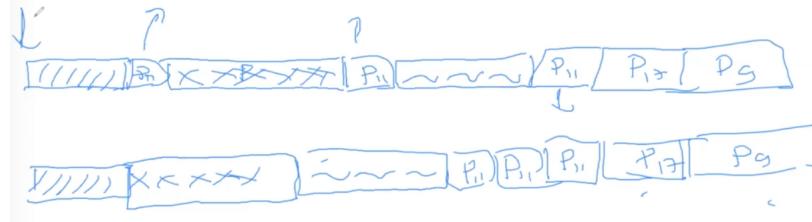
Remark. In a moment we map process to such a core the map will be constant i.e the process will not change cores ever till it finishes.

Batching vs. preemption.

Claim. Let $\text{avgResp}(x)$ be the average response time under algorithm x . Assume a single core system and that all processes arrive together, assume X is preemptive algorithm with context switch price = 0. Then there exists a non-preemptive algorithm y s.t

$$\text{Avgresp}(y^*\text{batch}^*) \leq \text{Avgresp}(x^*\text{preemptive}^*)$$

Proof intuition. Assuming we have scenario in which we do have preemption (p_{11} is scattered)



We change the first timing in the next and it won't harm any of the processes we examine that none of the processes left to p_{11} response time is changed i.e maybe not changed or maybe it get shorter, but none of this processes response time got longer so the response time in the new timing better, and now p_{11} runs with no preemption (i.e no context switch), so we can do that n times, after that all the processes will run with no preemption and no longer response time.

Why we learn preemption?

We saw the SJF is optimal. But it's not good, since long processes could wait too much unlike what we want,

Proof. Let p_k be the last preempted process to finish computing.



Compress all of p_k quanta's to the right (assume time progresses left to right), such that the last quantum remains where it is, and all the rest of the quanta move to the right towards it as the following:



(p_k response time didn't change and the response time of other processes improved or stay the same)

Keep doing that till no preempted processes are found. \square

Claim. Assume

- 1-core system
- All processes arrive together
- Quantum is identical to all processes + no context switch overhead

Then,

- $\text{AverageResponse}(RR) \leq 2 \cdot \text{AverageResponse}(SJF)$

Notations/definitions.

- p_1, \dots, p_n processes
- R_i = run time of p_i
- W_i = wait time of p_i
- $T_i = W_i + R_i$ = response time of p_i
- $\text{delay}(i, k)$ = duration p_i delayed in p_k in RR (how long k waited due to i)
- $\text{delay}(i, i) = R_i$
- Note that $T_k = \sum_{i=1}^n \text{delay}(i, k)$

Proof. For any scheduling algorithm A

$$n \cdot \text{AvgResponse}(A) = \sum_{k=1}^n T_k = \sum_{i=1}^n \sum_{j=1}^n \text{delay}_A(i, j)$$

We can think about it as sum of all matrix elements, so we can write it also as sum of all the diagonals

$$\sum \text{delay}(i, i) + \sum \sum \text{delay}(i, j) + d(j, i)$$

Which mean that

$$= \sum_{i=1}^n R_i + \sum_{1 \leq i < j < n} [delay(i, j) + delay(j, i)]$$

Remark. The first sum in the term ($\sum_{i=1}^n R_i$) is constant no matter which algorithm run, unlike the second term which change according to algorithm.

$$= \sum_{i=1}^n R_i + \sum_{1 \leq i < j < n} [delay(i, j) + delay(j, i)]$$

Assuming we have something as the following,



The SJF algorithm is batch so or p_i run first or p_j so p_j doesn't delay p_i but p_i delay p_j in R_i therefore, $[delay(i, j) + delay(j, i)] = R_i$. (p_i run first since it's shorter, so we take the minimal between shortest running time process between R_i, R_j) so for each pair in the sum we can replace the term above by,

$$= \sum_{i=1}^n R_i + \sum_{1 \leq i < j < n} min(R_i, R_j)$$

All that for $A = SJF$. Now for $A = RR$ we have,



Since we have epoch each process delay the process before it in the same epoch in the next epoch till it finish so at max each process delay other process 2 times * $min(R_i, R_j)$

$$= \sum_{i=1}^n R_i + \sum_{1 \leq i < j < n} 2 \cdot min(R_i, R_j)$$

(Assume p_i is shorter, then P_i delays p_j by R_i , and since it's a perfect RR , p_j delays p_i by R_i). \square

SRTF (Shortest-Remaining-Time-First).

- Assume different jobs may arrive at different times.
- SJF is not optimal.
 - As it's not preemptive, and
 - A short job might arrive while a very long job is running
 - \Rightarrow convoy effect
- SRTF is just like SJG, but
 - Is allowed to use preemption
 - hence, it's "optimal" (assuming a zero context switch cost etc.)
- Whenever a new job arrives or an old job terminates
 - SRTF schedules the job with the shortest remaining time
 - Thereby making an optimal decision

Selfish RR Algorithm.

- New processes wait in a FIFO queue
 - Not yet scheduled
- Older processes scheduled using RR
- New processes are scheduled when
 - No ready-to-run “old” processes exist
 - “Aging” is being applied to new processes (some per-process counter is increased over time); when the counter passes a certain threshold, the “new” process becomes “old” and is transferred to the RR queue
- Fast aging –
 - Algorithm resembles RR
- Slow aging –
 - Algorithm resembles FCFS

We are going to see new algorithm which uses RR algorithm and FIFO algorithm together, we are going to create two queues, the first queue is specified for young processes and queue for old processes when a process arrives and inserted to a young queue then get old until it passes an age, after it passes the age the process turns to the old queue then the scheduling takes only the processes in the old queue and run RR between them i.e. young processes don't participate, if in the old queue no processes exist then we go to the queue of young processes and take the most old then put it into old queue and run it. Now if we have fast aging i.e. a process gets old quickly after it arrives then all our processes are in RR then it's behaviour of RR and if we have slow aging i.e. process arrives and stays at the young queue as long as we have older process and if there is no one we take the most old in the young queue and wait till it is taken so it resembles FIFO so using again behaviour we can behave as FIFO and RR.

Scheduling using priorities.

- Every process is assigned a priority
 - That reflects how “important” it is in that time instance –
 - Can change over time
- Processes with “higher” priority are favored – Scheduled before processes with lower priorities
- The priority concept can also be used for batch scheduler
 - SJF: priority = runtime (smaller => higher)
 - FCFS: priority = arrival time (earlier => higher)

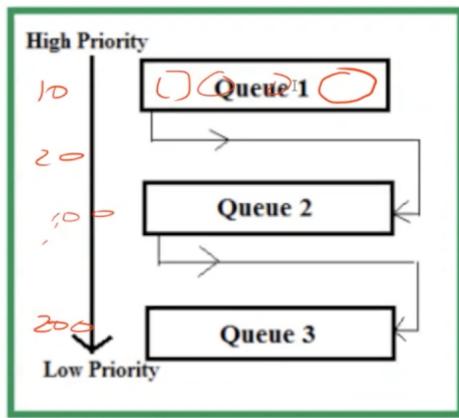
There are processes which have different priority for user, hence we want to know if it's important process, e.g. in selfish RR we can say that aging rate for process is so quick (immediately joining the old queue i.e. RR queue unlike not important process which has its aging rate is so old).

Negative feedback principle. The important thing for OS is to understand which type of process it is, for example, if we look at Matlab, for such a Matlab program what's important is CPU time while running, not all the processes do that, e.g. WORD, the most of time it doesn't exploit CPU time just waits for user input and WORD does something very short and then sleeps, so what's important for this type of programs is quick response. So OS divides the processes into two types, (1) CPU bound (2) I/O bound. It's important to mention that certain process could be both and it's not binary for example when we write code in Matlab it

behaves like WORD (i.e I/O) but when we run it behave as CPU bound. So we as users can tell OS which type of application running, the OS does that. How does it do it? OS does a very simple thing thereby many consequences, assuming OS gave a process 100ms to run, it expects from an I/O bound to run only a few ms and go back to sleep unlike CPU bound which will exploit most of the time we're given, so what happens is that after quanta finish, the OS check how much process exploits from the quanta and determine which type of process it is, but again it's not a binary decision, since the process, for example, can behave 70% CPU bound and 30% I/O bound. Hence, assuming OS deduced it's a CPU bond so the next time it runs, OS will try to give more running time e.g 150ms instead of 100ms but in return, it will wait for more till it gets it.

Multi-level priority queue.

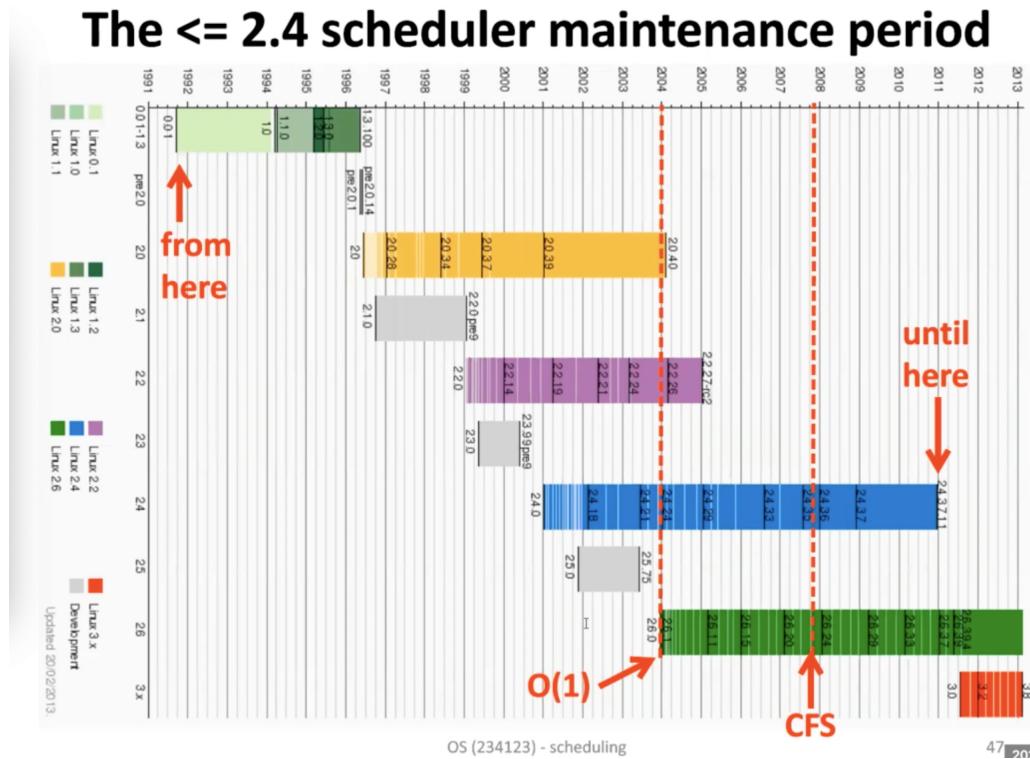
- Several RR queues –
 - Each associated with a priority –
 - Higher-priority queues at the top –
 - Lower-priorityatthebottom
- Processes migrate between queues
 - So they have a dynamic priority –
 - “Important” processes move up (e.g., I/O-bound or “interactive”)
 - “Unimportant” move down (e.g., CPU-bound or “non-interactive”)



An idea developed to define many queues in WINDOWS OS e.g in WINDOWS there is 31 queues and 15 of them are for regular processes, so each thing we insert to one of the 15 queues and for each queue there is specific quanta, and for example in the picture above each procces in queue 2 get 100ms, so what OS do is taking procces just from the highest quanta queue, for example if we take from Queue 1 queue procces (we are expecting him to response quickly) we give him 10ms if after the 10ms the procces vanish then it's fine, otherwise, it gave him 10 sec and didn't vanish so in that case OS put it in lower queue since at first it thought this procces was meant to be I/O bound but turn out that it was CPU bound, but now this procces should wait more time since in the second queue the quanta is higher. Vice versa, OS put also in height queue if it thought such procces was meant to be CPU bound but turn out that it was I/O bound, but now this procces should wait less time since in the higher queue the quanta is higher. The priority of procces appeared in many places, first of all no restriction for the proccesses in

queue 1 of getting the same quanta which E.g more important process may get 10ms and less important process get 5ms i.e for high probability the processes in queue 1 are I/O bound so 10, 15, 20ms won't affect them, but it may affect their ability to reach the queue, e.g if process run 5ms but sometime run with 7ms then a non-important process will level down its queue, and important process still at high queue level since it didn't exploit its quanta. The processes exist in low level which are CPU bound their quanta size is significant assuming we have two processes with high priority (given quanta of 300ms) and other 2 with lower priority (given quanta of 150ms) so we divide our CPU in a way such that one of the high priority processes get $\frac{1}{3}$ of CPU and each of lower priority processes get $\frac{1}{6}$ CPU time. So this is the method to consider processes priorities. (Other concept used for considering priority of processes).

The linux scheduler. In linux there was 3 scheduler, the first one survived to much time, the scheduler we are going to discuss run in linear time,



In 2004 were changed with scheduler run in $O(1)$ but seems to have many consequences due it's low time complexity, and were changed in CFS scheduler.

Definitions: Task.

- For the remainder of this presentations
 - All definitions henceforth relate to the $<=2.4$ scheduler
 - Not necessarily to other (Linux) schedulers
- Within the Linux kernel
 - Every process is called a “task”

- Every thread is called a “task”

Standard POSIX scheduling policies. For each task there is scheduler which run it, E.g linux implements many schedulers and they can live together “peacefully”.

- POSIX dictates that:
 - Each task is associated with one of three scheduling policies
 - * Realtime” policies –
 - (1) SCED_RR (round robin),
 - – (2) SCED_FIFO (first-in, first-out), or
 - * • The default policy
 - – (3) SCED_OTHER » As opposed to the realtime policies, POSIX defines that the meaning of SCED_OTHER is determined by OS
 - Typically employs some multilevel priority queue with the negative feedback loop –
 - Realtime tasks are always favored by the scheduler, if exist
- Focusing on SCED_OTHER
 - Ignoring real time

Epoch.

- (As mentioned earlier for RR)
- Every runnable task gets allocated a quantum
 - CPU time the task is allowed to consume before it's stopped by the OS •
- When all quanta of all runnable tasks becomes zero
 - Start a new epoch, namely –
 - Allocate an additional running time to all tasks
 - * Runnable,or not
- In CFS we are going to see that OS try to devide quamtoms s.t the epoch still constant, i.e we can decide that for every 2 sec give all proccesses to run and assuming we have 17 proccesses get running time due to its priority, but at total the epoch time will be the time we determined in advance, in the new version we are going to disscuss it's not the case, we determine quantom for each proccess according to its priority and (nice which learnt in toturials) any epoch would be derived (not matter).

static/dynamic priorities.

- Task's priority –
 - Every task is associated with an integer
 - Higher value indicates higher priority to run
 - Every task has two different kinds of priorities:
- Task's static priority component
 - Doesn't change with time unless user invoke the nice() system call
 - Homework: browse <https://linux.die.net/man/2/nice> (syscall) & <https://linux.die.net/man/1/nice> (shell utility) –
 - Indirectly determines the max quantum forth is task
- Task's dynamic priority component
 - The (i) remaining time for this task to run and (ii) its current priority
 - Decreases over time (while the task is assigned a CPU and is running)

- When reaches zero, OS forces task to yield the CPU until next epoch
- Reinitialized at the start of every component
- A process has priority, in Linux two types of priorities exist, the first priority is static which is determined by user, and the second priority is dynamic which OS determine for now under many criteria (satisfied or not).

Hz, resolution, ticks.

- Hz – Linux gets a timer interrupt Hz times a second
 - Namely, it gets an interrupt every $1/\text{Hz}$ second
 - ($\text{Hz}=100$ for x86/Linux 2.4)
- Tick – 1. 2. Two meanings (overloaded term):
 - The time that elapses between consecutive timer interrupts
 - * Namely, tick = $1/\text{Hz} = 10$ milliseconds by default for x86 / Linux 2.4
 - The timer interrupt that fires every $1/\text{Hz} = 10$ milliseconds
- Scheduler timing resolution
 - The OS measures the passage of time by counting ticks
 - The units of the dynamic priority component are ‘ticks’

Hz is constant which till the frequency of interrupt arrive to OS (no connection with CPU frequency) i.e if $\text{Hz} = 100$ then OS get 100 interrupt in sec or interrupt arrive every $10ms$. Tick is function or time unit, e.g if we get 100 interrupt in sec so each $10ms$ we apply function which takes care of the interrupt. All the time measurements in old OS versions based on timer interrupts, i.e if we get 100 interrupts in sec our resolution is $10ms$ and OS can’t take care of $4ms$ since the resolution is $10ms$. In more updated OS the mechanism is different.

per-task scheduling info.

- Every task is associated with a task_struct
- Every task_struct has 5 fields used by the scheduler (to be discussed in the next few slides)
 - 1. nice
 - 2. counter
 - 3. processor
 - 4. need_resched
 - 5. mm

Nice is a mechanism which lets user determine the priority of process, a user passes a number from -20 to 19 and this determines the process priority, the most negative number is the higher priority, i.e. process with nice -20 is the most important process running, most of the processes we don't change their nice and we don't use it, so usually the process has nice of 0.

task's nice (kernel vs. user).

- The static component
 - In kernel, initialized to 20 by default
 - Can be changed by nice() and sched_setscheduler()
 - * To be any value between 1 ... 40
 - * • (Homework: browse https://linux.die.net/man/2/sched_setscheduler)
- User's nice (POSIX parameter to the nice system call)

- Between-20...19
- Smaller value indicates higher priority (<0 requires superuser)
- Kernel's nice (field in task_struct used by the scheduler)
 - As noted, between 1...40
 - Higher value indicates higher priority (in contrast to user's nice) •
- Conversion
 - Kernel's nice = 20 - user's nice

task's counter.

- The dynamic component (time to run in epoch, & priority)
 - Upon task creation (integer arithmetic) •

$$\text{child.counter} = \text{parent.counter}/2; \text{parent.counter} = \text{child.counter};$$
 - Upon a new epoch

$$\ast$$
- $$\text{task.counter} = \text{task.counter}/2 + \overbrace{\text{NICE_TO_TICKS}(\text{task.nice})}^{\alpha}$$
- half of prev dynamic + convert_to_ticks(static)
- Decremented upon each tick (task.counter -= 1)
- NICE_TO_TICKS
 - Scales 20 (=DEF_PRIORITY) to number of ticks comprising +50ms
 - Namely, scales 20 to 5+ ticks (recall that each tick is 10 ms by default):

$$\#define \text{NICE_TO_TICKS}(\text{kern_nice}) ((\text{kern_nice})/4 + 1)$$
- Quantum range (without epoch accumulation) is therefore
 - $(1/4+1=)$ 1 tick = 10ms(min)
 - $(20/4 + 1=)$ 6 ticks = 60 ms (default)
 - $(40/4+1=)$ 11 ticks = 110ms(maxforSCED_OTHER)
- In the scheduler runs in constant time there is one application of nice, In the scheduler we are talking about what we do in OS is taking the nice and calculate kernel nice define by 20 – user's nice i.e in the kernel process which took nice –20 will be 40. So what nice affect? “nice” affect the quantum of process which is direct function of nice, we have macro NICE_TO_TICKS which is calculated by the following formula

$$\text{NICE_TO_TICKS}(\text{kern_nice})(\left(\frac{\text{kern_nice}}{4} + 1\right))$$

This happens every epoch end i.e in the end of epoch we go on all processes and calculate for them new quantum using the formula above, the field counter till how much time still for process to run in the current epoch, most of the processes at the end of the epoch their counter is 0. Therefore, the formula

$$\begin{aligned} \text{task.counter} &= \text{task.counter}/2 + \overbrace{\text{NICE_TO_TICKS}(\text{task.nice})}^{\alpha} \\ &= \text{half of prev dynamic} + \text{convert_to_ticks(static)} \end{aligned}$$

The formula above equivalence to NICE_TO_TICKS according to nice, but why we add $\text{task.counter}/2$ in the formula even though the most of processes counter at the end of epoch is 0? The reason is not all the processes finish at the epoch, some of the processes wait to such an event

and for them counter won't have 0 so what we do is taking their counter and devide it by 2 and it happen each epoch end. So assuming procces wait infinite time, what it's going to converge given that β was it's counter before waiting? it's counter is going to be, as following, at first we will have $\frac{1}{2}\beta + \alpha$ after the first epoch it's going to be $\frac{1}{4}\beta + \frac{1}{2}\alpha + \alpha$ afterward $\frac{1}{8}\beta + \frac{1}{4}\alpha + \frac{1}{2}\alpha + \alpha$ so,

$$\lim_{n \rightarrow \infty} \left(\frac{1}{2^n} \beta + \sum_{i=0}^n \frac{1}{2^n} \alpha \right) = 2\alpha$$

i.e it can never reach infinity, at the most 2α and it's important. Now, if a procces with a user nice =19 then in the kernel the nice of it is 1 and we give it 1 time interrupt since $(\frac{1}{4} + 1) = 1$ and if it was 0 then we have $\frac{20}{4} + 1 = 6$ so it will get 6 clock ticks.

processor, need_resched, mm.

- • Task's processor
 - Logical ID of core upon which task has executed most recently
 - If task is currently running
 - * • 'processor' = logical ID of core upon which the task executes now
- Task's need_resched –
 - Boolean checked by kernel just before switching back to user-mode –
 - If set, check if there's a "better" task than the one currently running –
 - If so, context switch to it –
 - Since this flag is checked only for the currently running task
 - * • Can think of it as per-core – rather than per-task – variable •
- Task's mm –
 - A pointer to the task's memory address space

In a procces descriptor we have 3 fields, the first is procces. A procces know on which core runned recently so simply know the id of procces in order to keep running the procces on the same core but in linux it's not neccessarily but we try, the second field, need_resched, OS can do context switch in many places but not where it wants, so there is breakpoint in code where it can do context switch, a common breakpoint is when procces try to quit from kernel mode to user mode, the OS look if context switch needed and if yes then it do it, it know that using the field need_resched in the procces descriptor if it's false then OS do context switch.

Scheduler is comprised of 4 functions.

- (1) goodness(task,cpu) – This function calculate the dynamic priority of a process
 - (a) Given a task and a CPU, return how "desirable" it is for that CPU
 - (b) Compare tasks by this value to decide which will run next on CPU
- (2) schedule() - choose which procces going to run
 - (a) Actual implementation of the scheduling algorithm
 - (b) Uses goodness to decide which task will run next on a given core
- (3) __wake_up_common() – applied on each procces want to join to running
 - (a) Wake up task(s) when waited-for event has happened

- (b) (Event may be, for example, completion of I/O)
- (4) reschedule_idle() - chooses for such a process which CPU recommended to run it
 - (a) Given a task, check whether it can be scheduled on some core
 - (b) Preferably on an idle core, but if there aren't any, by preempting a less desirable task (according to goodness)
 - (c) Used by both __wake_up_common() and by schedule()

```

int goodness(task t, cpu this_cpu){ // bigger = more desirable
    g = t.counter
    if( g == 0 )
        // exhausted quantum, wait until next epoch
        return 0
    if( t.processor == this_cpu )
        // try to avoid migration between cores (why?)
        g += PROC_CHANGE_BONUS
    if( t.mm == this_cpu.current_task.mm )
        // prioritize threads sharing same address space
        // (why?)
        g += SAME_ADDRESS_SPACE_BONUS
    return g
}

```

Code explain for the function goodness. First we can see that task t which is the process in which we are going to calculate the priority, and the cpu we want to run it, first we take the counter which is time remain to process to run and put it into variable g then we do the check if $g == 0$ i.e do the process finish running in the current epoch, if yes then we return 0, otherwise, we look at the field processor of process which contain the id of processor the process run on it last time, if this processor is the wanted processor given as parameter then we give it a credit +1 as Bonus, afterward, we check the memory map of the task and compare it with the current process run on the cpu, if they map to the same memory it follows that on that processor run one of the process thread since only thread share the same memory and are different processes, if that is the case then we add bonus again (we want to give bonus because this is the process thread and run on the same cpu).

Wakeup blocked task(s)

```
void __wake_up_common(wait_queue q) {
    // the blocked tasks residing in q
    // are waiting for an event that has just happened,
    // so try to reschedule all of them
    foreach task t in q
        remove t from q
        add t to ready-to-run list
        reschedule_idle(t)
}
```

Code explain for the function wake_up_common. The following function get a queue of processes and walk on it, for each process in the queue it delete it from the queue and add it to the ready to run list, then apply on it reschedule_idle which check on which cpu recommended to run it.

Try to schedule task on some core

```
void reschedule_idle(task t) {
    next_cpu = NIL
    if( t.processor is idle )           // t's most recent core is idle
        next_cpu = t.processor
    else if( there exists an idle core ) // some other core is idle
        next_cpu = least recently active, currently idle core
    else                                // no core is idle; is t more desirable
                                         // than a currently running task?
        threshold = PREEMPTION THRESHOLD
        foreach c in [all cores]          // find c where t is most desirable
            gdiff = goodness(t,c) - goodness( c.current_task,c )
            if( gdiff > threshold )
                threshold = gdiff
                next_cpu = c
    if( next_cpu != NIL )               // found a core for t
        prev_need = next_cpu.current_task.need_resched
        next_cpu.current_task.need_resched = true
        if( (prev_need == false) && (next_cpu != this_cpu) )
            interrupt next_cpu
}
```

OS (234123) - scheduling

60 202

Code explain for the function reschedule_idle. (responsible to choose a cpu in which the processor going to run). First we have process as parameter, the default is to run it on the processor used last time, and check if it's in idle i.e it free, if yes, then we choose it, otherwise, we walk on all the process and check if any process is idle then we run on it, of not, then all the processors are not idle (not free) in that case we walk on all cores, and for each core, we check the priority on the task we want to run (using goodness function) also we check the goodness on each process run currently on each core, then we search in the loop the maximal *gdiff*, but notice that there is condition, we declare threshold i.e if we find a core with yields a optimal *gdiff* E.g on core 5 but this *gdiff* is neglected (E.g 1) then we don't want to do context switch, because the credit we get won't cover the time take context switch, and it's superfluous, at the end, we reach the if statement which

checks if ($next_cpu \neq NIL$), we reach it in case no core found or cores are doing something more important or we indeed found core doing something less important but didn't pass the threshold, so if we reach it, it mean we are not going to run $task_t$ on any core at this point (maybe afterward) but for now we don't want to access any core and ask him to do context switch, so if $next_cpu == NIL$ then we are finish and the process join the queue ready to run but not going to run for now. Otherwise, $next_cpu \neq NIL$ and there is core on which we want to run the process, i.e or the core not doing anything, or running something which is way less important the running task t , now we can't do context switch immediately, first we go into the task running on the core chosen and let its $need_resched == true$ i.e we marked in it's file descriptor field ($need_resched==true$ which mean it's going to be changed with other process), but what the problem? it's good that we marked by OS should do something with it and when it do something? assuming we have scenario in which the process t were running on core 3 and now is going to run on core 5 so we need core 5 to be in kernel mode and try to go back to user mode, and while going back core 5 will figure out that $need_resched$ is true and try to do context switch, so core 3 should do something in order to let core 5 enter the kernel mode, so the way to do that is interrupts, those type of interrupts, one core can send other core. May focus, not always we need to send interrupts, assuming we task t ran on core 3 last time and now is going to run on core 3 also, so the core is already in kernel mode and no need for interrupts. Othercase, we want to prevent sending interrupts is when core 5 in example above was already marked so in that case we check if the field $task_need$ was already true, if yes then we don't send interrupt from core 3 to core 5 because other core x will send interrupt to run other process on it, not the given process as parameter.

Synchronization.

ABSTRACT. Recently, we have been revealed to the algorithmic part to find the next process to execute. Today, keeping in mind that in each second two processes can switch. So we are talking about Synchronization i.e determining the order of processes running in case required.

Context.

- A set of threads (or processes) utilize shared resource(s) simultaneously
 - For example, threads share the memory –
 - Processes can also share memory (using the right system calls)
 - In this talk we will use the terms thread/process interchangeably
- Need to synchronize also in uni-core setup, not just multicore
 - Userland: a thread can always be preempted in favor of another thread
 - Kernel: interrupts might occur, triggering a different context of execution
 - •
- => Need to learn about synchronization –
 - This is true for any parallel code that shares resources –
 - An OS kernel happens to be such a code. (Many believe students should learn about parallelism from the very first course...)

Example. withdraw money from bank. Assuming we have a bank account and each person can withdraw money from his bank account using a function **withdraw** by accessing a server which calculates the balance after withdrawing afterward we update the server with the new balance. Assuming a scenario we have two persons (two threads in terminology of OS) s.t. the first person withdraws from an account with 50k the first person withdraws a 20k from the account and the second 30k then we have no problem since the balance will be 0 and updating the server. But what happens if two requests happen simultaneously? it can happen in two cores, or two different computers, but now we are focusing in case we have only 1 core with context switch, i.e. the first person withdraws 30k and before updating the server with the new balance a context switch occurs then the other person (thread 2) asks for the balance in the account and gets 50 then withdraws 20 so in that case the two persons withdraw 50k and still have 30k balance.

Example: withdraw money from bank

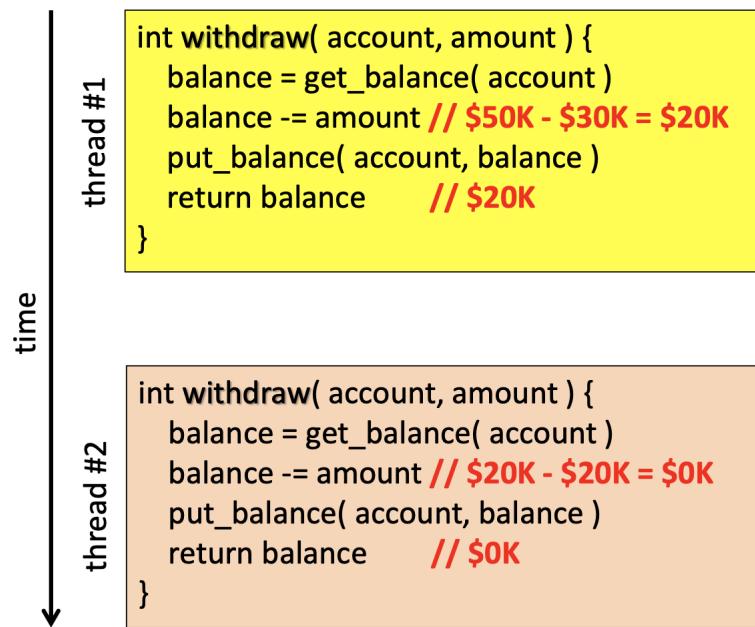
- Assume – This is the code that runs whenever we make a withdrawal –

```
int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount
    put_balance( account, balance )
    return balance
}
```

- Account holds \$50K –
- Account has two owners –
- Both owners make a withdrawal simultaneously from two ATMs
 - 1. One takes out \$20K
 - 2. The other takes out \$30K
- Every operation is done by a different thread on a different core

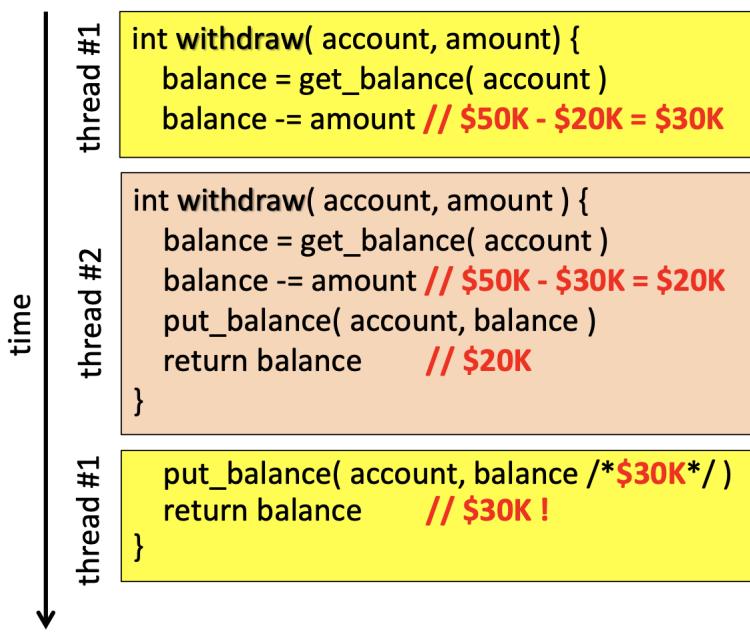
Example: withdraw money from bank

- **Best case scenario from bank's perspective**
 - \$0 in account



Example: withdraw money from bank

- **A better scenario from owners' perspective**
 - \$30K in account...
- **We say that the program suffers from a “race condition”**
 - Outcome is nondeterministic and depends on the timing of uncontrollable, unsynchronized events



Example: too much milk. Assuming we have two persons living together and they want to do coffee with milk, each one goes to the fridge check it and if there is milk then it's fine. Otherwise, one of them goes to grocery and buy milk, so each one runs the following algorithm:

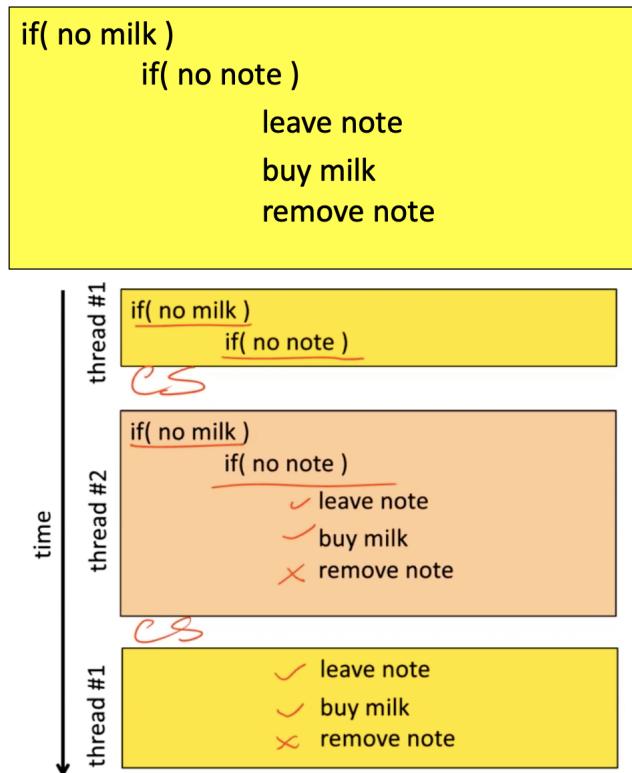
- (1) Go to fridge
- (2) Check if milk exists
- (3) if not
 - (a) go grocery buy milk

At the end of the day we want to stay with 1 milk (no 0, 3 or other number) so what can we do as humans? we can inform the other using a note on a fridge say “I am going to grocery, we run out of milk” then the other person will know that no need to go buying milk.

Milk problem: solution #1? • Leave note on fridge before going to supermarket –

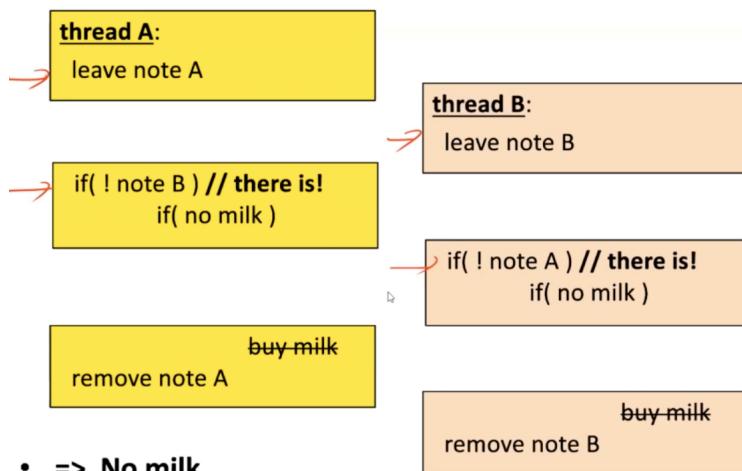
- Probably works for humans
- But for threads...not actually
- assuming we have a yellow thread check there is no milk and no note and before it succeeds to put a note there is context switch
- Then other thread arrives, CHECK IF THERE milk or not, and note, and currently there is no note, so it puts note and buy milk and take the note.

- Now the yellow thread back to run but in this case we do have milk but no note on the fridge so he put note and go buy milk so w/e have 2 milks in fridge.



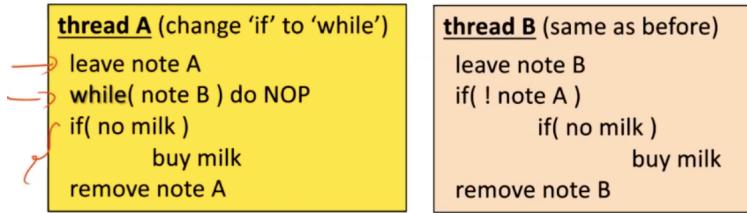
Milk problem: solution #2?

- Leave note, check other note, then check fridge!**



First *A* put the note, and check if there is note of other, if not exists then start taking care of buying the milk i.e check that no milk exist and if yes then buy. and if there is note of *B* then it takes it note. Now we can get a case in which no milk will be assumig thread *A* arrive put its note, then context swich thread *B* arrive and put its note, not *A* see the not of *B* and *B* see the note of *A* so both of them don't buy milk.

Milk problem: solution #3? The solution for the problem above. First *B* do the same as above, but *A* different do other thing, he put note then wait that no note of *B* was putted and if no then he take care of milk, otherwise wait for *B* to finish and check that *B* indeed bought a milk. For this solution there is problem, for example on those two threads there is no uniform factor, in general who is going to buy the milk? if *A* arrive without *B* then *A* buy the milk. if *B* arrive without *A* then *B* buy the milk. If both arrive together then *B* won't buy the milk because he will give up after seeing the note of *A* and won't buy milk, and *A* won't give up he will let *B* to give up about buying the milk then he will buy the milk. Therefore, the factor is different between *A* and *B*. Moreover, it's hard to implement this solution for more than 2 threads e.g 5 threads.

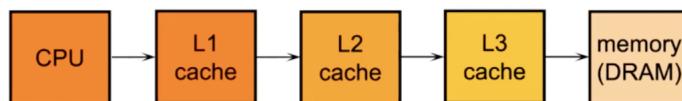


- **In the past, they would have told you**

- That this solution works! ☺ Due to the asymmetry! ☺
- But that
 - Asymmetry complicates things
 - It's "unfair" ('A' works harder: if arrive together, 'A' will buy the milk because 'B' removes its note shortly after)
 - Works for only two threads (what if there are more?)

Reminder: memory tradeoffs.

- Large (dense) memories are slow
- Fast memories are small, expensive and consume more energy
- Goal – give the processor a feeling that it has a memory which is large (dense), fast, consumes low power, and cheap
- Solution – a hierarchy of memories, exploiting locality principle



speed.....	fastest	→	slowest
size.....	smallest	→	biggest
cost.....	highest	→	lowest
power.....	highest	→	lowest

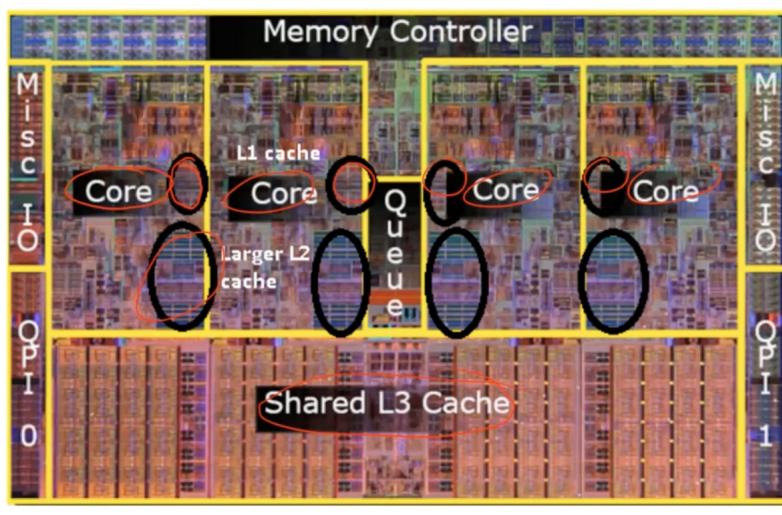
Reminder: typical memory hierarchy.

Intel Core i7-6700 (Skylake), Aug 2015

Base frequency : 3.40 GHz (cycle \approx 1/3 nanosecond)

Turbo frequency: 4.00 GHz (cycle = 1/4 nanosecond)

memory level	size	latency
CPU registers	100s of bytes	< cycle
L1 cache	32 KB (i) + 32 KB (d) (i=instructions; d=data)	4–5 cycles
L2 cache	256 KB	12 cycles
L3 cache ("LLC" = last level cache)	8 MB	42 cycles
Max main memory size (DRAM)	64 GB (for example)	42 cycles + 51 nanosec = 246 cycles (@ 4GHz)



(Nehalem, i7)

Cache coherency (or memory coherency)

Relates to a single memory location

time	event	cache contents for core-1	cache contents for core-2	memory contents for location X
0				1
1	core-1 reads X	1		1
2	core-2 reads X	1	1	1
3	core-1 stores 0 in X	0	1	0

Stale value, different than corresponding memory location and core-1 cache.
 (The next read by core-2 might yield "1".)
 A "coherent memory" doesn't allow such stale reads.

Assuming we have two threads running on different cores and both of them want to access variable x a variable in memory with value 1 now thread 1 access x then it's value load to cache of core 1 with value 1. Now thread 2 try to access x and get 1 and load to cache of core 2 now thread 1 on core 1 change x to 0 so in its cache x is 0 in memory x is 0 ut in cache of core 2 x still one so thread 2 runs on core 2 won't see the change. So there is no consistency, we can solve it but it would harm the performance.

Cache consistency – relates to different locations.

- • Assume
 - Locations A & B are cached by C1 & C2 •
- If store operations are
 - 1. Immediately seen by other cores
 - 2. Cannot be reordered with load ops
- Then it's impossible for both "if" conditions to be true
 - Reaching the "if" statements means either A or B must hold 1
- On modern HW, however, both #1 and #2 can be violated due to performance considerations
 - 1. For performance, stored values can be temporarily placed in a local "
 - 2. For certain implementations, cores might reorder load & store ops if there are no (apparent) dependencies between them
- Should this be allowed?
 - Determined by the memory consistency model

Core C1	Core C2
A = 0; // init	B = 0; // init
...	...
A = 1;	B = 1;
if (B == 0) // I'm alone // => fast path!	if (A == 0) // I'm alone // => fast path!

Consistency models

https://en.wikipedia.org/wiki/Memory_ordering

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y			Y	Y					Y	
Atomic reordered with stores	Y	Y			Y	Y	Y				Y	
Dependent loads reordered	Y											

↑
TSO = total store order model

Back to the milk example, when a thread one to check if other put note, this check is not necessarily true, indeed it can be unupdated value in which the core thread 1 runs on doesn't see it in its cache. (it sees not updated value) so we should update the variable and other cores can see the change.

Towards a solution: critical sections. In our code we want to define some instructions as atomic section instruction. What is atomic? e.g. we are building a house and while building we can enter and check results and update accordingly. Other option is paying a constructor and he is responsible to give a sufficient result at the end, but we can't see what's going on while building, we can only see results after a lot of time. So it's similar to critical section.

- • The heart of the problem –
 - Un同步ized access to data structure with “incomplete” changes
 - In our examples, this was simple vars (“note”, “milk”, “account”) •
 - But could be linked lists, hash tables, a composition of various data structures, etc.
 - (Regardless of whether it's the OS, or a user-level parallel job) –
- Doing only part of the work before another thread interferes
 - If only we could do it atomically
- A group of operations we need to do atomically are called “critical section”

- Atomicity of the critical section would make sure
 - * Other threads don't see partial results
- The critical section can be the same code across all threads
 - * But can also be different

- Example for the same code across threads

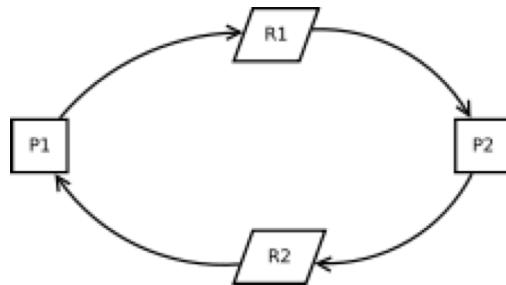
```
int withdraw( account, amount ) {
    balance = get_balance( account )
    balance -= amount
    put_balance( account, balance )
    return balance
}
```

critical section

- Example for different code
 - If one thread increments (“`++`”) and the other decrements (“`-`”) a shared variable
 - Insertion/deletion of an element to/from a linked list

Towards a solution: requirements.

- Mutual exclusion (“mutex”) – (there is only one thread can pass through the mutex)
 - To achieve atomicity
 - * Threads execute an entire critical section one at a time
 - Never simultaneously
 - Thus, a critical section is a “ serialization point”
- Progress - if many threads arrive so one should pass
 - At least one thread gets to do the critical section at any time
- No “Deadlock”
 - Deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus no one ever finishes –
- No “livelock”
 - Same as deadlock, except state changes
 - E.g., 2 people in narrow corridor, trying to be polite by moving aside to let the other pass, ending up swaying from side to side.
- Fairness, which means...
 - No would eventually succeed starvation, namely, a thread that wants to do the critical section, would eventually succeed
 - Nice to have: bounded waiting
 - Nice to have: FIFO



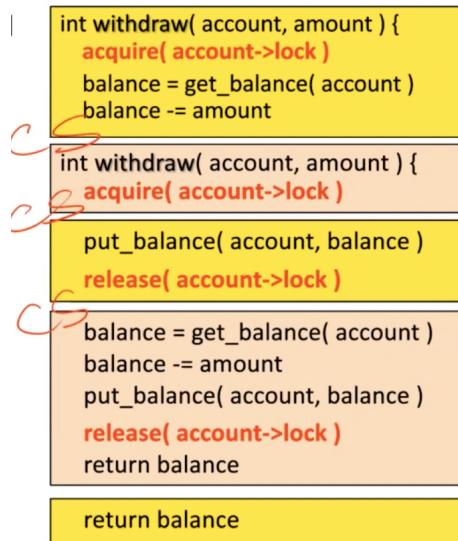
Solution: locks.

- Abstraction that supports two operations
 - acquire(lock)
 - release(lock)
- Semantics
 - – It's a memory fence, and
 - – Only one thread can acquire at any given time, and
 - – Other simultaneous attempts to acquire are forced to wait
 - • Until the lock is released, at which point another thread will get it
 - – Thus, at most one thread holds a lock at any given time •
- Therefore, if a lock “protects” a critical section, meaning –
 - Lock is acquired at the beginning of the critical section
 - – Lock is released at the end of the critical section
- Then atomicity is guaranteed

```

int withdraw( account, amount ) {
    acquire( account->lock )
    balance = get_balance( account )
    balance -= amount
    put_balance( account, balance )
    release( account->lock )
    return balance
}
  
```

- 2 threads make a withdrawal
- What happens when the pink thread tries to acquire?
- Is it okay to return outside the critical section?
 - – Depends
 - – Yes, if you want the balance at time of withdraw(), and you don't care if it changed since
 - – Otherwise, need to acquire lock outside of withdraw(), rather than inside



In the above a yellow thread arrive and acquire lock then access the server and get the balance, calcualate the new balance, afterward, we get context swich, then the other thread runs then try to acquire lock and figure out it's already locked, after sometime, it will run and acquire lock again then update the server with new balance after withdraw.

Implementing locks. A first method is using hardware offered tools in order to implement a lock. Other method is OS provide tools in order to implement a lock.

- When you try to implement a lock –
 - You quickly find out that it involves a critical section.
 - Recursive problem
 - There are 2 ways to overcome the problem
 - 1. Using SW only, no HW support
 - * Possible (was once part of OS courses), but complex, error prone, wasteful, and nowadays completely irrelevant, because...
 - 2. Using HW support (all modern processors provide such support)
 - * • There are special ops that ensure mutual exclusions (later) •
 - * Fairness aspects aren't ensured
 - – That's typically not a problem within the kernel, because
 - » Critical sections are very short
 - » Or else we ensure fairness

FINE-GRAINED SYNCHRONIZATION

```
struct spinlock {
    uint locked; // is the lock held? (0|1)
};
```

```
inline uint
xchg(volatile uint *addr, uint newval) {
    uint result;
    asm volatile("lock; xchgl %0, %1" :
        "+m" (*addr), "=a" (result) :
        "1" (newval) :
        "cc");
    // atomic[ result = *addr,
    //         swap( *addr, newval ) ]
    return result;
}
```

(The ‘while’ is called “spinning”)

```
void
acquire(struct spinlock *lk) {
    disableInterrupts(); //kernel/this core
    while( xchg(&lk->locked, 1) != 0 )
        // xchg() is atomic, and the “lock”
        // adds a “fence” (so read/write ops
        // after acquire aren’t reordered
        // before it)
    ;
}
```

```
void
release(struct spinlock *lk) {
    xchg(&lk->locked, 0);
    enableInterrupts(); //kernel; this core
}
```

code explain. In OS we use a struct which has one field, called `locked` which can have two values 1, 0. 0 mean the lock is open and 1 is locked. Now a function `xchg` is a function get two parameters value and address, and apply a machine instruction called `xchgl` which is atomic instruction. and do the following, assume we have

```
xchgl(int *address, int value){
int res = *address;
*address = value
return res;
}
```

We can notice that there is `volatile`, which means that the code can run on different cores so the access should be forwarded to memory and not cache. Inside `asm volatile` we have `lock`. The reason is OOOE (out of order execution), first assuming we have 5 core operations $op_1, op_2, op_3, op_4, op_5$ defined as following:

```
op1:  $R_1 + R_2 \rightarrow R_3$ 
op2:  $R_4/R_5 \rightarrow R_6$ 
op3:  $R_3 + R_7 \rightarrow R_8$ 
op4:  $R_6 + R_{12} \rightarrow R_{15}$ 
```

Examine that op_1, op_2 are dependent because both of them rely on register R_6 also R_2, R_4 are dependent. What a compiler can do is change order, first the compiler sees that op_2, op_4 are dependent and assuming a deviating instruction (op_2) too much time, so it would affect other instructions so a core can decide to change order e.g op_2, op_1, op_3, op_4 as long as it won't change the final result. The reason we put `lock` for example before op_3 as following

```
op1:  
op2  
lock  
op3  
op4  
op5
```

The lock let the core change order between instruction before lock then execute the instruction we chose to put the lock before. Now look at the acquire function we are going to analyze how it works. First, we do disableinterrupts this function prevent interrupts from coming, so first we take the lock we are trying to lock, in the while we check if its old value was 1 i.e it was locked before by someone else, if yes then we it mean lock didn't succeed, we keep trying to lock (we do busy wait) which won't affect much, the alternative of busy wait is looking at the lock and see that its locked the we do context switch and let other one run. It's recommended when we don't know how much time we need to wait, but remember in OS critical section are very short, so if we do context switch a waste time for context switch and cold start of starting process (it will run slow) so in order to try to solve all those problem we use instead busy wait which solve them easily. Back to disableinterrupts, ignores the coming interrupts. Why? In case we have different cores it's fine, assuming we have a scenario as following, a lock is responsible to prevent from interrupts to access a code which other one can affect it e.g OS changing a data then interrupt arrive and change the same data. Assuming we have system call and start running on core 0 and lock the locker and start update the data structure, now interrupts arrive for core 1 and try to read the same data the system is currently updating, so it locks the locker and this interrupt do busy wait (it can't make progress) and the system call is doing progress and at some point open the locker then the interrupt make a progress, so when we are talking about different cores its fine. What happens if all above run on one core? system call arrives and acquire the locker and start running, interrupts arrive take the system call and put it aside then start running so it checks the locker, it sees that it's locked and do busy wait, now we will have deadlock, because this interrupt do busy wait and who needs to open the locker is the system call which is not running because interrupt took it out from the core and here there is no context switch because we have only one core. So we need to avoid a case of getting interrupts on a same core we have a locked locker. After that we do release but notice that enableinterrupts should occur after releasing so we don't get a case of deadlock.

Kernel spinlock issues.

- In our implementation, interrupts are disabled
 - While the lock is held –
 - Including while the kernel is spinning
 - If we want to allow interrupts
 - while spinning...

```

void
acquire(struct spinlock *lk) {
    disable_interrupts();
    while( xchg(&lk->locked, 1) != 0 ) {
        enable_interrupts();
        disable_interrupts();
    }
}

```

- Why?
 - Responsiveness
 - (Kernel threads don't go to sleep with locks, and they hold locks for very short periods of time)

Spin or block? (applies to kernel & user). OS when acquire a locker or see a locker closed it can know many stuff. E.g no one can do context switch if OS didn't decide to, so when thread acquire the lock, no one can take him out of core only if the thread decided to. So when we see acquired locker we are certain the it will open very very soon, because critical section in OS are very short, and the thread acquired locker no one stop him to stop running so he will finish the critical section as soon he can and give the core control to other thread after opening the locker. In user space this is way forlorn, unfortunately, we can't know for how much time we are closing the locker, so if we look at code, in the code we have section for compute, section for sync, in general section for compute can be very long, so there is no need to do busy waiting, but its more worst, assuming we have section of compute short so we are writing code with short critical section as in OS, also here busy wait is forbidden, since assuming we acquire the locker and start running critical section very short then we do context switch then the thread our of running and other thread start running and want to acquire the same locker then figure out that the locker is already acquired now if he do bust wait, then it's all quantum will keep asking if the locker acquired, this is not deadlock, because after the thread quantum finish and other thread open the locker, but its time wasting since a quanta was wasted. So in general we don't do busy wait in user space and is yes, we should contemplate it deeply.

- Consider the following parallel program canonical structure
 - ```
for(i=0;i<N;i++){
 compute(); // duration is C cycles
 sync();// takes S cycles;
}
```
- Runtime is:
  - $N * (C + S) = N * C + N * S$

- $N * C$  is given by user; as an OS, we can't really do anything about it
- But with S, we may have a choice...
- What happens if  $C \ll S$ ? •
  - \* Sync time dominates –
- If we have a fairly good idea that spinning would end much sooner than a context switch  $\Rightarrow$  should spin, or else runtime would explode
- This is how it's typically done within kernels (spinlocks are used to protect short critical sections)

**Semaphore – concept.** Now we are going to discuss how synchronization is done in user space, we see a locker acquired then a process arrives we put them into a waiting queue also we can't ensure that there is no context switch, and that critical section is shared and more. First of all there is a locker for user space in PTHREADS library. Other concept is called semaphore, a semaphore behaves like a station sells single use tickets, each time we want to pass through the gate we take a single use ticket and pass then less single use tickets remain, if we arrive and there is no single use ticket then we have to wait so in general Semaphore arrives with counter which counts not how many threads can pass the gate

- Proposed by Dijkstra
  - (1968)
- Allows tasks to
  - Coordinate the use of (several instances of) a resource
- Use “flag” to announce
  - “I'm waiting for a resource”, or
  - “Resource has just become available”
- A party who announces that it's waiting for a resource
  - Will get the resource if it's available, or
  - Will otherwise go to sleep –
  - In which case it'll be awakened when the resource becomes available

*Remark.* There is Semaphore also in kernel but it's not related. Here we are talking about the Semaphore of user space.

#### **Semaphore – interface - (Contain two operations).**

- Wait(semaphore)
  - $-value = value - 1$
  - If(  $value \geq 0$  ) // it was  $\geq 1$  before we decreased it
    - \* Task can continue to run (it has been assigned the resource)
  - Else
    - \* Place task in waiting queue
  - A.k.a. P() or probe

If a thread arrives and there is a single use ticket then the thread exploits it. Otherwise, the thread enters a waiting queue. So for each Semaphore there is a counter “value” in case it's positive then it tells how many single use tickets available i.e. how many threads can pass, and if it's negative then it tells how many threads are in the waiting queue.

- **Signal(semaphore) – (opposite behaviour of wait).**

- value  $+= 1$
- – If( value  $<= 0$  ) // it was  $<= -1$  before we increased it
  - Remove one of the tasks from the wait-queue
  - Wake it (make it runnable)
- A.k.a. V() or verhogen

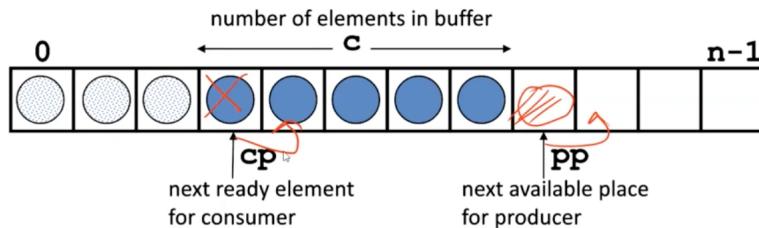
Signal and one single use tickets to the structure by incrementing +1 to the value, if when we arrive “value” was non positive then we have thread waiting, we let one of the waiting thread chosen randomly take that ticket and pass the gate, and increment by one the variable “value”.

*Remark.* Unlike mutex the thread which acquire the lock should open it. In semaphore there is no connection between who do wait and signal. It may be the same thread can do only signal and other only wait.

### Semaphore vs. spinlock.

- Typical granularity
  - Coarse (semaphore) vs. fine (spinlock)
- Spinlock is “more atomic” or “more primitive”
  - Spinlock usually used to implement a semaphore
- • If the maximal ‘value’ of a semaphore is 1
  - Then the semaphore is conceptually similar to a spinlock
  - Sometime called a “binary semaphore”
- But if the maximal ‘value’ is bigger than 1 –
  - Counting a resource
- Don’t have to “wait” in order to “signal”
  - As apposed to locks (must acquire in order to realease)

**Producer/consumer.** We have threads producing elements and other threads consuming those elements. In general we do it using queue, first we have cyclic array of size  $n$  and a pointer which point where we produce element which in general point to empty elements and a pointer for the next real element for consumer



- Problem

- Two threads share an address space –
- The “producer” produces elements
  - \* • (E.g., decodes video frames to be displayed (element = frame))
  - \* –
- The “consumer” consumes elements •
  - (E.g., displays the decoded frames on screen)

- Typically implemented using
  - \* – Cyclic buffer ” (indexed modulo n) a.k.a. “ ring buffer”

```

int c = 0; // shared variable

producer:
while(1)
 →busywait until (c < n);
 buf[pp] = new item;
 pp = (pp+1) mod n;
 c += 1;

consumer:
while(1)
 busywait until (c >= 1);
 consume buf[cp];
 cp = (cp+1) mod n;
 c -= 1;

```

*Basic implementation to producer and consumer.* First in the producer it waits till  $c < n$  i.e it stops when  $c == n$  so we have  $n$  pieces and there is no place to put new one so we need to wait, but when such a place available, we increment in a cyclic wait. In the consumer we do busy wait till  $c >= 1$  i.e there is such element to consume. The problem here is related to the counter, we can see that its in the left code we increment it and in the right side we decrement, so we have confusion. It can be 7 or 8 and we can't know which value it is. So the solution is semaphore.

## Producer/consumer – semaphore solution

```

→ semaphore_t free_space(n);
→ semaphore_t avail_items(0);

producer:
while(1)
 → wait(free_space);
 buf[pp] = new item;
 pp = (pp+1) mod n;
 signal(avail_items);

```

```

consumer:
while(1)
 wait(avail_items);
 consume(buf[cp]);
 cp = (cp+1) mod n;
 signal(free_space);

```

**ABSTRACT.** Earler we discussed how to use semaphoe to implement consumer and producer problem. We will continue to talk about semaphore and how we use it to implement critical section. We will use sempahore to solve reader and writer problem. We don't have problem that several threads read in parallel but when thread write we want to lock the interface so no one can either read or right.

*First approach to solve it.*

```

 int r = 0; // number of concurrent readers
 semaphore_t sRead (1); // defends (serializes) 'r'
 semaphore_t sWrite (1); // writers' mutual exclusion

writer:
 wait(sWrite)
 [write]
 signal(sWrite)

reader:
 wait(sRead)
 [read]
 signal(sRead)

```

In this solution we don't have any synchronization between reader and writers and we can do both in parallel and no one could prevent it. Other problem, is that thread can't read in parallel while we do want them to do it (at most one reader).

```

 int r = 0; // number of concurrent readers
 semaphore_t sRead (1); // defends (serializes) 'r'
 semaphore_t sWrite (1); // writers' mutual exclusion

writer:
 wait(sWrite)
 [write]
 signal(sWrite)

reader:
 wait(sWrite)
 wait(sRead)
 [read]
 signal(sRead)
 signal(sWrite)

```

*Second approach to solve it.*

In this approach we do have synchronization, when a reader arrive he lock other thread from writing while reading so reader and writers are synchronized, but we didn't solve the case that reader can share the interface and read in parallel. The approach is equivalent to regular mutex which is not what we wanted.

*Third approach to solve it.*

```

 int r = 0; // number of concurrent readers
 semaphore_t sRead (1); // defends (serializes) 'r'
 semaphore_t sWrite (1); // writers' mutual exclusion

writer:
 wait(sWrite)
 [write]
 signal(sWrite)

Idea
 - Only 1st reader waits for write semaphore
Works
 - But might starve writers...
 - Think about how to fix
 - Solution:

```

```

reader:
 {
 wait(sRead)
 r += 1
 if(r==1)
 wait(sWrite)
 signal(sRead)
 }
 [Read]
 {
 wait(sRead)
 r -= 1
 if(r==0)
 signal(sWrite)
 signal(sRead)
 }

```

Here writer still the same when he arrives he locks the semaphore for writing the write and open the semaphore. Now for reader, first they lock the semaphore for reading then increment counter of reader by 1 so if the thread was the first one to read then counter will be 1 afterward it checks if the counter value is 1 i.e. this is the first reader so should prevent other threads from writing simultaneously so

if there is active writer in the system it will be blocked and after we lock it we open the semaphore for reader and do reading, assuming other reader come then it increments again the counter and block writer from writing and read so many threads can read in parallel. When a reader finishes its reading then again he locks the semaphore of reading and decrements the counter by 1 and if the counter is 0 then we open the semaphore of writing since it is the last active thread in the system and open the semaphore for reading. In this solution we have a fairness problem, assuming we have a writer arrives then it should wait for all reader to finish and it can take much time, for example each read takes a minute and each 0.5 minute a reading thread arrives and they keep coming so the writers will starve from time.

### Semaphore implementation.

```

struct semaphore_t { int value; wait_queue_t wq; };

void wait(semaphore_t *s) {
 s->value -= 1;
 if(s->value < 0) {
 enqueue(self, &s->wq);
 }
 block(self);
}

void signal(semaphore_t *s) {
 s->value += 1;
 if(s->value <= 0) {
 p = dequeue(&s->wq);
 wakeup(p);
 }
}

```

#### Doesn't work

- The semaphore\_t fields (value and wq) are accessed concurrently
- (For starters...)

First we have a struct of semaphore which contains value which is the counter and waiting queue. When we do wait we decrement the counter by 1 and insert the process to waiting queue then we do block. What is block? Block is equivalent to the following lines of code

self->state = WAITING

schedule()

Basically, we do context switch using schedule, i.e. the process gives up on running since it was blocked. In signal we increment the counter and we check if such a process is in a waiting queue then we do wakeup for him which is equivalent to the following lines of code,

self->state = READY

schedule()

This implementation is problematic because we have shared variables and we do everything without synchronization e.g. value is used in both code and if two threads run in parallel then there is a problem so the result is not obvious. We don't know what the result. So we have two methods to implement it properly, the first is using a hardware instruction as what OS does, and the other method for users we can get support from OS by providing lock\_t which is the spinlock and let us do synchronization so some one can't run the code from different cores.

```

struct semaphore_t { int value; wait_queue_t wq; lock_t l; }

void wait(semaphore_t *s) {
 lock(&s->l);
 s->value -= 1;
 if(s->value < 0) {
 enqueue(self, &s->wq);
 unlock(&s->l);
 block(self);
 }
 else
 unlock(&s->l);
}

void signal(semaphore_t *s) {
 lock(&s->l);
 s->value += 1;
 if(s->value <= 0) {
 p = deque(&s->wq);
 wakeup(p);
 }
 unlock(&s->l);
}

```

**Doesn't work**

- The well-known “problem of lost wakeup” (make sure you can find it)

We can see that in the strucut semaphore there is spinlock. In signal we lock him and execute the code, the we do unlock, so when we do the critical section no one could disturb. Now we have problem thing if we decrement the counter in wait and then we do nothing beacse counter is non-negative then open the lock and finish. But the problem is that we decremt the counter and enter if statement now if we do first enqueue then block the afterward umlock it won;t work becasue block will do context switch so we lock the locker and block the process and won;t back to running so anyone can enter for example to signal and try wake it up because he will be blocked by a locker, as the code as above we have no option rather than doing enqueue then unlock then block to make it work. In this senario assuming we have unlocked the locked by process A and before we do block in other core other process B arrive try to do signal which first increment the counter and take A out from the waiting queue and wake up A, so B try to wake up A but A never want to sleep i.e A wake up so A keep running and try to do block to himself, he do for him waiting as state so we take him out of core by schedule and no one wake him up again because it B already try to wake him up before and he no loner in the waiting queue. So this is very known problem “The lost wake up” and each OS solve it in its way. E.g in Linux it see if A process was already wake up and give up on context switch then continue running it.

**Amdahl's law.** How much can we time with parallelizing and how much synchronization can harm. If we look at a procces exectued by one thread then it take time to do time to finish e.g  $T_1$ . Now assuming this procces executed by  $T_n$  it doesn't mean that the time is minimized by  $n$ becase we have many critical section that we can run in parallel so a part of the time we can't run in parallel and only by 1 thread. So assuming  $s$  is the part which we run by one thread. So the remainder time is  $\frac{1-s}{n}$  we can run with  $n$  threads so  $T_n = T_1s + T_1 \cdot \frac{1-s}{n}$  so at least is  $T_1 \cdot s$  since,  $T_1s + \frac{1-s}{n} \geq T_1s$ . So before we want parallelizing we need to check whether  $\frac{1-s}{n}$  is small.

## DeadLock.

**ABSTRACT.** A non succeeded synchronization, a case in which we are stuck so some processes won't do progress any by that we don't mean that e.g process arrive to locker and can't make progress because after period of time the process will start making a progress when the locker open so its not deadlock. Deadlock, mean a process arrive and won't make progress forever. So more formal a set of threads  $T_1, \dots, T_n$  non empty a deadlock is a subset of thread in which all the process at this set wait for such event and that event depend on other process in the same subset then its a deadlock.

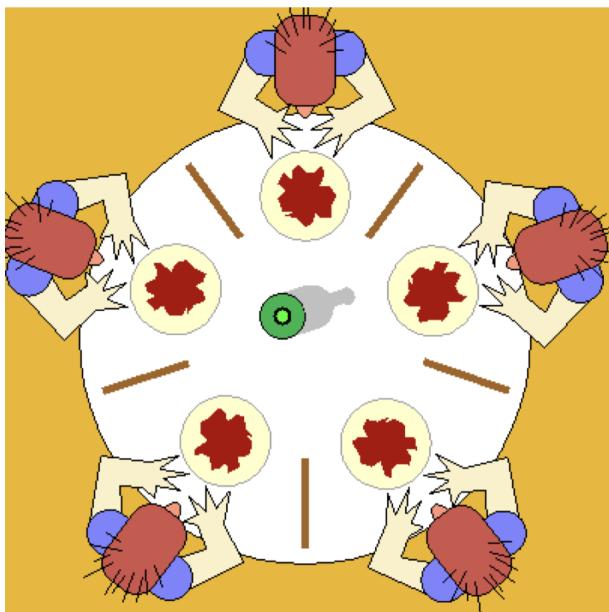
- In the previous lecture
  - We've talked about how to synchronize access to shared resources
- When synchronizing, if we're not careful
  - Our system might enter a state
- The popular formal CS definition of a deadlock
  - “A set of processes is deadlocked if each process in the set is waiting for an event that only a process in the set can cause”
- Typically associated with synchronizing the use of resources
  - Let's revise the definition accordingly
  - A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set
- “The dining philosophers problem” – The canonical example in introductory OS lectures to demonstrate deadlocks

## Dining philosophers – naive solution.

- • Semaphore for each chopstick
  - semaphore \_tchopstick[5] –
  - (What if forks were places in the middle of the table and any philosopher would be able to grab any fork? Would we still need 5 semaphores?)
- Naive (faulty) algorithm
- – philosopher(i):
  - while(1) do...
  - • think for a while
  - • wait( chopstick[i] )
  - • wait( chopstick[(i+1) % 5] )
  - • eat
  - • signal( chopstick[(i+1) % 5] )
  - • signal( chopstick[i] )

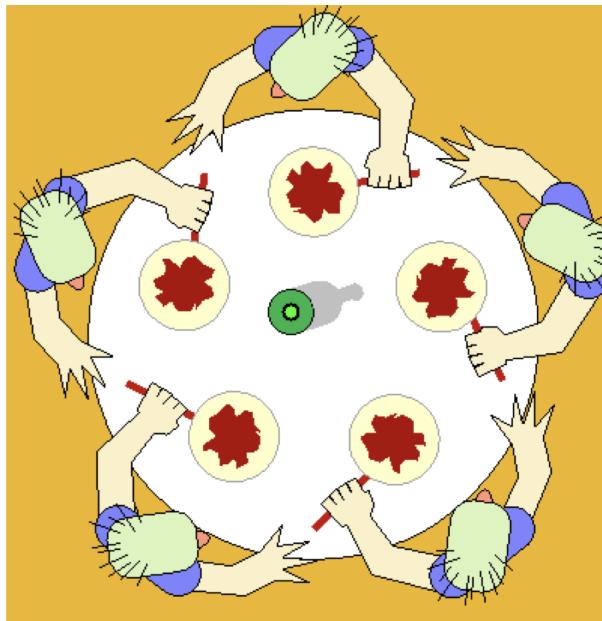
## Example. Dining philosophers – rules.

- Five philosophers are sitting around a round table, each with a bowl of Chinese food in front of him
- Between periods of meditation, they may start eating, whenever they want
- But there are only five chopsticks available, one between every pair of bowls -- and for eating Chinese food, one needs two chopsticks...
- When a philosopher wants to start eating, he must first pick up the chopstick to the left of his bowl; then the right



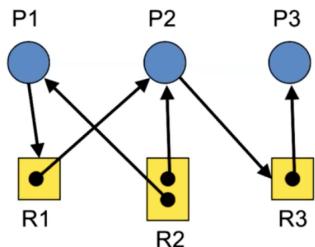
Above on dining table a philosophers each one need to chopstick to eat so we can let it work by letting one get chopstick and other wait till he finish and continue in that way so we have . So this providing code describes the following solution. But what happen if all philosophers take chopsticks and each one wait for other?

- All the philosophers become hungry at the exact same time
- They simultaneously pick up the chopstick to their left
- They then all try to pick up the chopstick to their right
- Only to find that those chopsticks have already been picked up (by the philosopher on their right)
- The philosophers then continue to sit there indefinitely, each holding onto one chopstick, glaring at his neighbor angrily
- They are deadlocked



### Resource allocation graph.

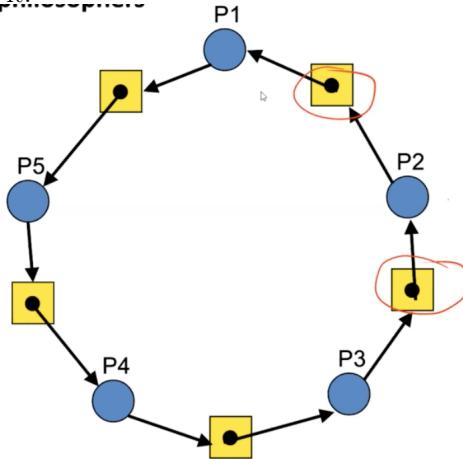
- When considering resource P1 management
  - Convenient to represent system state with a directed graph
- 2 types of nodes
  - Process = round node
  - Resource type = square node
    - \* • Within resource, each instance = a dot
- 2 types of edges
  - Request = edge from process to resource type –
  - Allocation = edge from resource instance to a process



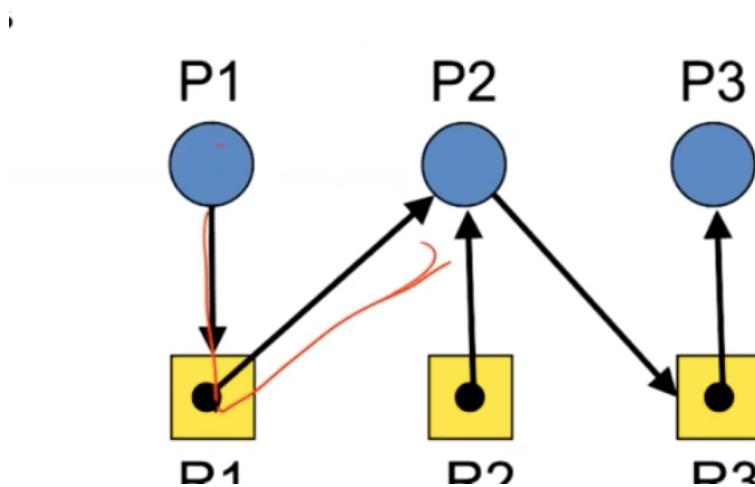
|           |                                              |
|-----------|----------------------------------------------|
| <b>P1</b> | Holds instance of R2.<br>Waits for R1.       |
| <b>P2</b> | Holds instances of R1 & R2.<br>Waits for R3. |
| <b>P3</b> | Holds instance of R3.                        |

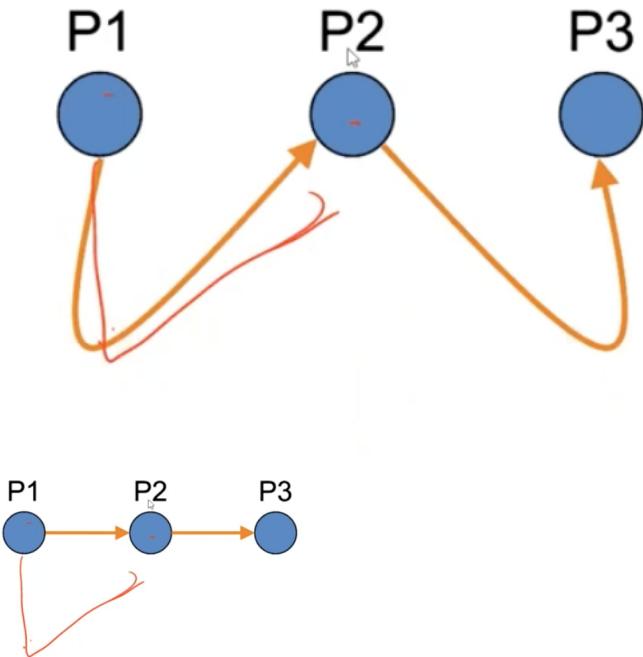
Each yellow box is a resource, for resource can be one copy or many copies for example, if we put a chopsticks at the middle of the table we can think about it as

one resource with 5 copies. A edge start from resource and end in process node denotes that this resource copy in use by that process and edge end in resource starting from a process mean that this process request for this resource and wait for it.

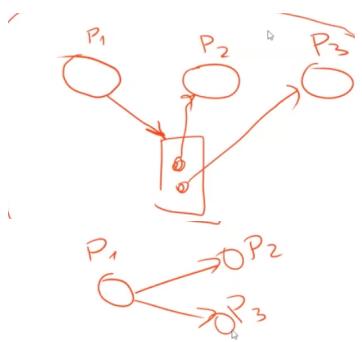


- When there's only one instance per resource type
  - Can simplify graph –
  - By eliminating resources and only marking dependencies between processes

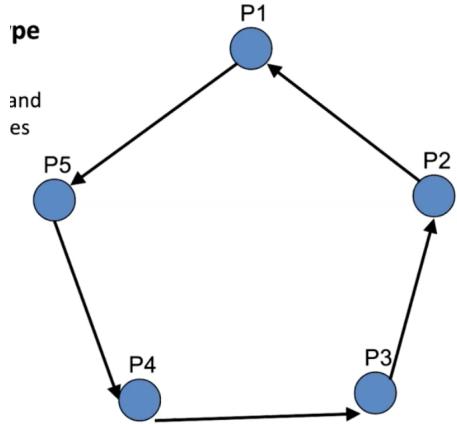




- It won't work when there is resource with many copies.



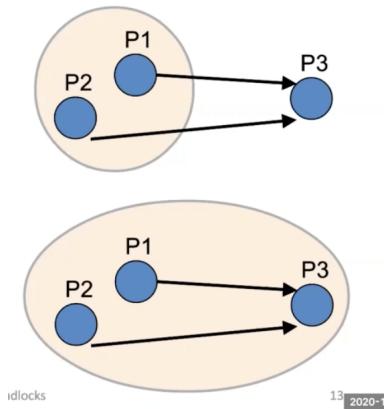
For example above we have more than one resource copy at first we can see that  $p_1$  wait for one of the resource namely the first one finish. When we try to simplify it we get the next picture which won't describe our system since  $p_1$  wait only for one resource i.e or  $p_2$  or  $p_3$  and not both. So back to the philosophers we can simplify since it one resource with no copies.



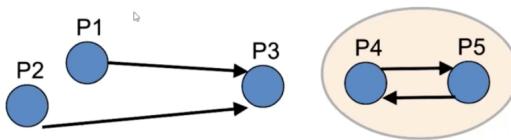
Recall the formal definition of deadlock.

**Definition.** A set of processes is deadlocked if each process in the set is waiting for a resource held by another process in the set

- Why “in the set”? – No deadlock, even though every process in the set is waiting for a resource held by another process: •
- Why “each”? – If including P3, then since P3 isn’t waiting for a resource held by another process => no deadlock:



For example in the set above there is no deadlock, P1 wait for P3 where P3 in the set and P2 wait for P3 but P3 doesn't wait for any.



Here we have deadlock, since we got subset in which each process wait for other to finish.

#### Necessary conditions for deadlock.

- All of these must hold in order for a deadlock to occur

## (1) Mutual exclusion

- Some resource is (i) used by more than one process, but is (ii) exclusively allocated one process at a time (cannot be shared)
- If used by only one process, or can be shared => can't deadlock

## (2) Hold &amp; wait

- Processes may hold one resource and wait for another
- If resources allocated atomically altogether => can't deadlock
- E.g A process holds locker 2 and three and wait for a lock 3 which can never be opened

## (3) Circular wait

- P(i) waits for resource held by P((i+1) % n) // some enumeration
- Otherwise, recursively, there exists one process that need not wait
- Set of processes in which P0 waits for P1 and P1 to P2 ... till Pn waits for P0

## (4) No resource preemption

- If resources held can be released (e.g., after some period of time), then can break circular wait

*Who's responsible?*

- Who is responsible for dealing with deadlocks?
  - Typically, you (the programmer)
  - You need to be aware and do it yourself
  - The OS doesn't do it for you

*2 approaches to cope with deadlocks.*

- (1) Design the system such that it is never allowed to enter into a deadlock situation –
  - (a) Example: this is how we'll acquire locks
- (2) Allow the system to experience deadlock, but put in mechanisms to detect & recover –
  - (a) Example: memory exhaustion => kill random process (or write it to disk and leave it there for a long while)

We as users can implement a code that ensures no deadlock i.e. we are very cautious with locks or anything to prevent. Otherwise, implement efficient code and once in a while can enter a deadlock then we figure it out and reset the process. In 99% we will use code that ensures no deadlock.

**Prevention = violate 1 of the 4 conditions.**

- All 4 conditions must hold for deadlock to occur –
  - So violating even one eliminates possibility of deadlock

No "hold and wait" (#2). We want to prevent a case and hold many references and wait for other references, so we can solve it by giving him all resources before running, but it will harm parallelizing. A modification of that is that give up on all resources peocess hold then after the process ask for all mutex together after. In general this is not easy to implement and most OSes don't support it.

- Instead of acquiring resources one by one

- Each process requests all resources it'll need at the outset
- System can then either provide all resources immediately –
- Or block process until all requested resources are available
- Con
  - Processes will hold on to their resources for more time than they actually need them –
  - limits concurrency and hence performance
- Refinement (allow compute phases with different resources) –
  - Before a process issues a new (atomic) request for resources –
  - It must release all resources it currently holds] • (After bringing system to consistent state, of course) –
  - Risking the resources will be allocated to other processes

*No “no resource preemption” (#4).* Under some circumstances, for some resources

- Can choose a victim process and release all its resources – For example, if there isn't enough memory, can

- Either write the victim's state to disk and release all its memory
- Or kill it

**No “mutual exclusion” (#1) ].**

- Resource is sharable, for example...
- Can implement some data structures (e.g., linked list)
  - Multiple threads can concurrently use of the data structure
  - Without using any form offered explicit synchronization
  - No spinlocks, no semaphores, etc. •
- How? –
  - Using only HW-supported atomic operations (such as test-and-set)
  -
- These algorithms are (also) called “lock free” • Overloaded term – Not to be confused with another definition of “lock free” (= “some thread always makes progress”)
- Mature field
  - Books on how to do it (proving implementations are correct)
  - Existing libraries to use without being exposed to the complexities
  - Composition of operations is problematic

*No “circular wait” (#3).*

- Probably the most usable / practical / flexible way to prevent deadlocks
- How it's done
  - All resources are numbered in one sequence  $\text{Ord}(\text{printer})=1, \text{Ord}(\text{scanner})=2, \text{Ord}(\text{lock\_x})=3, \text{Ord}(\text{lock\_y})=4$
  - E.g if process using resources of printer and lockx and need more resource then he can ask only for resources with higher order for example lock\_y.
  - Processes must request resources in increasing  $\text{Ord}()$  order –
  - Namely, a process holding some resources can only request additional resources that have strictly higher numbers –
  - A process that wishes to acquire a resource that has a lower order

- \* Must first release all the resources it currently holds

*Proof.* Proof that it works □

- – Assume by contradiction that there exists a cycle
- – Without loss of generality, further assume that •
  - $P(i)$  waits for  $P((i+1) \% n)$  /\*  $i = 0, 1, \dots, n-1$
  - \*/ – Let  $M(i)$  be
    - \* • The maximal  $Ord()$  amongst the resources that  $P(i)$  holds
    - \* – Thus, since
    - \* • Each  $P(i)$  acquires resources in order, and
    - \* •  $P(i)$  waits for a resource held by  $P((i+1) \% n)$  –
  - Then
  - $M(i) < M((i+1) \% n)$
  - $\Rightarrow M(0) < M(1) < M(2) < \dots < M(n) < M(0)$
  - $\Rightarrow M(0) < M(0) \Rightarrow$  contradiction

*Proof.* More comfortable way to proof is that a process can hold many resources and each resource got order e,g

$$ord(printer) = 1, ord(scanner) = 2, ord(lock\_1) = 3, ord(lock\_2) = 4$$

For example if resource hold the printer and  $lock\_1$  and if the process request for addition resorce then this process could ask for  $lock\_2$  but not scanner sice  $ord(lock\_1) > ord(scanner)$ . Assuming in this method we got circular waiting

$$p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_n$$

For example  $p_0 \rightarrow p_1$  denote that  $p_0$  wait for resource  $R$  which  $p_1$  is using. Denote,

$$ord(p_i) = \max\{ord(R_i) | \text{for all } i \text{ resources under use of } p_i\}$$

So

$$ord(p_0) < ord(R)$$

$$ord(R) \leq ord(p_1)$$

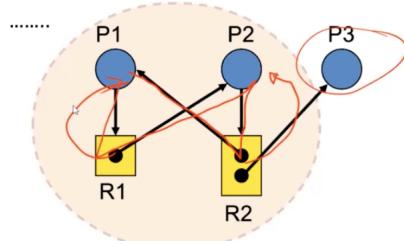
We can continue till we get that

$$ord(p_0) < ord(R) \leq ord(p_1) < ord(p_2) \dots < ord(p_n) < ord(p_0)$$

So we got contradiction. □

### Deadlock detection.

- If there's only one instance of each resource type
  - Search for a cycle in the (simplified) resource allocation graph •
  - Found  $\iff$  deadlock
- In the general case, which allows multiple instances per type
  - Necessary conditions for deadlock != sufficient conditions for deadlock –
  - A graph can have a cycle while the system is not deadlocked
- Can nevertheless detect deadlocks in general case
  - But algorithm outside of our scope



- Circular wait doesn't mean sure deadlock. There are cases like above in which circular exists but no deadlock. Namely, when there is duplicate resource. For example above  $p_3$  finishes and free his duplicate of  $p_2$  to  $p_2$  then can give it to  $p_2$  and now  $p_2$  got all wanted resources then he can run and when ever he finishes he free  $R_1$  to  $p_1$  and  $p_1$  can finish.

#### Recovery from deadlock after detection.

- After a deadlock has been detected (previous slide)
  - Need to somehow recover
- This could be done by terminating some of the processes
  - Until deadlock is resolved
  - Sometimes make sense, sometimes doesn't
- Or it could be done by preempting resources
  - Of deadlocked processes
  - Sometimes make sense, sometimes doesn't
- Finding a minimal ("optimal") set of processes to terminate or resources to preempt is a hard problem. A approach to solve is find a minimal subset of processes we need to remove in order to not get deadlock this is NP problem and we can't solve it in accurate. The second approach is killing randomly till we don't get deadlock.

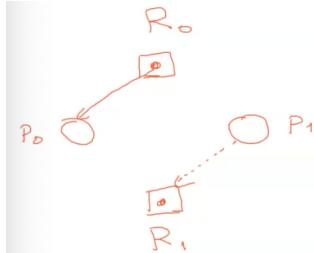
#### Banker's algorithm for deadlock avoidance.

- Rules
  - n processes
  - k resource types (each type may have 1 or more instances)
  - Upon initialization, each process declares the maximal number of resource-instances it'll need for each resource type
  - While running, OS maintains how many resources are currently used by each process – And how many resource instances per type are currently free
- Upon process resource allocation request –
  - OS will allocate iff allocation isn't "dangerous", namely
  - If it knows for a fact that it'll be able to avoid deadlock in the future
  - Otherwise, the process will be blocked until a better ("safer") time
  - Algorithm is thus said to be conservative, as there's a possibility for no deadlock even if allocation is made, but OS doesn't take that chance
- Upon process termination (assume: process do terminate)

- Process releases all its resources

*Explain. The system manage resources and worry that no deadlock will be while running, in a system in which every process has one resource we can shot that existsance of deadlock could occur  $\iff$  there is circular waiting of processes by definition of deadlock. What happen in which there is resource with dublicate this condition is not neccessary for detecting deadlock. e.g the example above. Now what happen if everything is not dependent on us assuming we have two processes running the first one open two files first we lock the first file then the second and there is other procces which open them and lock them in different order, so if there is shared resources between two resources we can ensure that there is no deadlock. In this case we can do system which prevent deadlocks, the system know how much resources there are, each procces ask for max dublicate of each resource, and while running he can ask for more resources.*

**Example.** Assuming we have p0, p1 and resources R0, R1 p0 start running and ask for R0 and get it then P1 dtart running and ask for R1 so the question is to give or not?



We will see the answer soon.

**Example.** Assuming we have a bank which has 100k\$ in all. Invesotor arrive to bank and ask for 70k\$ and after two years he will return 200k\$. But for know he asked only for 50k\$ and after month he will take the other 50k\$ then he went. After week other invesor come and ask for 70k\$ and promise to return back 300k\$. And for now he know there is no 70k\$ in the bank so he ask for 40k\$ only. So did we behaive well? Absoulutely no, we can't give any of the investor money when one of them return back to the baank. This describe the crisis in 2008 at USA. So this exmple, is the same as the example earlier, which its answer is not to give for obvious now since we dre doing something risky. Sometimes dealock can't occur for example if the first investor take debt of 70k\$ and ask for 50k\$ for beggining then other invester come and ask for 70k but for now ask for 30k\$ so in the bank will still 20k\$ so the first investor come and take them and after the while return the money for the bank with benefits hence second inversot can debt money now and we won't get deadlock. So the Banker Algorithm solve this type of case to avoid deadlocks.

### Banker Algorithim.

- Example Banker's algorithm
  - Uses the notation of “safe state”
  - • A state whereby we're sure that all processes can be executed, in a certain order, one after the other, such that each will obtain all the resources it needs to complete its execution

- By ensuring such a sequence exists after each allocation => avoid deadlock
- $O(n^2)$ 
  - Even though number of possible orders is  $n!$
  - Since resources increase monotonically as processes terminate
  - As long as it's possible to execute any set of processes
    - \* Execution order not important
    - \* (There is never any need to backtrack and try another order)
- Banker's data structure
  - $\text{max}[p] = (m\_1, m\_2, \dots, m\_k)$  = max resource requirements for process p
  - $\text{cur}[p] = (c\_1, c\_2, \dots, c\_k)$  = current resource allocation for process p
  - $R = (r\_1, r\_2, \dots, r\_k)$  = the current resource request (for process p)
  - available =  $(a\_1, a\_2, \dots, a\_k)$  = currently available (free) resources (global)
- Example
  - –  $\text{max}[p] = (3,0,1)$ ,  $\text{cur}[p] = (3,0,0)$
  - – Note that  $\text{max}[p] \geq \text{cur}[p]$  always hold

5 processes  $P_0$  through  $P_4$ ;

3 resource types:

$A$  (10 instances),  $B$  (5 instances), and  $C$  (7 instances)

Snapshot at time  $T_0$ :

|       | <u>Allocation</u> | <u>Max</u> | <u>Available</u> |
|-------|-------------------|------------|------------------|
|       | $A\ B\ C$         | $A\ B\ C$  | $A\ B\ C$        |
| $P_0$ | 0 1 0             | 7 5 3      | 3 3 2            |
| $P_1$ | 2 0 0             | 3 2 2      |                  |
| $P_2$ | 3 0 2             | 9 0 2      |                  |
| $P_3$ | 2 1 1             | 2 2 2      |                  |
| $P_4$ | 0 0 2             | 4 3 3      |                  |

*Algorithm Run example.*

Here we have three resources each of the resource has a instance e.g A got 10 instances and B got 5 instances and C got 7 instances. We snapshot the allocation in second  $T_0$  for example  $p_0$  process own 0 1 0 i.e 0 instance of A and 1 instance of B and 0 instance of C the same for  $P_1, P_2, P_3, P_4$ . The second column tell us the max resources a process can use while running e.g in the example above it was 100k\$ for investor one. The third column tell us how much of the resources available we can

see 3 3 2 and its normal since we can see in alloctaion that  $p_1$  use 2 instances of  $A$  and  $p_2$  use 3 and  $p_3$  use 2 and  $p_4$  use 0 so 7 in total in use at  $T_0$ . 3 3 2 are avaialble In order to check that we look at the need column. which we get by substracting each cooordinate from the max vector by the allocation vector, so we get the following result,

■ The content of the matrix **Need** is defined to be **Max – Allocation**

| <u>Need</u> |       |
|-------------|-------|
|             | A B C |
| $P_0$       | 7 4 3 |
| $P_1$       | 1 2 2 |
| $P_2$       | 6 0 0 |
| $P_3$       | 0 1 1 |
| $P_4$       | 4 3 1 |

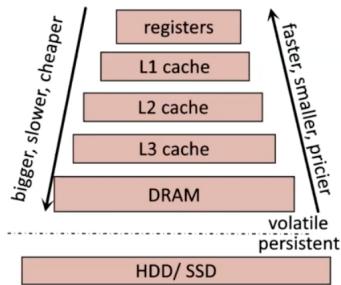
Now we get the need column and we have available of 3 3 2 we walk on each cooordinate in the need column and check if one of them is less the the availabe so we can give him the resources instances we can see that  $p_1$  need  $(1, 1, 2) < (3, 3, 2)$  in each coordinate  $available_i, need_i$ ; therefore, we give him the available which is 3 3 2 after it finish we free all the resources and now the new available is  $3 \ 3 \ 2 + allocated(2, 0, 0) = 5 \ 3 \ 2$  now we continue in the need table to the first proccess implies that his need is less than the new available, if no such a proccess exists then there is a deadlock all the table means that our system is not safe. Otherwise, we again find the new available and continue. For illustration purpose we will continue, now our new available is 5 3 2 we check if  $p_2$  need is less (its not) then we continue to  $p_3$  which is 0, 1, 1 i.e less therefore we finish with him and our new available us 5, 3, 2 + 0, 1, 1 = 6, 4, 3 then we continue to  $p_4$  we have 4, 3, 1 now our new available is 6, 4, 3, +4, 3, 1 = 10, 7, 4 now we continue to the head of the table i.e  $p_0$  which needs 7, 4, 3 less than 10, 7, 4, so we use him and now we have 17, 11, 7 qw continue to  $p_1$ (already finished) then  $p_2$  witht 6, 0, 0 < 17, 11, 7 we give him wanted and our new available is 23, 11, 7 so we are finnished and the system satisfies the safety criteria.

## Virtual memory

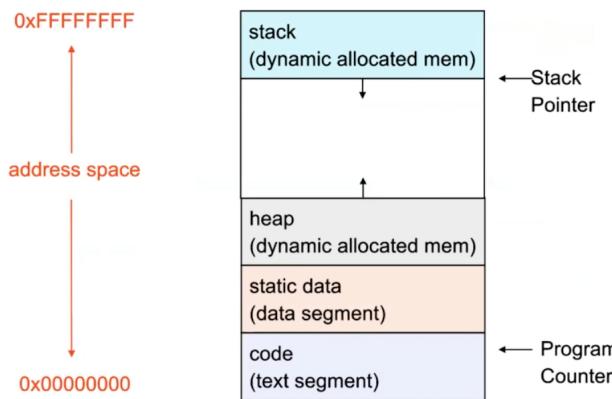
**Virtual memory concepts.** 1–32cores@1–5GHz

- Memory hierarchy
  - L1 = O(64 KB) per core
    - \* • Latency = O(2 cycles) ≈ O(1 nanosecond)
  - L2 = O(256 KB) per core
    - Latency = O(10 cycles) ≈ O(5 nanoseconds)
  - L3 = LLC = O(8 MB) shared •

- Latency =  $O(40 \text{ cycles}) \approx O(20 \text{ nanoseconds})$
- (LLC is “last-level cache”)
- DRAM =  $O(1 - 1000 \text{ GBs})$  shared
  - • Latency =  $O(200 \text{ cycles}) \approx O(100 \text{ nanoseconds})$
- Persistent storage = HDD / SSD
  - Latency = 10s to 100s of usec (SSD) to a few ms (HDD)



We can see here hierarchy of a memories devices we have register which can store order of 1000k bytes and the access to it take 1 machine cycle. Then L1 cashe which is 64kb and need to cycles for access. Then L2, L2 then the DRAM which is very very huge but low for access 200 cycles which is pretty bad. All those devices are volatile i.e if we shut down the PC al the data they is gone but they are quick. Unlike system stoarage which store data for all time, e.g solid disk, hard disk. Our focus is to know how memory managed can be found on all of those devices. For example OS sometimes can take data from caches and put it temporarily into DISK.



We already saw that OS task is to simulate computer. I.e when procces start running he got virtual computer because he has not to worry for other proccesses running and he is the only one running in the system and no one can do for him anything. When procces look at his procceses he see a contagious block of memory. For example if define `int x, int y, int[10] arr`, then those thing are one after other in memory. So we want to see how OS give illusion for procces to think that he got a contiguous memory and enormous amount of it.

### Virtual memory (vmem) – motivation (Read them all to understand).

- Per-program Illusion of contiguous memory
  - Programmers need not worry about where data is placed exactly
  - They use the simplistic, ideal address space from the previous slides
- Isolation between processes
  - Processes can concurrently run on the same processor
  - Yet vmem prevents them from accessing the memory of one another – (Although still allows for convenient sharing when required)
- Dynamic growth –
  - Can add memory to process's heap/stack at runtime, as needed
- Illusion of large memory => memory overcommitment
  - DRAM is costly parts & often the bottleneck resource – Vmem size can be bigger than physical memory size
  - Allows for memory “overcommitment” • Sum of vmem spaces (across all processes) can be  $\geq$  physical • Access control
  - Decide if individual memory chunks can be read / written / executed

### Virtual memory – terminology.

- Virtual address (VA)
  - Used by the program/programmer
  - “Ideal” = contagious & as big as we'd like
- Physical address (PA) – The real, underlying physical memory address
  - Completely abstracted away, by OS+HW
- Memory (virtual & physical) is divided into fixed size blocks
  - “page” = Page = chunk of contagious data (in virtual or physical space)
  - “Frame” = physical memory exactly big enough to hold one page
  - $|page| = |frame| = 2^k$  (bytes)
  - Typically,  $k = 12$ , namely a page (and frame) size is 4KB
- Pages & frames are always – Both in physical and virtual memory spaces: 0, 4KB, 8KB, 12KB, 16KB, ...

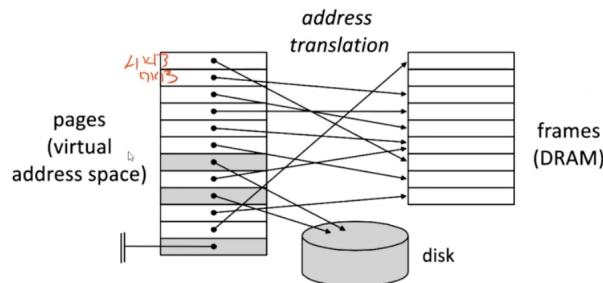


*Explain. Assuming for each process we want to allocate a virtual memory space. A simple option is that for each data address we store where he saved the physical memory, e.g. if process access a virtual address 30 we do a calculation and we figure out where this address in RAM (i.e. physical address) so each address we can calculate where its in physical memory, so we allow each process to use any virtual memory he wants then hardware does the calculation and access a different address in physical memory. So how are we going to do it? We first devide for each process a table which map virtual addresses to physical addresses. Assuming our physical*

address is 32bit - $4B$  and number of virtual addresses is  $4GB$  so for each virtual address we need to store physical address so in total we will stay with  $4GB \times 4B = 16GB$  for each process which is enormous and not efficient at all. So what we do? We divide our physical memory and virtual to chunks of  $4KB$  which in virtual memory is called page and in frame in physical memory. So the map tell us in which frame every page, then we do map between chunk from virtual memory to chunk in physical memory. Now we will have  $\frac{4GB}{4KB} = 1M$  of chunks which want to map therefore, we need  $1M$ .  $4B$  =  $4MB$  so now it more efficient but still not good.

*For each entry*

### Per-process virtual memory simplistic illustration.



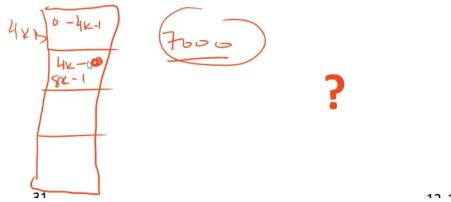
- (1) Illusion of contiguity & of having more physical memory
- (2) Actual physical location (of program and its data) unimportant
- (3) Dynamic growth, isolation, & sharing are easy to obtain
- (4) On-demand allocation

### Virtual memory – basic idea.

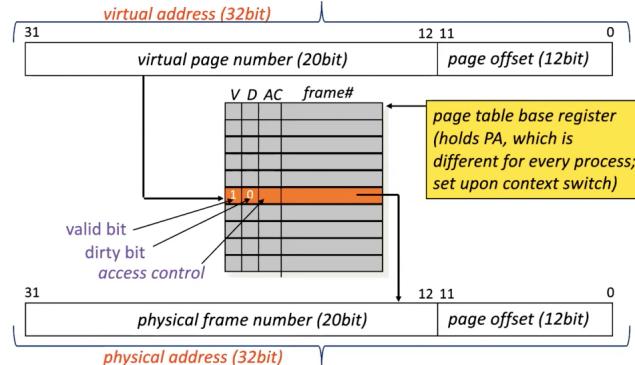
- Map” (virtual) pages to frames, such that VA spaces are contiguous – Pages can be “mapped” into (associated with) arbitrary frames at arbitrary locations
- • Pages can reside – Either in memory (used recently)
  - Or on disk (not recently; they’ll be paged-in on demand)
  - Or be simply unallocated yet (& then allocated on-demand)
  - (We thus allow for the aforementioned memory overcommitment)
- • All programs are written using VAs
  - And “somehow” VAs are seamlessly translated into PAs
  - As we will see later, translation is a HW/SW mechanism, whereby
    - \* OS sets the VA= $\Rightarrow$ PA mappings (“control path”)
    - \* • HW does on-the-fly translation from VA to PA (“data path”)

*Explain. The mechanism of virtual memory is based on page table which translate each page to specific frame. This translation must be very efficient since for each access in memory a process can't access physical memory, he can only access virtual memory so each access memory for some reason e.g reading the next instruction to run, access a data, store data all of them require memory access, so if our access is slow the program will run slow. So there is big investment to modify the mechanism. For now it should be obvious that the mechanism is implemented by hardware. The hardware takes a virtual address then access the page table to do the translation afterward access the physical memory. When we say this mechanism, is hardware we mean a part of processor pipeline which is responsible to do it.*

*Life example for how hardware decode the physical address.* Assuming we have hotel with a different floors and we are given in the reception room number 412 we deduce than our room is in floor 4. But why not floor 41? because if then we deduce that in this floor there is only 0 – 9 rooms which seems not efficient. So in hotel we have more than 9 rooms in each floors. Now back to memory problem assuming we give him to access virtual address number 7000 in which page is it actually? we know that each page is  $4kB$  so we divide by  $4k$  and see that it should be in page number 2



Now how we devide by 4096? Notice that deviding with 4096 of number with 32 bit in register is equivalent to take the first 20 bits of the number since  $2^{20} = 4096$  and from there we take the page number. Afterward, the **hardware** access the page table by the value in register which store where the table start and to it  $4 \cdot N$  where  $n$  stands for the number produce by the 20 upper bits. Then we will get the address of the page table entry which we want to use for map.



As we see here in the picture the hardware take the first 20 bits and access the desired page table entry. Now the critical point that hardware should examine is that checking if the entry is valid or not. Why? because

- 1). The data we want to access it is not allocated and OS wanted to put it temporarily in DISK neither memory.

Hardware can know it by cheking valid bit which hold 1 if valid and the data in memory. Otherwise, we it holds 0 and we have page fault. The OS invoked and understand the reason causes of page faults

- 2). If the access is valid, assuming we have data is read only and we are trying write to it.

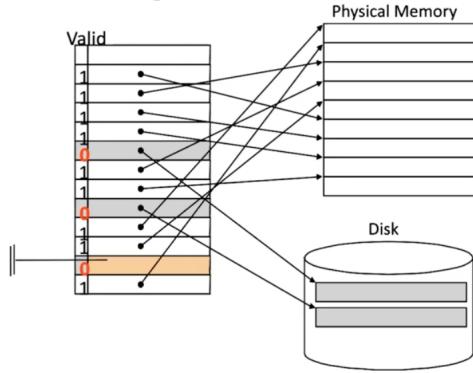
If all of the checks successed, then hardware will take the frame number. How much bytes we need for frame number? remember that the size of virtual address is *4bytes* i.e 32 bits and we devide our physical memory to chunks of  $4KB$  so number of frames at total could be is  $1MB$ . (coincidence only) so we need 20 bits to indicate

frame number which we take them from the entry first  $N$  which we already found (The page number) (4kb) + the offset.

*Illustrating example from life.* Assuming we have container including boxes in specific order. Then we change the container order and we know how we change them e.g container 1 was in order 3 now is in order 8 and else.. Assuming we were supposed to search for box number 13 in container 1. After changing order the box will be the same place in the container which is 13. It's equivalence to what the hardware do, when he search for page he take the old offset in the virtual memory and access for it in the frame after the frame is found. The offset in this case is the lower 12 bits of the number.

*Remark.* The page table is filled by OS and used by hardware OS.

*Remark.* In the example above we said that there is 1MB since we have 20 bits for page number 12 bits for offset. But it's not the case always. Sometimes we can represent a address of physical memory with 64 bit and virtual memory with 32 bits and vice-versa. In intel architecture we use different size for virtual address we use 48 bits and physical address we use 64 bits.



*Remainder:* if 0 is the valid bit then the data is not in memory i.e page fault. Maybe not allocated or in DISK since OS put it there or many other reasons.

**Major page fault.** • Major => need to retrieve page from disk

- (1) 1. CPU detects the situation (valid bit = 0)
  - (a) • It cannot remedy the situation on its own
    - (i) – CPU doesn't communicate with disks
    - (ii) – Moreover, CPU has no say regarding page table contents
- (2) CPU generates interrupt and transfers control to the OS
  - (a) • Invoking the OS page-fault handler
- (3) OS regains control, realizes page is on disk, initiates I/O read ops
  - (a) • To read missing page from disk to DRAM
  - (b) • Possibly need to write victim page(s) to disk (if no room & dirty)
- (4) OS suspends process & context switches to another process
  - (a) • It might take a few milliseconds for I/O ops to complete
- (5) Upon read completion, OS makes suspended process runnable again
  - (a) • It'll soon be chosen for execution

(6) When process is resumed, faulting operation is re-executed •

(a) Now it will succeed because the page is there

*Page fault explain.* We said that page fault is situation in which hardware can't access data because the data is in DISK or not allocated. It wasn't succeed because the valid bit is 0, then interrupt invoke the OS and check the cause of page fault and figure out that the data is not in memory so it ask DISK CONTROLLER to load back the data into memory and through this time a context switch and run the process and when the loading finish from DISK to memory then the DISK controller will send interrupt the data is back to memory by interrupt, OS will treat it and update in the page table to point on the frame in memory so we can access the frame, update valid bit to 1. Now we can put the process into the ready queue and when OS choose to run it wishing time everything will work with no page fault.

### Minor page fault.

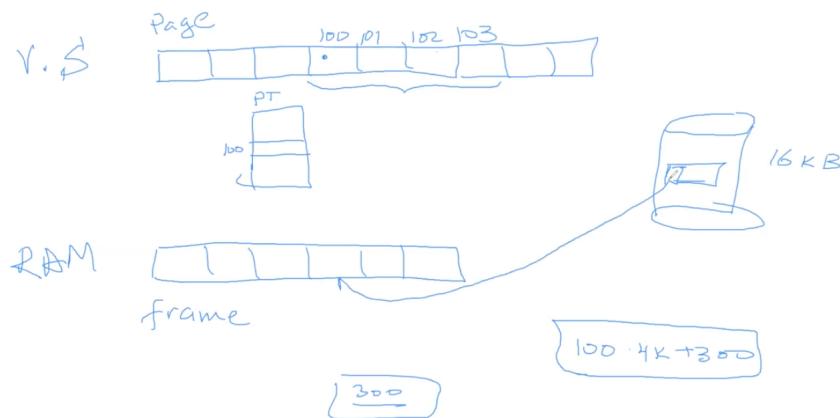
- Not all page faults are major...
  - Sometimes we don't need to go to disk to resolve the fault
  - Instead, we can resolve fault quickly (no need to suspend process)
- Examples
  - 1. CoW (copy-on-write)
    - \* Used for, e.g., implementing fork()
    - \* In child, map all pages to parent memory space, but for reading
    - \* When child writes => page fault => create a private copy for it
  - 2. Demand based (lazy) memory allocation.
    - \* OS allocates memory "lazily" – pages are truly allocated only when actually used for the first time
  - 3. Reading a file content that is already found in the "page cache"
    - \* OS caches previously read files, possibly by other processes
    - Reading a file content that is already found in the "
    - \* • All read/write ops go through the page cache

*Remark.* Not always in page fault situation we should access DISK and load data from DISK to memory. There are examples which don't require DISK access e.g. Cow (copy on write) when a process does fork it duplicates all the data from parent to child but it may be a lot of data moreover when child starts running he will do execv and all the memory space is new. Therefore, in fork we share the page table of parent to children but when parent changes something it won't affect child. We walk on all pages and mark them for read-only, along with parent and child access data for read-only it's fine, otherwise if someone tries to write to parent or child we will have page fault, OS should absorb that its CoW i.e. a child were created and writing to pages is blocked. At this stage OS will duplicate only the page tried to access it and not all the memory of the parent in case parent is the one tried to write, and will point to duplicate of that page then he can do writing. So the treatment is done only in OS and doesn't require DISK access so this type of treatment is called Minor page fault. Other example when a stack finishes after we add more and more data to it then we access something outside the stack, and OS solves it by allocating a new page. (This behaviour is called lazy allocation).

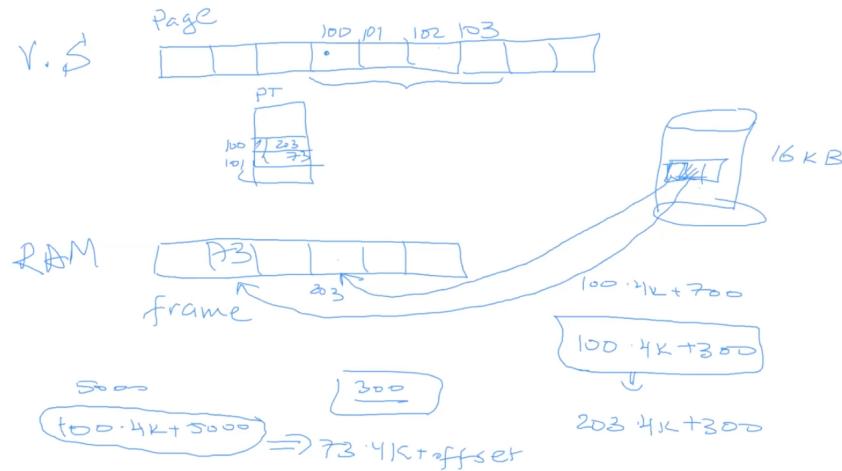
*Termonology.*

- “Page in” & “page out” a chunk of data
  - Page in copy page from disk to DRAM (= read)
  - Page out copy page from DRAM to disk (=write)

*How in general we read from a file?* The first thing we do is sending OS a buffer which says how much bit we need to read from the file into the buffer then we can use it. Its complicated approach since each time we need to copy from the file and buffer, why not instead take the file and load it to memory. First OS ask for the file size, allocate in memory enough space and copy all the file to memory. So how can we do it (IT CALLED MEMORY MAPPED FILE)? Assuming we have a physical space and virtual space, we also assumed that there is page table which enable to translate from virtual memory to frames, assuming we have a file on DISK of size  $16kB$  the mechanism memory map file is choose  $16kB$  contagious in the virtual memory, assuming page number 100, 101, 102, 103. Assuming we want to access a file in physical address 300 so we are trying to access page  $100 \cdot 4k + 300$  this is the virtual memory we want to access in the page table at the place 100 and figure out where is the file frame. In this case we will get page fault since the file loaded in DISK therefore, OS invoked and take the first  $4kB$  of the file from the disk and loaded to a space in memory e.g into 203.



Now when a process try to access the virtual address, the hardware do translate and announce that the file is in physical address  $203 \cdot 4k + 300$ . What happen when if there is branch and we jump 5000 instead of 300 offset, now what will happen is that hardware try to access page number 101 again we will have page fault then OS invoked load the next  $4kB$  to memory e.g in 73 and now we can translate that to  $74 \cdot 4k + offset$ . When it could be problematic? When the file is very big, since the amount of virtual addresses is huge e.g map a file of 8GB. Also when we run video each time we read a chunks and we don't scan the whole movie. If we have data base we jump a lot so apparently its better to do a memory map.



### On-demand paging & readahead.

- • On-demand paging rpt
  - OS maps / reads page from disk into DRAM only if/when the process attempts to access it for the first time (and, hence, a page fault occurs)
  - That is, OS pages-in data only via page faults (+ prefetching; see below)
  - Thus, a process begins execution with most pages “unmapped”, possibly not residing in physical memory (executable code included), and page faults occur until its pages are placed in in DRAM & mapped
  - Also called “lazy” loading – What’s the benefit? Reading/mapping is costly, & with demand-paging we only read what we need
- Notice: on-demand allocation also works rpt
  - For anonymously mmap-ed/(s)brk-ed memory
- Readahead prefetching (anticipatory paging) –
  - read() does prefetching when identifying sequential access
  - The page fault handler does the same when doing on-demand paging
    - \* Complements demand-paging in an attempt to minimize page faults

*Explain. We want to talk about swapping or paging (the same mechanism which let OS take a data from a memory to DISK, they do it in case no place remain at memory, there is a lot of processes running and there is no memory still). First we want to see then it happen, we should examine that moving pages from memory to DISK we want to avoid because in after we want to take it out. In other hand if process run and want to access such a data in which is not available in DISK neither memory then OS should free some space in memory by sending the data freed to DISK and copy the non available data to it each time og this page fault will consume to much time. So we try to avoid (swap out which is freeing memory) this*

*type of page fault by keeping enough memory in advance to avoid those situations (not always possible) by keeping empty blocks for use this progress is called Working set to ensure that there is enough space in memory. (Used in WINDOWS). Working set also load pages in advance to memory. Our target is to avoid loading from DISK to memory we don't do prefetching i.e we don't load pages in advance for use, we just do it when page fault interrupt occurs i.e for the demand.*

**Working set.** One of the trials is to ensure that there is enough place in memory or load pages in advance is called working set and used successfully in WINDOWS. The principle is the last accessed e.g in the table here we look at the 3 last accesses  $WS(k = 3)$  e.g in  $t = 10$  we accessed 1, 5, 7. If the process back to run we want to ensure that these pages are in memory. Our assumption if the working set is small i.e we look at small amount of last accesses then we can assume that the process will access those pages again and we don't want that process back running and trying to access and they are not in memory then we get a page fault so we worry in advance to load it in memory to save number of page faults. But notice that it can be guaranteed for use after loaded back to memory.

- Definition:  $WSP(k)$  – Pages accessed by process (or thread) P in the last k accesses

| time          | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---------------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|
| page accessed | 2 | 6 | 1 | 5 | 7 | 7 | 7 | 7 | 5 | 1  | 6  | 2  | 3  | 4  | 4  | 4  | 3  |
| WS( $k=3$ )   | 2 | 6 | 1 | 5 | 7 | 7 | 7 | 7 | 5 | 1  | 6  | 2  | 3  | 4  | 4  | 4  | 3  |
|               | 2 | 6 | 1 | 5 | 5 |   |   |   | 7 | 5  | 1  | 6  | 2  | 3  | 3  | 4  | 4  |

- For a fixed value of  $k$ , smaller  $WSP(k)$  indicates more locality
- During time 1—17, there are 7 pages that p accesses • We informally call these pages p's current “working set”
- What happens if the current working set is too big to reside in DRAM (and/or if memory is currently tight such that there isn't enough of it to house the current working set)

### *Thrashing.*

- When we've overcommitted too much memory
  - There isn't enough physical memory to back the currently-inactive-use virtual memory
- The system might enter a state of “thrashing”, that is
  - Virtual memory is in a constant state of paging, rapidly exchanging data between memory & disk – Nearly nothing else is done in the system (causes performance to degrade/collapse)

In the 2000 new version of WINDOWS ME they wished it would be successful but many problems occurred was Thrashing of memory i.e when OS with ideal factor in processes amount OS was doing a lot of swap in and out, which take too much time for memory management.

### Virtual memory: did we achieve our goals?

- • Illusion of contiguous memory –
  - Yes: virtual memory is contiguous by definition
- • Illusion of large memory (possibly bigger than physical mem)
  - – Yes: chunks that don't fit into physical memory may reside on disk and on-demand allocation also helps
- Dynamic growth
  - – Yes: heap & stack can grow at runtime by mapping more VAs to PAs, in order
- Isolation between processes
  - – Yes: same VAs point to different PAs; as long as we keep PAs disjoint on a per-process basis, the processes are isolated
  - – Still, sharing memory is possible and easy (how?)
- Memory overcommitment
  - – Yes: using the disk,  $\sum_{i=1}^n vmem_i$  (for n processes) can be  $>$  physical size
- Access control
  - Yes: HW enforces PTE (= page table entry) bits; e.g., it will reject a write to a page that is marked 'read-only'

### But how does it perform?

- (1) Temporal locality helps
  - Typically, during a given time interval, a process uses only a fraction of its memory, over and over again
  - So it's fine to keep currently unused parts on disk – As long as the working set is in memory most of the time
- (2) Asynchronous nature of OS, when doing I/O, helps
  - Writes are non-blocking by default: when writing a page to disk, we don't need to block the associated process
  - When reading stuff, we can run other processes
- (3) Demand-allocation and paging help
  - Pages fetched from secondary memory only upon the first page fault, rather then, e.g., upon file open - we bring only what we need
  - Likewise, page allocation is done only when needed
- (4) Special locality helps
  - – First, we work in page resolution = compatible with special locality
  - – Second, when reading page P from disk, we may employ the readahead optimization = bring in some additional file pages from immediately after/before P, even though they weren't accessed yet
  - As we've learned, "locality principle" suggests these pages would be used soon
    - \* Locality = special locality + temporal locality
    - \* • Locality principle contends that most programs exhibit special and temporal locality when utilizing the memory

- So some pages don't induce major page faults when accessed for the first time; they are simply "there"

*Explain 4. locality in space. E.g local variables, we access a variable then we access variable closed to it. for example local variables and they are closed one to other. Or adjacent entries in array. This is important because if we assume that it not the case then each time we want to acces data then each time we need a new page which may be not in memory, in this way since all the variables, instruction are closed we avoid allocating new pages. So the chance to get page fault will be lower.*

- (5). Making VA=>PA translations fast helps
  - The TLB (translation lookaside buffer) is a small, fast HW structure that caches recently used VA=>PA mappings
    - \* Size of TLB L1 can be, e.g., 32-64-128
    - \* Size of TLB L2 can be hundreds of entries
  - Given a VA, HW first searches for its translation in the TLB; only if it's not there HW access the in-memory page table
  - Accessing the TLB takes very little time (e.g., a few cycles, similarly to L1)
  - Even though the TLB is relatively small, locality principle typically ensures it is rather effective
    - \* Special locality: since we work in 4KB page granularity, lots of nearby accesses fall in the same page
    - \* Temporal locality: same pages are used repeatedly
  - Lots of workloads (though certainly not all) approach 100% TLB hit rate

*Explain 5. We already understand that number of pages faults are small otherwise the whole mechasim is bad. For each access there is need to translate, e.g in order to access adress of x we should access the page table translate and then acess x so there is two accesses to memory one for the table and one for x. So how we solve the problem by making effcient mechanism also in translating aspect from virtual address to physical address?.*

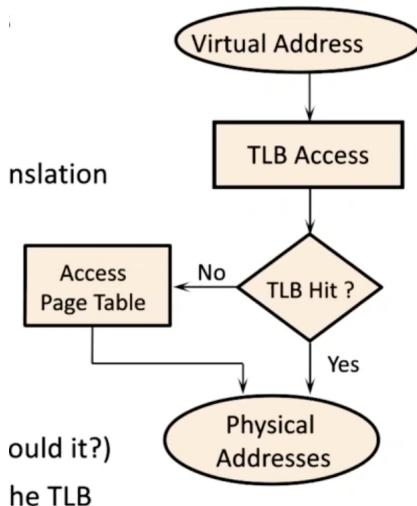
*Remark.* We don't need for ach process allocate all the memory in advance because no memory will remain for other processes so they will do swap in and swap out

**Reminder.** We talked about virtual address and saw how translate occur from virtual address to physical address and how can we isolate memory of each proccesses and why contagious, and we end up in the point of performance. Actually, we want to improve the time takes for translating from virual address to physical address. So how can we improve the translation? In orer to do it effcient there is special hardware known by TLB its a part of LVL1 cashe (LVL1 cahce part of it allocated for TLB which remember the last translation).E.g if want to translate page 30 then we access memory and we read in the which frame assuming we got 300 then the hardware update after translation finish to hold the map, and next time instead of accessing memory it access the cache TLB and get the translation in immediate way. What is the TLB size? first we need to notice that there is two thing which affect the performance of TLB, we talked about locality in time and locality in space i.e localityt in time for the same variable we access again and again appreantly in short period (e.g counter in loop) or we access a adjacant address (e.g

in array lot of time we access a adjacent addresses). So why its important? The TLB help translating page to frame so we do once a translation then we get in TLB translation for a whole page i.e for each address in the page we don't need to do translate using memory instead we use TLB. Hence, the most our program access less variables in small time period the most TLB performance is better. In reality most of the TLB we use are very small (32-64-128 entries) even though TLB are small we got a good coverage i.e if we look at process from starting his quantum to end of the quantum 99.99% accesses to memory would do translation using TLB and not using page table.

$VA \Rightarrow PA$  translation with TLB (Translation Lookaside Buffer).

- Page table resides in memory
  - Each translation requires a memory access
  - Required for each load/store!
- • TLB
  - Cache recently used PTEs  $\Rightarrow$  speeds up translation
  - Typically:
    - \* HW fills TLB automatically by reading the page table on its own (no SW involvement)
    - \* OS can invalidate TLB entries (when should it?)
    - \* Processes are completely unaware of the TLB



We can see in the photo the whole progress, first we want to translate virtual address and split it to offset and page number and using page number we access the TLB the TLB could say that he know the translation, for example you are searching for frame number  $x$  then we can access it. Otherwise, it access the page table and do translation using page table, and we will see how are those table organized in OS.

*Remark.* Who remove list and to lists is OS of the TLB? In the intel TLB the OS invalidate the lists in two cases. The first case when context switch occur since everything in the TLB now is not valid (other process occupy), and he got different

memory than the current one). In other case is that OS decided to do swap out i.e it take frame and put it in DISK and from now the entry in TLB is not valid since the page is not in memory so OS do invalidation for one entry only in TLB. The one who update the TLB is hardware, with the new entry of the translation. Notice that context switch between threads the way Linux implemented we have register CR3 which point to the page table. Each time we do context switch between processes it check if the previous CR3 is equal for two processes in case the answer is yes it mean that context switch is between thread so OS won't invalidate the TLB and this is one of the reasons when we saw the old implementation scheduler of LINUX when he try to run thread on the same core he gave them bonus, because when we do context switch etween two thread the penalty of the cold start is much lower (because no need to invalidate the TLB unlike processes context switch ).

*But how does it perform?*

- Using and intelligent page replacement policy help
  - When we need to evict a page from memory to disk
    - \* • The page replacement policy decides which page it'll be
    - \* Chosen page called the “victim” page
    - \* – Also called “page reclamation policy”
  - – Goal
    - \* • Minimize number of future page faults • Minimize price of paging (evicting dirty pages costs more; why?)
  - – Typically done via a daemon process (“swapper”) that runs in the background
    - \* • Start: when number of free frames drops below some
    - \* • Stop: when number of free frames exceeds some threshold

*Last thing which effect the performance is the algorithm for freeing the memory is the order of freeing that memory. We need to decide in which oder we want ot free the memory assuming we have more than one frames and we need to choose between them. In non good algorithm we free that frame and insert to that frame a new data then we continue to run and immediately we want to access a memory which was free before short time period hence there is big effect of the reclamation algorithm to decide which data more worthwhile to stay at memory.*

### Page reclamation algorithms.

- • Belady – optimal, but theoretical
  - – Greedily page out page accessed furthest in the future
  - – An “off-line” algorithm = we know the future memory accesses
- • LRU (least recently used) – not theoretical, but still impractical
  - – Leverages the locality principle
  - – But typically too wasteful to be used in practice
    - \* • OS data structures must be updated up on each mem ref
- • Clock – a practical LRU approximation
  - – Pages are cyclically ordered –
  - Maintain a per-frame bit: “was it referenced since I last checked?”
    - - \* Cleared every n seconds (a linear pass “zeroing round”)

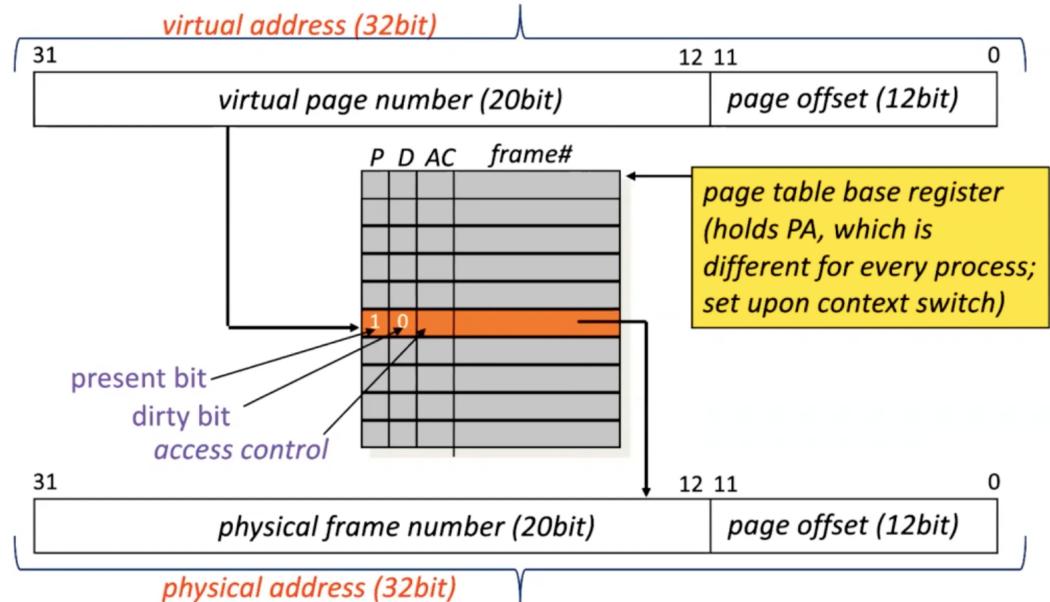
- \* • Set upon first access since zeroing round (implemented in SW or HW)
  - – “Current” iterator points to last examined page frame
  - – Search clockwise for first page with bit=0 – this is the victim page
  - – Set bit=0 for pages with bit=1
- NRU (not recently used) – a more sophisticated LRU approx.
  - – Can be based on ‘clock’
  - – Underlying principles (order is important):
    - \* • Prefer keeping referenced over unreferenced
    - \* • Prefer keeping modified (dirty) over unmodified
  - – HW or SW maintain per-page ‘referenced’ & ‘dirty’ bits •
    - \* Periodically (every n seconds), SW turns ‘referenced’ off
  - – Replacement algorithm partitions pages to
    - \* • Class 3: referenced, dirty
    - \* • Class 2: referenced, not dirty
    - \* • Class 1: not referenced, dirty
    - \* • Class 0: not referenced, not dirty –
  - Choose victim page from the numerically smallest class

*Belady algorithm is optimal for deciding which data stay at memory and give the minimal number of page fault, the main problem of the algorithm is that he need to predict the future in order to know which acceses may occur. Since the beladt algorithm wach time want to move frame from memory to DISK the first thing do is looking at the content of all frames and evaluate which frame we are going to access after a long period of time and free it. The algorithm is not in use except if we do offline proccesing and we know in advance which pages need to be freed otherwise we have noting to do. So the algorithm is not useful since we can predict the future. Here comes the idea to look at all the pages we need soon and pages we dont need soon so we free the one not soon, but instead of predicting the future an alternative way is to learn from the past the access of pages was in use and consider the page which its access to it occured long time ago apperently this page we don't use it frequently, so the algorithm LRU look at all frames and choose the page which its access to it occurred long time ago and free it (for high probability the data in that page is not critical for now, in genereal LRU give a high performance but not always. But why we don't use it? the reason LRU is not in use because of hardware limitations since each access to a page/frame we need to understand what is the time stand and update the time stand i.e for each page we access we need to updaten the time stand which is impossible. Moreover, we want the hardware manage the heap by finidng the page we want to free efciently so all of that reasons let LRU be not practical. So all what can we do is approximation of LRU (clock) which in realiry is not even approximation to LRU however its LFU which mean least frequently use and not LRU which is (least recently use) i.e the more accessed more frequently got more priority in memory and OS try to save them in memory. The algorithm allows to doit which constitute a base for many reclamation algorithms. Algorithm clock do the following, for each page we add bit called bit access and we rely on hardware each time it access a page then it update the bit access i.e each time hardware do translation and access a page it will update the bit access to 1. So why we need this bit? When we want to find a frame to free we scan all page tables i.e we walk on all*

*the page table page by page and we check if the page is not in memory then we skip it. Otherwise, we look at bit access if the bit is turned on then it tell the algorithm that recently that page were accessed hence all the OS do is turn off the bit and move to the next page. It keep that page in memory, and sometime, it will return it because it walk on the pages in cyclic way so after it finish scanning all the pages if OS return to the same page if we keep accessing this page again and again then the bit will be one because each access we turn the bit on hence OS again turn it off and skip it. Otherwisem if OS find a page with bit access 0 it will free him from memory to DISK. I.e the more we access a page his probability to stay in memory is higher. So its approximation of LFU. Now NRU is extention of clock algorithm (modified) with experience to approximate LRU which free the pages we didn't use recently. First NRU deivse the pages into classes where the lower class the high classes hold the pages which were recently accessed called referenced, we always try to free the pages in class 0 as priority also in class 0 there is examination between the pages were changed or not changed earlier and the page didn't change earlier are more qualified to be freed from the memory. When OS start find the pages to free it start scanning of the pages and if a page with bit access 1 then we turn it off and put it in reference otherwise we put that page in non reference (i.e class 0) at the beginning we search for the pages which are not referenced and weren't changed if we find adequate number of pages like that we free them all, if we didn;t find we complete from not reference but weren't changed, and if still then we complete by referenced pages but weren't changed and if still not enough we take from reference pages and were changed. Now how do we know that a page changed? Each time hardware access a page it turn on the bit access in addition if it write then it turn on addition bit called dirty. So using that we can know if page changed or not.*

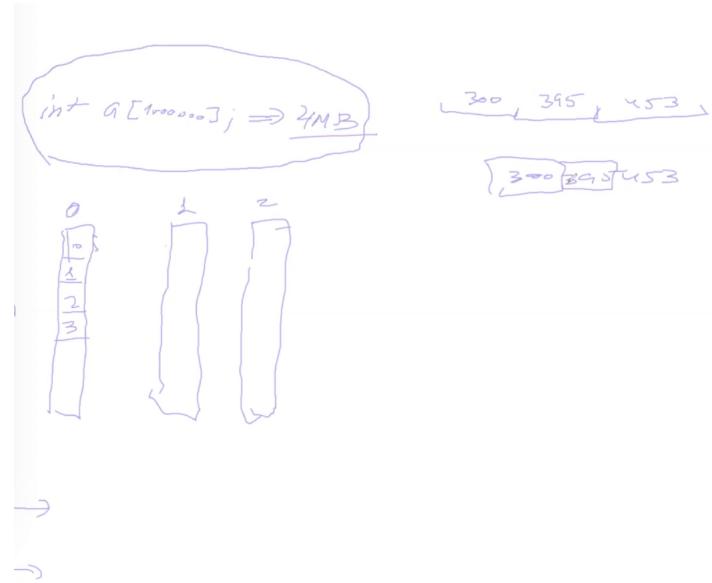
*Remark.* Bit access is also in TLB.

## Reminder: 32bit address translation

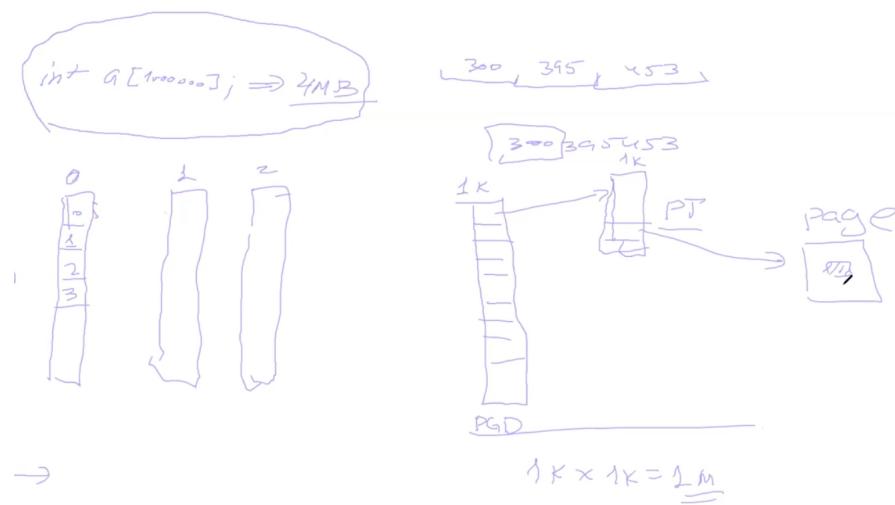


We already saw that for translation purpose we need first for each process save the page table then we get virtual address and devide it by 12 bits for offset and 20 bit for virtual page number. Assuming in page offset we got 70 and the virtual page number is 3001. Then we access the entry 3001 and check bit present and etc. We need this table which include  $2^{20}$  entries all combination of 20 bits therefore in total we have  $2^{20} \cdot 4\text{ byte} = 4\text{ MB}$  (each address is 4 byte - 32 bit) so for each process we need to allocate translate table is  $4\text{ MB}$  and assuming we have 100 processes then we need  $400\text{ MB}$  of physical memory which is enormous, most worst is that this table must be allocated in contagious form since the access of the entry of the table depend on offset e.g in our example the hardware will do  $\text{CR3} + 3001 \cdot 4$  (CR3 point to the page table register) in order to reach the page and this work only if the table is contagious in memory so now in memory we have pages which are set in memory so we will get external, interior fragmentation. Observe that most of the affine processes use  $4\text{ MB}$  i.e 1k pages but its exaggerated and not needed.

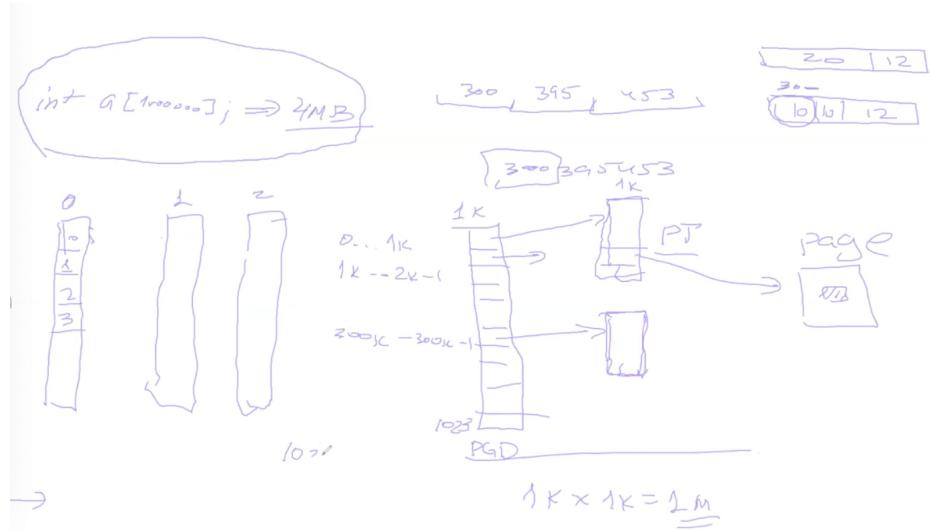
*Reminder: External fragmentation is a case in which we run out of contagious memory for other processes. And interior fragmentation is using less than allocated. Assuming we have a storage which holds containers ordered in rows,*



At first we go to the row number 300 we search for contaienr number 305 and item number 453. so this is exactly what happens in memory. We build a one table in which in it there is  $1k$  entries, each entries point to a additional table with also  $1k$  entry so we have  $1k \cdot 1k = 1M$  which is the same as we have at the beggining we remember the the page table contained  $1MB$  enries therefore,  $4MB$ .



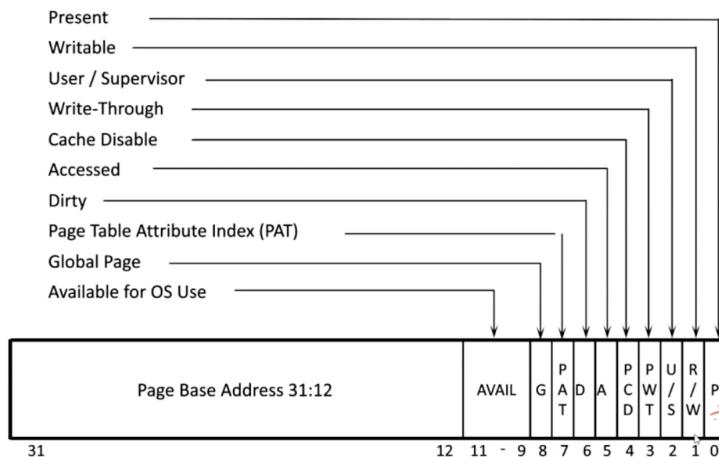
Now as we see above if we have 32 bit virtual address now insted of 20 bits for table and 12 bit for accessing frame, so back then we got a enormous table of  $1M$  entries. Now our virtual address is devided into three parts first part is 10 bit which till the entry at the first tabke, then another 10 bit which till the entry at the second table and 12 bits to determine the entry in the frame.



Now we can see that there is no external fragmentation. Because there is no need for  $4MG$  contagious, so we need to allocate  $ogd$  which has  $1k$  entry each is  $4B$  hence  $4KB$  the  $pgd$  size and each entry point to table of  $4KB$  so we need to manage in our memory only block of  $4KB$ . So the problem of external fragmentation is solved. Now for that internal fragmentation no one force us to allocate all the page table or all the  $1k$  entries, we can allocate only two entries and we allocate just as we need which is more less than  $4MB$ . So internal fragmentation is also solved because we don't allocate more than we need. But now we are in performance problem, remember that when we devide virtual address to 20 bit and 12 bit we had only one accesses but here we need to read entry from  $pgd$  then another access to read drom  $page$  table then another access for reading the data therefore, we have 3 access i.e the translations is way harder, But what save us? The  $TLB$  help us in order to avoid the unnecessary accesses by giving him the 20 bit he will give us immderately the frame number and no need for heavy translation. So we got a system of virtual memory which is more comfortable.

*Remark.* We can see above the first table is  $PGD$  then each entry to it point tp  $PT$  (page table) and each entry in this table is called  $PTE$  (I.e page table entry).

## 4KB-page PTE format



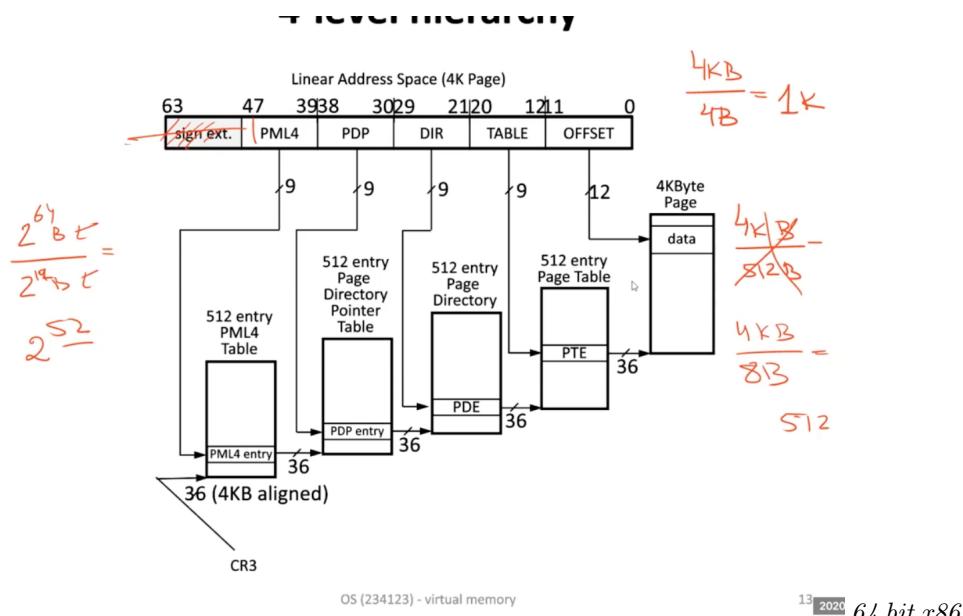
31

12 11 - 9 8 7 6 5 4 3 2 1 0

*What is the PTE format?* The bits 12-31 is the frame numer, bit number 1 is read/write it determine with it valid to write to page or not. User supervisor is bit 2 determine who can access. Bit number 3 we will discuss in files system but in general when we write data to DISK the data is not transmittied automatically first we write to memory thien the disk driver take from memory and write it to DISK if for each update (each time we want to write to DISK) we will get enormous amount of writing to DISK which are so slow, so we write to memory and won't ask the DISK driver to write, at some point when we close file or user ask for flash then we access the DISK driver and each thing we write to this frame write it to DISK, some of the information we don't want to do it. 3 Bit is cache disable which mean the data can't reached the cache. 4 bit is Accessed and 5 is the Dirty bit we already discussed. The 7 bit is not discussed in the course. The 8 bit is Global page bit. OS has not private space so it sit on proccesses spaces e.g assuming a procces run the OS can access a data using that procces space which means the there is pages are mapped from all memory spaces wihch belong to OS and they are marked as global page. So why its important? for example in context switch there is no need to remove it from the TLB when it do flash because its identical to all procceses. The availabe 3 bits are not is use (OS can do what ever with them).

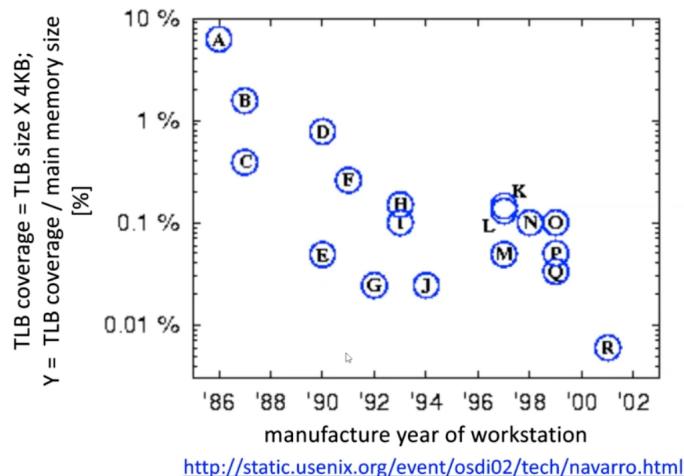
## 64bit x86 address translation

- **64bit address means  $2^{64} = 16$  Exabyte address space**
  - $2^{20} \approx 10^6$  = Megabyte
  - $2^{30} \approx 10^9$  = Gigabyte
  - $2^{40} \approx 10^{12}$  = Terabyte
  - $2^{50} \approx 10^{15}$  = Petabyte
  - $2^{60} \approx 10^{18}$  = Exabyte (= a billion GBs)
  - DRAM typically can't be that big; 64 bits are currently more than needed
- **Current x86\_64 HW usually uses "only" 48 bits => 256 TB**
  - Still more than enough for existing systems; if needed, can do 56bits
- **48bit address reflects a 4-level hierarchy, divides into 5 parts**
  - 9bits X 4 levels + 12bits offset
  - Each PTE is 8 bytes (rather than 4, to be able to hold the wider address)
- **The job of the x86 64bit virtual memory subsystem**
  - Translate 36 bits (= virtual page #) to 36 bits (= physical frame #)
  - As before, pages are 4KB-aligned



address translation. We got noting to do with 64 bit but we exploit 48 bit for virtual memory address in which its last 16 bit are sign extension, now the size of page in this architecture is 4KB but the size of PTE now is 64 it can be 64 because number of frame we need to support is  $\frac{2^{64}}{2^{12}} = 2^{52}$  so we need 52 bit for frame number hence t32 bit is not enough so it will lead to the in page the entries number os  $\frac{4KB}{8B} = 512$  entries in so we need 9 bits in order to access a entry. Now what's important here is the TLB performance, the most better performance then we don't need to do page walk.

## TLB (L1) coverage drops exponentially



The reason we need to move to 64 bit is that we need more memory. Not only more memory required, CPU are more powerful. E.g if in one quantum could access memory way less than today. Above appears a figure demonstrates the coverage could TLB we can see that 0.01 was the coverage in 86 is 0.01 from memory, untill 2002 the TLB coverage was reatlively low and page walk are required.

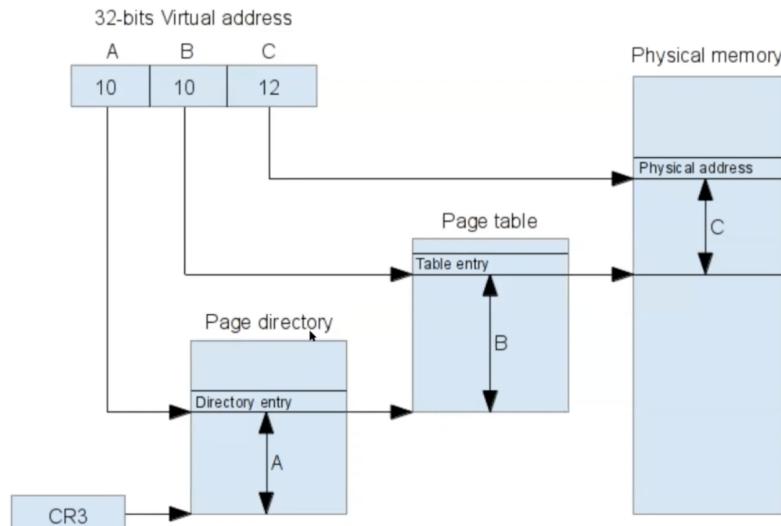
## How to increase TLB converge?

- **Superpages, a.k.a. hugepages, of page size > 4KB**
  - Different sizes supported by different architectures; for Intel/AMD:
    - 32bit architecture\* ("x86"): 4MB (why?)  
– \* With PAE (= page address extension) mode disabled
    - 64bit architecture ("x86-64"): 2MB, 4MB, 1GB
  - In Linux, THP (= transparent hugepages) is typically on by default
    - The khugepaged daemon dynamically+opportunistically creates hugepages in the background
    - Homework: think about how khugepaged works
- **TLB hierarchy**
  - Like the caches L1, L2, ...
  - Architectures can also support TLB-L1 (size = a few dozens), TLB-L2 (size = a few hundreds)
  - As usually in a caching hierarchy, TLB-L2 is bigger but slower

Intel fixed the problem by adding additional cache lvl2 which is bigger then lvl1 cache but the access to it is slower therefore the translation we do all the times will be in TLB so the variables we access a lot will be in TLB LVL1 cache i.e TLB and the ones we access once in a while are in TLB LVL2. So more heated data will be in TLB1 and less are in TLB2. So this optimize a lot TLB performance but assuming

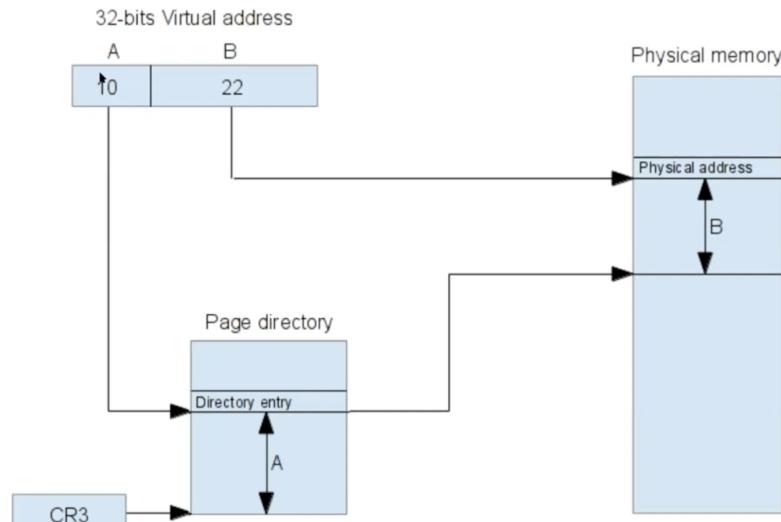
there are processes which require a large amount of memory and lot of time and they can't decide whether a data is cold or hot so they need many data that all of them are warm and accessing them is fast. So which optimization Intel did?

### 4KB pages translation in x86/32bit



What happens here is assuming we have a big process which exploits large amount of memory so it uses many pages therefore it requires many entries of TLB. What helps this process? if the pages are way larger then he will use less entries in TLB so intel could also run those type of processes. Therefore, it gives the process option to choose the page size. E.g. we can look at the following figure,

### 4MB pages translation in x86/32bit



we can see that the process frame size is  $2^{22}$  bytes i.e 4MB therefore, a small TLB gives enormous coverage for this type of processes.

### 3 address types: effective => virtual => real

- **Effective**
  - Each process uses **64-bit** “effective” addresses
  - Effective addresses aren’t unique per-process
  - More or less equivalent to x86 “virtual” addresses
  - Get translated to PPC “virtual” addresses
- **Virtual**
  - A huge **80-bit** address space
  - All processes live in and share this (single) space
  - Namely, if two processes have a page with the same virtual address
    - Then it’s the same page (= a shared page)
  - *Not* equivalent to x86 virtual addresses
  - Get translated into physical (“real”) address
- **Physical (a.k.a. real)**
  - **62-bit**

#### 64 BIT POWER-PC Architecture.

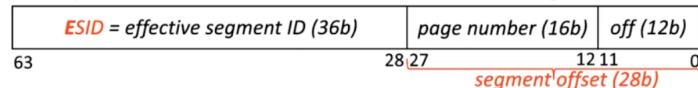
Its a system which the main rule is to serve lot of heavy services. The most we want use heavy proccesses and many data. Moreover, it assume in advance that many proccesses will share memory. Unlike Intel it has different hierarchy, they split the memory to 3 hierarchy the first hierarchy is called the effective memory its pareller the virtual memory of Intel and its private for each proccess and the size of address is 64 bit. The next hierarchy is virtual which has no connection of the virtual memory of Intel in termonoly that its memory of POWER - PC is shared for all proccesses e.g if *A* proccess access virtual memory address 30 and *B* proccess access virtual memory address 30 then both of them access the same data. Since the virtual space must contain the data of all proccesses then it must be very big so they decided to use virtual address of 8– bit. And the last hierarchy is physical address which is 62 bit in power pc architicture.

## PPC Segments

- **Effective & virtual spaces are partitioned into contiguous segments of 256MB ( $= 2^{28} = 2^{16}*2^{12} = 64K*4KB$  pages)**
  - Each segment is **contiguous** in the per-process **effective memory space**
  - Each segment is **contiguous** in the single, huge **virtual space**
  - Segments are 256MB-aligned and can be private or shared (why?)

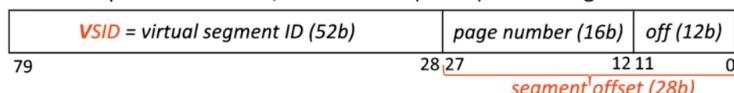
- **How many segments can there be in an effective space?**

- Effective space size is  $2^{64}$ ; so can be  $2^{(64-28)} = 2^{36}$  segments



- **How many segments can there be in the virtual space?**

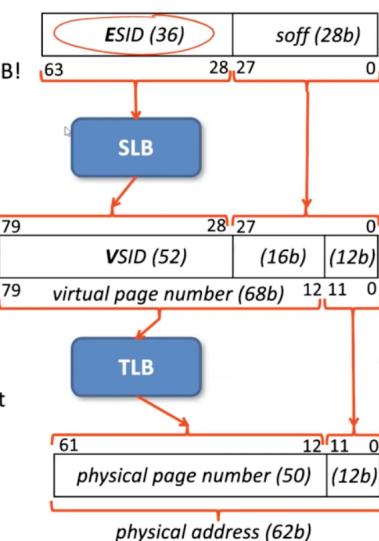
- Effective space size is  $2^{80}$ ; so can be  $2^{(80-28)} = 2^{52}$  segments



So how the translation work from effective address? The effective space we devide into big pages as we saw in Intel in order to improve TLB performance by devding address of 48 bit to 28 bits of offset and the reminders bits are *ESID* (EFFECTIVE SPACE ID) which is the page number in the effective space. So hardware take this effective number and translate it to virual page number, so as we see above hardware it translate those 35 bits to virtual segment ID which is 52 bit. How the hardware do it?

## PPC SLB (segment lookaside buffer)

- **SLB is a fast, small HW cache**
  - O(32) entries, each points to 256MB!
  - O(1) cycles to access
- **HW searches SLB**
  - To find *ESID*=>*VSID* mapping
  - Each STE (segment table entry) contains *ESID* & *VSID* info
- **OS explicitly manages SLB**
  - Maintains *ESID*=>*VSID* for all segments of the process
- **Upon SLB miss**
  - HW raises “segment fault” interrupt
  - OS will then insert right STE
- **Upon context switch**
  - OS invalidates SLB (not shared)



So as we see above the hardware store SLB (The same TLB in Intel). If SLB success to translate it then we get the virtual memory, otherwise (unlike intel which we did walk page) here we get page fault so OS translate it. After OS finish to

translate it update the SLB with the new translation then hardware can do the translation. Now we are moving to the harder step which is taking the 80 bits of the virtual memory and translate it to physical address. Now observe that physical frames can't be large enough because we could lose a physical address which is important so here we can see that the physical frames are  $4KB$  i.e 12 bits. So the hardware devide again the virtual memory, the 12 bit (LSB) are the offset and the other 68 bits are the virual page number which need to be translated to the frame number. So how the virual page number (VPN) is translated to frame number. First we got TLB as in intel (way larger but more slower) so hardware access TLB if the address found then we are finished because we got physical address. Otherwise, we have hash table which is defined in hardware and two hash functions are known to hardware. Hardware take the VPN and apply the first hash function on it then it access that hash table and it return the value in the key it calculated. (Notice that the hash table contain translation in the past). What there is in the key it calculated is that there maybe many collisions, assuming what there is a 8 translations that hardware got e.g 8 *VPN* 13 is translated to 23 and *VPN* 16 to 28 so what it do is just going on all those 8 translation and check if there is translation compatible if yes then it update the TLB. Otherwise, it take the other hash function again it apply on the VPN and get 8 translation in the new entry it get, if the translation found then we it finish. Otherwise, hardware ask OS to treat the page fault. Again OS can do the translation as it wishes unlike Intel archticture. After it finish it choose randomly one of the hash function on the VPN and add new translation to the table. Then the control is back to hardware so it doit again in wish it success. In general TLB of PC-POWER architicture is equivalence to TLB lvl2 of intel. The advantages of all this is that pages are in large physical space hence small SLB and effcient therefore, when we do context switch to fill it take small time amount. Thereforem the translation from effective space to virtual space is effcient, the problem is translation from virtual space to physical space. The first point is that in context swich there was a need to eliminate *TLB1*, *TLB2* becuase all the translations are invalid unlike PC-POWER because the virtual memory is shared to the procces therefore context switch won't eliminate TLB. But the translation of *TLB2* is so slow hence for small procces Intel archticture wins hence IBM engineers understood it, so they added ERAT cache which take the effective address and devide it to 52 bit for the key and 12 bits for the offset then we ask ERAT if the translation sound familiar then it gives the translation immediately from the effective address to physical address wishing the small procces wouldn't pass through the virtual space. So in this way POWER-PC run effciently small proccesses.

**Intel vs POWER-PC ARCHITECTURE..** Power pc architecture is better becasue it doesn't contain the hierarchy of pages table in Intel. But for the most of smaller procces Intel could be better the the POWER-PC architecture. So in total POWER-PC shows other approach to memory managment.

*Remark.* The hash table we have in POWER-PC is not like the hash table learnt in data structures. Its limited with number of element in entry, at most 8 elements in each entry.

# הרצאות

## מערכות הפעלה.

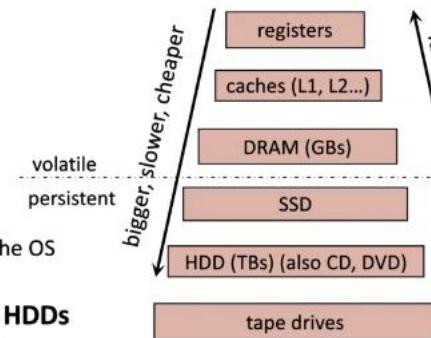
### Why disk drives? (Memory not enough?)

- **Disk drive = secondary storage**

- Persistent (non-volatile)
- Bigger
- Cheaper (per byte)

- **Recall that**

- CPU (or rather, its MMU = memory management unit), can't access disk drives directly. It's all done by the OS



- **We'll typically think on HDDs**

- Spinning hard disk drives
- Still being used in data centers (albeit rapidly replaced by SSDs)
- Spinning, moving head negatively affects random access

עד היום ראיינו התרשים הזה הרבה פעמים. דיברנו על רגסטרים, RAM cache, כל התקנים האלו הפ מהירים יחסית ופיזית יתר קרובים לprocessor אבל הם גם נזיפים, ככלומר שמכבים המחשב מידע בהם נעלם. אנו רוצחים התקנים שמאפשרים לנו לשמר מידע לאורך הזמן (לכבות מחשב, להעביר מידע למקום אחר). ופה בעצם מגיעים להתקנים שהפ הרבה יותר גדולים בונפה והרבה יותר איטיים למשל SSD(solid state Disk), HDD, ועוד שהם גדולים מבחןת נפה ואיטיים ובהרבה מקרים דורשים מיטבוצה פיזי לקרוא מידע. בהרצאות נדבר על SSD, HDD ולא נלמד tape ים. האם יש שימוש בtape ים? כי הם זולים משתמשים בהם לכל מיני נתונים שלא מצפים לשולוף אותם בכלל למשל בארה"ב כל הליך משפטי חייב להיות מוקלט ולשמור אותו להרבה שנים.

## Goals: when using disks, we want...

- **Persistence**
  - Outlive lifetime of a process, tolerate power outage
- **Support ownership**
  - Users, groups; who can access? (read/write)
- **Robustness**
  - In the face of failures
- **Performance**
  - As performant as possible
- **Concurrency support**
  - What happens if we access the information concurrently
- **Storage drive abstraction**
  - Access similar to HDD, SSD, DVD, CD, tape = all are “block devices”
    - Block device = IO is done in block (=several bytes) resolution

מה הדרישות שלנו ממחסן (מערכת קבצים).

צריכים שיהיה עמידות בזמן של נפילות מתח או חיבר שיהיה לנו גיבוי והמידע לא לך לאיוב. רוצים לשמר גם על פרטיות של קבצים, כלומר רק אני או אנשים שמוכן לחת להם הראשאות יוכל לגשת לקובץ. רוצים גם פיתרון שהוא הותן ביצועים טובים כמו בו שבייצועים יש להם מגבלה חומרת ככלمر לא נוכל להגיע לביצועים של שירות מידע תוך כדי 3 second nano מהdisk אבל יש חשיבות גדולה לכך שהמידע מאורגן בדיסק על הביצועים. רוצים גם אפשר תמייה קריאה וכחיה במקביל של כמה תהליכיים. ועוד דבר זה אבסטרקציה כלומר המכונה לא אמרה להתחשב באם מדובר במכונה של אינטלי או pc-power יש לה זיכרון אבסטרקט מסומלץ. אפליקציה ניגשת לזכרון הזה וקוראת כתובת 60 ולא אכפת לה איפה היא באמת נמצאת, אותו דבר שמדובר בstorage היא אמרה להיות מסוגלת לדיסק לקרוא ולכתוב ולא אכפת לה איך מערכת קבצים זו.

## Achieve our goals through: filesystems

- **Provides abstractions**
  - To help us organize our information conveniently
  - Such that we could easily find & access our data
- **Main abstractions**
  - File
  - Directory (in UNIX), folder (in Windows)
  - Soft links (in UNIX), shortcuts (in Windows)
  - Hard links (same term used in Windows)
  - Standardized by POSIX
- **We will discuss**
  - The filesystem API
  - The abstractions
  - And their implementation

OS (234123) - files

4/2020-12-09 09:00:00

בשביל להגיע לאבסטרקציה זו, קלומר לאפשר למשתמש לא להתעסק במא הולך מתחה השיטה בטרמינולוגיה באיזה חומרה משומש, ואיזה תוכנה מבחינה איזה מערכת קבצים. יש אبني בנין שמאפשרים לעשות האבסטרקציה זו, יש קבצים, תיקיות, ספריות, ויש סטנדרט POSIX.

## File

- **A logical unit of information**
  - ADT (abstract data type)
  - Has a name
  - Has content
    - Typically a sequence of 0 or more bytes  
(we'll exclusively focus on that)
    - But could be, e.g., a stream of records (database)
  - Has metadata/attributes (creation date, size, ...)
  - Can apply operations to it (read, rename, ...)
- **Persistent (non-volatile)**
  - Survives power outage, outlives processes
  - Process can use file, die, then another process can use file

OS (234123) - files

5/2020-12

קובץ זה ייחידה לוגית הכי בסיסית שיכולה לשמר מידע. והוא לא מוגדר באופן חד משמעי כלומר בLINQ ספרייה זה גם קובץ, תמונה זה קובץ. כלומר לכל קובץ יש שם ותוכן ויש לו metadata שמתראר את הקובץ למשל מה הוא הבעלים שלו ואיפה נמצא הנתונים שלו ומה הגודל שלו ומה הרשאות. ומה אנחנו יכולים לעשות זה להפעיל פעולה על הקובץ וכתוצאה לכך הקובץ יכול לשנתנות, כתוצאה לכך אנחנו יכולים לקבל מידע על הקובץ, ומה הקובץ.

## File metadata (a.k.a. attributes)

- Examples
  - Size
  - Owner
  - Permissions
    - Readable? Writable? Executable?
    - Who can read? Who can write? Who can execute?
  - Timestamps
    - Creation time
    - Last time `content` was modified
    - Last time `content or metadata` was modified
  - Physical location on disk
    - Recall that a disk is a “block device”
    - Where do the file’s blocks reside on disk?
  - Type (means various things in various filesystems)
    - E.g., regular file vs. directory

OS (234123) - files

מה זה ?metadata אמרנו שלכל קובץ יש data ו metadata. Metadata זה מהו שמתאר את הקובץ, בהמשך נראה איזה שדות הוא מכיל אבל בכלל הוא יכול הבעלים, הרשאות, גודל, וכל מיני דברים שאומרים מתי הוא נוצר וכו, ובדרך כלל מכיל את הסוג של הקובץ, האם הוא רגיל או ספרייה.

## POSIX file descriptors (FDs)

### A successful open<“file name”> of a file returns a FD

- A nonnegative integer
- An index to a per-process array called the “file descriptor table”
- Each entry in the array saves, e.g., the current offset
- Threads share the array (and hence the offset)

### FD or filename?

- Some file-related POSIX system calls operate on FDs
  - read, write, fchmod, fchown, fchdir, fstat, ftruncate...
- Others operate on file names
  - chmod, chown, chdir, stat, truncate
- Has security implications: FD versions are more secure in some sense
  - Because association of FD to underlying file is immutable
    - Once an FD exists, it will *always* point to the same file
  - Whereas association between file & its name is mutable
  - Using names might lead to TOCTTOU (time of check to time of use) races

OS (234123) - files

2020-12-3

נראה פעולות שאפשר להפעיל על קבצים. חלק גדול מהם ניתן לראות שאפשר להשתמש בשם של קובץ למשל אם נרצה לעשות rename לקובץ “ab” ל “cn”. לעיתים אנחנו הולכים להשתמש ב file descriptor והשאלה במה כדאי להשתמש? בשם למשל לקרוא תוכן של 30 בתים מקובץ “ab” או קודם כדי ליצור file descriptor וביתנו שיש אותו נוכל לקרוא 30 בתים. כלל אצבע אומר שאם יש לנו בחירה, אפשר להשתמש באחד מהם. אבל לא תמיד יש לנו בחירה לפעמים יש לנו מצב סט של קריאות שרק מקבל שם של קובץ או file descriptor. אבל אם יש לנו בחירה כדאי להשתמש ב file descriptor כי הוא לא מחייב רק את הקישור לקובץ אלא גם מה התכוונו לעשות אותו. ריק פתחנו אותו? לקריאה? לכתיבה? וכו'.

## Canonical POSIX file operations

- **Creation** (syscalls: `creat, open;` C: `fopen`)
  - Associate with a name; allocate physical space (at least for metadata)
- **Open** (open; C: `fopen, fdopen`)
  - Load required metadata to allow process to access file
- **Deletion** (`unlink, rmdir;` C: `remove`)
  - Remove name/file association & (possibly) release physical content
- **Close** (`close;` C: `fclose`)
  - Mark end of access; release associated process resources
- **Rename** (`rename;` -)
  - Change associated name
- **Stat** (`stat, lstat, fstat;` -)
  - Get the file's metadata (timestamps, owner, etc.)
- **Chmod** (`chmod, fchmod;` -)
  - Change readable, writable, executable properties

OS (234123) - files

10/2020-12-30 08:47

כמה פקודות לפי סטנדרט Posix.

יש פקודה `create` שמייצרת קובץ, ויש פקודה `open` שפותחת קובץ ובהתקגולים ראיינו שאפ עושים `open` לקובץ לא קיים או הוא פשוט נוצר. בסי יש פונקציה `fopen`. פתיחה של קובץ לשימוש בהמשך, זהה `open`. למחירה של קובץ יש `unlink`, `rmdir` ובסי יש `remove`. ויש את `close` שמנועת שימוש בקובץ שהוא בסי. `rename` שמשנה שם הקובץ. `Stat` מהזירה את metadata של הקובץ. ויש את `chmod` שמאפשרת לשנות הראשות של קובץ.

## Canonical POSIX file operations

- **Chown** (chown, fchown; -)
  - Change ownership (user, group)
- **Seek** (lseek; C: fseek, rewind)
  - Each file is typically associated with a “current offset”
  - Pointing to where the next read or write would occur
  - Lseek allows users to change that offset
- **Read** (read, pread, readv; C: fscanf, fread, fgets)
  - Reads from “current offset”; pread gets offset from caller
  - Need to provide buffer & size
  - “v” = vector of buffer+size; this version is called “scatter-gather” (why?)
- **Write** (write, pwrite, writev; C: fprintf, fwrite, fputs)
  - Change content of file; pwrite gets the offset explicitly; v=vector
  - Likewise, need to provide buffer & size
  - If the current offset (or the given offset in the case of pwrite) points to end of file, then file grows

OS (234123) - files

11 2020-12-30 06:49

יש גם chown שמאפשרת לשנות את owner, group של הקובץ שנראה בהמשך. ועוד פקודות read, write, seek שזכור שאפשר להעביר את file pointer כמו תווים קדימה או אחורה.

## Canonical POSIX file operations

- **Sync** (sync, fsync; C: fflush)
  - Recall that
    - All disk I/O goes through OS “page cache”, which caches the disk
    - OS syncs dirty pages to disk periodically (every few seconds)
  - Use this operation if we want the sync now
  - ‘sync’ is for all the filesystem, and ‘fsync’ is just for a given FD
  - Sync <> fflush; the latter flushes user-space buffers to the kernel
- **Lock** (flock, fcntl; C: flockfile)
  - “Advisory” lock (= processes can ignore it, if they wish)
  - There exists mandatory locking support (in Linux and other OSes)
    - E.g., every open implicitly locks; and can’t open more than once <https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt>
    - But that’s not POSIX

OS (234123) - files

1

יש את sync שבסי היכ והוא פקודה שמאפשרת לכתוב לקובץ מיד. שאנו ננו כותבים לקובץ אז הכתיבה לא מתבצעת מיד אחרית הדיסק היה עמוס מאוד והיה לנו

צואר בקבוק וכל המחשב היה עובד בצורה איטית בגלל שאנו חסום בתוכה כתיבות לדיסק. במקומות אנחנו אוגרים הכתיבה לדיסק, ומתיישה הכתיבה האלה יקרו ביחד, במקום לbezbo הרבה כתיבות לדיסק. אם לא נרצה להמתין לכתיבה אז יש לנו אפשרות לעשות את זה באופן מיידי דרך sync וכך אנו מבטיחים שכתיבות יתבצעו מיד. תנשי לעשות בבית תוכנית די פשוט,

Printf(abc)

Fork

Printf(daf)

מה שנראה זה ש abc תבוצע פעמיים, זה מאייד מזרר כי abc היא הדפסה שקורסית לפני fork. והסיבה לכך היא ש printf כותב לתוך buffer והוא buffer כמו כל נתון אחר של תהליך ab משתקפל ואז הבן חושב שיש לו buffer שmagil abc. וכך משווה זה לא יקרה, אז לפניו fork נוסיף ffflush ל stdout ואז abc תודפס פעם אחת בלבד. ודבר אחרון זה נעליה של קובץ locking ובסי flockfile, לפי סטנדרט יש משווה שנקרא advisor locking כלומר שפותחים קובץ אנו יכולים לבקש לנעול את המניעול ואז אם הוא פתוח אז באמת מצילח לנעול אותו ואם הוא כבר נעול על ידי משווה אחר אז אנו נחסם ונמתין. עכשו למה זה נקרא advisor? כי אנחנו יכולים לפתח קובץ אף אחד לא אמר שהחיבים לנעול את המניעול ואז כך כל עוד כולם משתמשים באותו מנעול יש סינכרון, אבל אם יש למישוה אפליקציה שפותחת את הקובץ ולא משתמשת במניעול אז כל הסינכרון הזה לא עובד. יש דבר שהוא לא סטנדרטי אבל נתמקד ברוב מערכות קבצים שנקרא advisor locking זה בעצם אותו מגנון של mandator locking רק שהוא קורה דרך מערכת הפעלה זו שאנו מבקשים לפתח את הקובץ מערכת הפעלה מנסה לנעול עכורות את המניעול ואז כל פתיחה של קובץ ללא יוצא מן הכלל גורמת לכך שמנעול הזה נבדק וכך אנחנו לא נعبر את הסינכרון. (כלומר יבוד היבט בלי טובת המשמש). נשים לב לא לכל הקבצים משתמשים ב locking בכלל עברו רוב פתיחות קבציםanno לא מקמפים אותם לשימוש ב advisor, mandator locking .

## File types

- **Some systems (not Unix/POSIX) distinguished between**
  - Text & binary
- **Some older systems decided type by name extensions**
  - In DOS, executables have the extensions: com, exe, bat
- **In UNIX (POSIX), types are**
  - Regular file
  - Directory
  - Symbolic link (= shortcut), a.k.a. soft link
  - FIFO (named pipe)
  - Socket
  - Device file
  - See: [http://en.wikipedia.org/wiki/Unix\\_file\\_types](http://en.wikipedia.org/wiki/Unix_file_types)

OS (234123) - files

14 2020-12-30 0

ה הפרדה ה/cgi בסיסית היא האם קובץ הוא בינהרי או טקסטואלי. מה ההבדל ביניהם? קובץ טקסט הוא מוקוד עד ידי טבלת אסקוי ז"א אם נראה בכתבתו TO A ואז הולכים לטבלת אסקוי ואם רואים שמתאים לו 97, אז כותבים 97 לתוך הקובץ. הקידוד נעשה דרך טבלת אסקוי, כל שאר הקידודים זה קידוד שהם בינהרים ודוגמאות לקבצים כאלה למשל קבצים שעברו קומפלציה. למרות שלא שומרם שם טקסט אבל אפשר להגיד שהם הם מקודדים על ידי טבלת אסקוי. אבל דוגמאות יותר פשוטות שום ויכוח שהם קידוד אסקוי או לא למשל doc, zip, rae, pmg כל מה שקורחה שם זה לא טקסט שמקודד בטבלת אסקוי. הסיווג הזה הוא גס מאוד כי כמעט רוב הקבצים במערכת הינט לא קבצים שמקודדים על ידי אסקוי, כמעט כולם בינהרים. פעם השתמשו בתורה ידי כבده במשהו שנקרא extension כלומר פורמט של שם של הקובץ היה מילה ואז נקודה ואז extention שบทחלתה היה בגודל 3-2 אותיות אחר כך 3-4 אותיות ואז נהייה כללי. מערכות הפעלה בעבר האמינו בצורה עוורת ל extension וזה בעיה כי אם משנים את extension או זה יהיה משנה סוג של הקובץ וזה לא יהיה הגיוני. לפי סטנדרט של posix socket, fifo, ספרייה, גיל, ועוד כל מיני.

## Magic numbers in the UNIX family

- A semi-standard Unix way to tell the type of a file
  - Store a "magic number" inside the file itself
  - Originally, first two 2-byte => only  $2^{16}$  => not enough
  - Nowadays, much a more complex scheme
- Examples
  - Every GIF file starts with the ASCII strings: *GIF87a* or *GIF89a*
  - Every PDF file starts with the ASCII string: *%PDF*
  - Script files start with a "shebang", followed by an executable name, which identifies the interpreter of the script; the shell executes the interpreter and feeds the script to it as input ("#" indicates comment for the interpreter, so it ignores this line)
    - `#!/usr/bin/perl -w`  
# Perl code here...
    - `#!/usr/bin/py`  
# Python code here...

OS (234123) - files

15/2020-10-09 10:09:10

דרך נפוצה לאפיין טיפוס של קובץ זה מספר קסם. זה היה די לא נוח ודי מעט ולכון במקומות התחליו להשתמש בmachroozot שאומרת כל הדברים על הקובץ למשל כל קובץ pdf מתחילה ב *%PDF*, קובץ GIF מתחילה ב *GIF87a* או *GIF89a*

תלוי בסוג של ה-GIF. כל מיני דברים שקשורים לshell, python, perl מתחילה בmachroozot שאומרת איזה אפליקציה להריץ למשל בפייתון ניתן לראות לעלה כדי להפעיל פiython.

## Magic numbers in the UNIX family

- Pros
  - File's content, rather than metadata, determines what this file is
  - (Metadata like the file's name might be altered independently of the content, potentially erroneously)
- Cons
  - Magic logic became fairly complex
  - Somewhat inefficient because
    - Need to check against entire magic database, and
    - Need to read file content rather than just metadata
- More details
  - [http://en.wikipedia.org/wiki/File\\_format#Magic\\_number](http://en.wikipedia.org/wiki/File_format#Magic_number)
- Helpful – the 'file' utility
  - A shell utility that, given a file, identifies its type
  - <http://linux.die.net/man/1/file>

OS (234123) - files

1

מצד אחד זה נכון, זה מאפשר לזרף את הסוג של הקובץ לתוכן של הקובץ וכך לא משנה איך אנחנו קוראים לקובץ זהה, או לא משנה لأن מעתיקים כי אפשר לדעת הטיפיס האמתי של הקובץ. החיסרונו שהוא לא ניתן להרחבה. נניח שאנו ממצאים פורמט חדש # משהו. אין גורמים לכל מערכות הקבצים להכיר אותו? אין איזשהו מנגן שמעדכן את מערכות הקבצים או רשיימה של טיפוסים מוכרים אז כן אם הפורט שלנו יהיה בשוק הרבה זמן ופופולרי אז כל מערכת הקבצים תעשה תיקון ואת העדכון שלו בשביל לתמוך בפורט הזה, אבל אין איזשהו דרך לעשות push לupdate הזה. וכך אם נסתכל על רשימה של סוגי קבצים שנתמכים בצורה נזאת הרשימה הזאת תהיה די סטטית ולא מתעדכנת.

תזכורת:

1. User: the owner of the file (person who created the file).
2. Group: the group can contain multiple users. Therefore, all users in that group will have the same permissions. It makes things easier than assign permission for every user you want.
3. Other: any person has access to that file, that person has neither created the file, nor are they in any group which has access to that file.

In Linux, a group is a **collection of users**. The main purpose of the groups is to define a set of privileges like read, write, or execute permission for a given resource that can be shared among the users within the group. Users can be added to an existing group

## ■ POSIX file protection: ownership & mode

- **Motivation**
  - In a multiuser system, not everyone is allowed to access a given file
  - Even if they're allowed to "access", we don't necessarily want to allow them to perform every conceivable operation on that file
- **For each file, POSIX divides users into 3 classes**
  - "User" (the owner of the file), "group", and "all" the rest
  - (Users can belong to several groups; see: man 2 getgroups)
- **POSIX associates 3 capabilities with each class**
  - Read, write, and execute
- **Hence, each file is associated with**
  - $3 \times 3 = 9$  class/capabilities => this is called the "file mode"
  - Mode is controlled by the (f)chmod syscall
  - Ownership (user & group) is controlled by the (f)chown syscall

OS (234123) - files

ראינו בתרגולים שלכל קובץ יש 3 סוגי של הרשות, כל self, group, universe משתחמש שיש יכול להשתיק לכמה קבוצות שהוא רוצה למשל, group1, group2, group3 עבור כל קובץ יש owner שלו שיכול להיות user ולאיזה קבוצה יש הרשות מוחדרת. הקבוצה הזאת חייבת להכיל את user למשל הקבוצה יכולה להיות group2 וה קונפגרציה תקנית, אבל הקונפגרציה של קובץ שמשתיך ל user ו הרשות מוחדרת ל group4 זאת קונפגרציה לא חוקית כי user אינו(group4). נשים לב שיש לנו 4 קבוצה קבוצה ראשונה זה לכתב לקרוא, שנייה רק לקרוא, שלישית רק לכתוב וuser יכול להשתיק לכמה מהם.

## POSIX file protection: ownership & mode

- Stat returns these 9, and 'ls -l' displays them
  - 10 “bits” are displayed
  - Leftmost is the “type”
  - Remaining 9 bits are read/write/execute X user/group/all

|             |   |         |       |        |        |        |               |           |
|-------------|---|---------|-------|--------|--------|--------|---------------|-----------|
| brw-r---r-- | 1 | unixguy | staff | 64,    | 64     | Jan 27 | 05:52         | block     |
| crw-r---r-- | 1 | unixguy | staff | 255    | 255    | Jan 26 | 13:57         | character |
| -rw-r--r--  | 1 | unixguy | staff | 290    | Jan 26 | 14:08  | compressed.gz |           |
| -rw-r--r--  | 1 | unixguy | staff | 331836 | Jan 26 | 14:06  | data.ppm      |           |
| drwxrwx---x | 2 | unixguy | staff | 48     | Jan 26 | 11:28  | directory     |           |
| -rwxrwx---x | 1 | unixguy | staff | 29     | Jan 26 | 14:03  | executable    |           |
| prw-r--r--  | 1 | unixguy | staff | 0      | Jan 26 | 11:50  | fifo          |           |
| lrwxrwxrwx  | 1 | unixguy | staff | 3      | Jan 26 | 11:44  | link -> dir   |           |
| -rw-rw----  | 1 | unixguy | staff | 217    | Jan 26 | 14:08  | regularfile   |           |

OS (234123) - files

18

למטה רואים שיש קובץ רגיל שנקרא regularfile והוא שייך ל user unixguy .staff . והוא שפирיה לספרייה .staff . הוא עודכן פה אחרונה ב 25 לינואר וגודלו 217 בית . וניתן לראות ש lsuser ניתן לקרוא ולכתוב ו אסור להריצ'ן ול staff מותר לקרוא לכתוב ולא להריצ'ן . ולכל השאר אין גישה לקובץ . דבר הבא שמשמעותו זה directory שזה ספרייה בשורה 5 ניתן לראות ש מתחילה ב d שזה מסמל ספרייה . וההרשאות שלה read write execute ל unixguy , staff ו רק execute לשאר . עכשו מה זה read write execute execute ל ספרייה ? נתחיל מ read , היא נותנת לנו לעשות ls כולם הרשאות לדעת איזה קבצים יש בספרייה . מה נותן לנו execute ? לעשות cd כולם לקרוא הקובץ בתחום הספרייה או תחת הספרייה . למשל אם לנו משתמש שהוא לא ב staff אז מותר לעשות לו רק execute . והמשתמש הזה הוא לא unixguy כי רואים ש unixguy יש לו את כל הרשאות . מה עושה write ? זה לעשות עדכון של ספרייה כלומר להוסיך אליה קובץ או להוסיך עוד ספרייה בתחום הספרייה או למחוק . אחר כך רואים executable שזה קובץ רגיל , וההבדל בין קבצים אחרים זה שיש אפשרות להריצ'ן אותו . Fifo זה מה שלמדנו בתרגולים (תרגול של pipe ) שיצרים בעזרתו fifo . make . הדבר הבא זה link רואים יש 1 בתחלת הקובץ שמדובר עליו אחריו . fifo . תזכורת ל

A FIFO special file **sends data from one process to another** so that the receiving process reads the data first-in-first-out (FIFO). A FIFO special file is also called a named pipe, or a FIFO . A FIFO special file can also be shared by a number of processes that were not created by forks.

נשים לב שם block device, character device שראינו גם בתרגולים ויש להפ שני מספרים שהוא major, minor שם ראיינו בתרגולים. מה זה אומר שיש אותו major בהנחה שיש לנו שני קבצים מסוג block למשל block1, block2 operations operation במערך chdrev ויקבלו אותם אולם אם המ מטיפוסים שונים ויש להם אותו major אז יש להם מערכים שונים וככלום לא אותו דבר למשל למעלה block, character devices יש להם אותו אינדסטרה במערך.

נשים לב יש מערך לblock devices,block devices ומערך לcharacter devices

## Access control lists (ACLs)

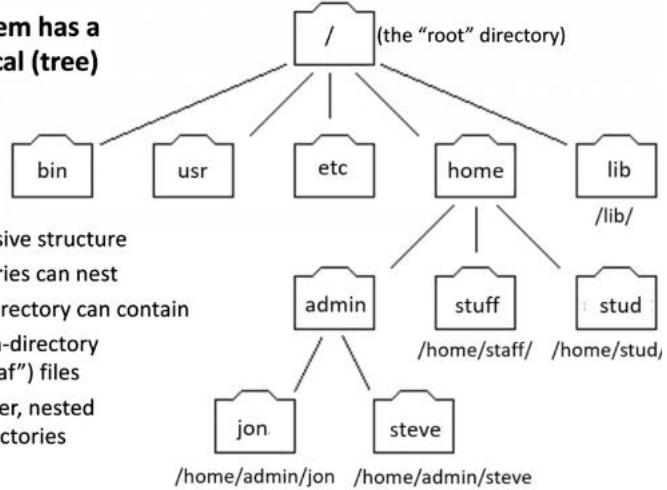
- OSes can support a much finer, more detailed protection
  - Who can do what
- Most OSes/filesystems support some form of ACLs
  - Many groups/users can be associated with a file
  - Each group/user can be associated with the 3 attributes (r/w/x)  
<http://static.usenix.org/events/usenix03/tech/freenix03/gruenbacher.html>
  - Or more, finer attributes (“can delete”, “can rename”, etc.)
- Con: not part of POSIX
  - Effort to standardize ACLs abounded in Jan 1998
    - Participating parties couldn’t reach an agreement...
  - Hence, it’s hard to make programs that use ACLs portable
    - (Recall: a program is “portable” across a set of OSes if it works on all of them without having to change its source code; in particular, a program that adheres to POSIX works unchanged in all OSes that comply with POSIX = the UNIX family: Linux, AIX, Solaris, FreeBSD, Mac OS, ...)

מה ניתן לעשות עם הקובץ?

אנו רוצים למשל להגיד שלאוnid מותר לעשות משהו ומוחמד יכול לעשות משהו לקובץ כלומר לכל קבוצה יש ACLs ובכל אחד כזה יש user או קבוצה ומה מותר לעשות. למה צריך ACLs? נניח שיש לנו קובץ ואני רוץ לחת לכל מי שעושה מthem ומערכות הפעלה לקרוא אותה או ACLs לקרוא אותה.

## Directories

- A filesystem has a hierarchical (tree) structure



## POSIX directory operations

- mkdir(2)** – create empty directory
- rmdir(2)** – remove empty directory (fails if nonempty)
- Special directory names (relative paths that always exist)**
  - Current directory “.”
  - Parent directory “..” (root is its own parent)
  - Hidden by default by ‘ls’, as start with a “.”
- Dir content traversal – opendir(3), readdir(3), closedir(3)**

*FILE* \*

```

struct dirent *de;
DIR *p = opendir("..");
while((de = readdir(p)) != NULL)
 printf("%s\n", de->d_name);
closedir(p);

```

- What do we need to do to make it recursive?

ה抽象ציה שלמדנו עד עכשיו זה קובץ. עכשו נלמד ספרייה, ספרייה עצמה זה מהهو שמאפשר לבנות היררכיה או עץ של תיקיות שבעץ הזה עלים הם קבצים וצמחיים פנימיים זה ספריות רגילים. שפונים לקובץ ציצים מתחת שם מלא למשל / או מסלול יחסית לספריית העבודה. מבחינת הספרייה. ספרייה זה קובץ לכל דבר, הדבר המיעוד שיש לה זה שהתוכן שלה הוא בפורמט זה סוג של רשימה מקוורת של dirent שהוא סטרukt אחד השדות בו שנקרא `d_name` שהוא שם קובץ. לדוגמה אם רוצים להדפיס את התוכן

של הספרייה או יש פקודות Si שעוזרות לקרוא תוכן של ספרייה בדרך נוחה `opendir` של ספרייה בדרכָה `dir` ו`file`\* שהוא מציין לסדרקט `dirent` עד שmagics לסקור הרשימה מהזירה `null` וכך מדפיסים את כל הקבצים בספרייה.

## Symbolic links (soft links)

- **Unlike hard links**
  - Which point to the actual underlying file object
- **Symlinks (“shortcuts” in Windows terms)**
  - Point to a *name* of a “target” file (their content is typically this name)
  - They’re not counted in the file’s ref count
  - They can be “broken” / “dangling” (point to a nonexistent path)
  - They can refer to a directory (unlike hard links in most cases)
  - They can refer to files outside of the filesystem / mount point (whereas hard links must point to files within the same filesystem)
- **When applying a system call to a symlink**
  - The system call would seamlessly applied to the target file
  - For example, `open()`, `stat()`, `chmod()`, `chown()`, ...

OS (234123) - files

מה זה URL או shortcut בוינדוס ומה יש בתוכו? בתוכו רשום מסלול של קובץ, אז UI של ווינדוס פותח את הקובץ וקורא משם את המסלול והולך פותח קובץ אחר. בלינוקס יש משהו שנקרא soft link שהוא גם מכיל מסלול לקובץ אחר אבל שעושים פתיחה זה לא רק UI של לינוקס גם מערכת הפעלה, היא פותחת softlink ואז תפתח את הקובץ שעליו היא מצביעה.

## Symbolic links – example

```
<0>dan@csa:~$ echo hey > f1
<0>dan@csa:~$ cat f1; # content of f1
hey
<0>dan@csa:~$ ln -s f1 f2; # f2 is a symlink to f1
<0>dan@csa:~$ ls -l f2
lrwxrwxrwx 1 dan 2 Jun 10 07:56 f2 -> f1 # notice arrow & perms

<0>dan@csa:~$ cat f2; # content of f1
hey

<0>dan@csa:~$ rm -f f1; # f1 no longer exists
<0>dan@csa:~$ cat f2;
cat: f2: No such file or directory # so the 'cat' fails
```

OS (234123) - files

3

.דוגמא.

קודם כל מדפיסים hey לתוך קובץ f1 ואז עושים לו .cat.  
עכשו מיצרים softlink לעושץ זאת משתמשים בפקודה s - ln ומיצרים f2 שהתוכן  
שלו מצביע ל f1. עכשווונאם הולכים ומוחקים את f1 ועכשו ננסה לעשות ל f2 א  
נראה ש אין קובץ זה.

## Hard links – intro

- **File != file name**
  - They are *not* the same thing
  - In fact, the name is not even part of the file's metadata
  - A file can have many names, which appear in unrelated places in the filesystem hierarchy
  - Creating another name => creating another "hard link"
- **System calls**
  - link( srcpath, dstpath )
  - unlink( path )
- **Shell**
  - ln <srcpath> <dstpath>
  - rm <path>

OS (234123) - files

2

משהו שעד יחסית לא מזמן לא היה קיים בכלל ב windows היום יש תמייה בזה במערכת קבצים כלשהי תומכת בזה. בעצם אם נסתכל על קובץ אז הוא מכיל data ו metadata אבל איפה נמצא שם של הקובץ? שם של הקובץ נמצא בספרייה שעשויה מיפוי למשל תוכנן של תיקיה A זה f1 ממופה ל metadata שלו. f2 ממופה ל metadata שלו. אז אין סיבה בעולם שלא יהיה שתי ספריות שונות לא יוכל למלאות אותו קובץ או לחולופין אותה ספריה למפות אותו קובץ בשמות שונים. למשל כך,

## Hard links – intro

- **File != file name**
  - They are *not* the same thing
  - In fact, the name is not even part of the file's metadata
  - A file can have many names, which appear in unrelated places in the filesystem hierarchy
  - Creating another name => creating another “hard link”
- **System calls**
  - `link( srcpath, dstpath )`
  - `unlink( path )`
- **Shell**
  - `In <srcpath> <dstpath>`
  - `rm <path>`



OS (234123) - files

24 2020-12-30 09

בעצם שקובץ נוצר לקובץ זהה קישור מספרייה לקובץ. למשל נניח שמייצרים קובץ f17 או הוא יהיה hard link ראשון של קובץ וכולם אותו חשיבות יש להם והקובץ נשאר במערכת אם יש לו אחד להבדיל softlinks שיכול להימחק קובץ גם אם יש עליו .softlink 100

## Hard links – example

```
<0>dan@csa:~$ echo "hello" > f1
<0>dan@csa:~$ cat f1
hello
<0>dan@csa:~$ ls -l f1
-rw-r--r-- 1 dan 6 Jun 10 06:48 f1 # "1" = how many links to the file
<0>dan@csa:~$ ln f1 tmp/f2; # ln <src> <dst> creates the link
<0>dan@csa:~$ cat tmp/f2;
hello # f1 & f2 are links to same file
 # so they have the same content
<0>dan@csa:~$ ls -l f1; # 'ls -l' reveals how many links:
-rw-r--r-- 2 dan 6 Jun 10 06:48 f1 # 2 links
<0>dan@csa:~$ echo "goodbye" > f1; # override content of f1
<0>dan@csa:~$ cat tmp/f2; # content of f2 also changes
goodbye
```

OS (234123) - files

25

נשים למעלה שעושים כותבים לקובץ f1 וואז עושים פעמים 1 hard link, או 2hard link, או 3hard link וכו'. אחד מהם זה f1 והשני זה tmp/f2. למשל אם נשנה תוכן f1 או התוכן של tmp/f2 ישנה.

## Directory hard links

- **A noted, most filesystems don't support directory hard links**
  - HFS+ is an exception
- **Still, all filesystems that adhere to POSIX provide at least some support to directory hard links**
  - Due to the special directory names “.” and “..”
  - What's the minimum number of hard links for directory?
    - 2 (due to “.”)
  - What's the maximum?
    - Depends on how many subdirectories nest in it (due to “..”)

מבחינת הספריות, רוב מערכות קבצים חוסמות אפשרות לעשות hard link בספרייה. אף ננסה לעשות hard link בספרייה או הפעולה תיכשל. יחד עם זאת הן יכולות בhard link בספרייה כי לכל ספרייה יש לפחות שני hard link. למשל נkeh ספרייה

A. אז יש לה hard link לספרייה עצמה ושתי נקודות זה נקודה (נקודה). אז אם נסתכל על שתי ספריות A,B כך ש B בן של A בצורה הבאה,

## Directory hard links

- A noted, most filesystems don't support directory hard links
  - HFS+ is an exception
- Still, all filesystems that adhere to POSIX provide at least some support to directory hard links
  - Due to the special directory names "." and ".."
  - What's the minimum number of hard links for directory?
    - 2 (due to ".")
  - What's the maximum?
    - Depends on how many subdirectories nest in it (due to "..")

OS (234123) - files

28

כמה ייש לhard link ל-A? אחד לאבא שלו, ואחד נקודה שזה לא ועובד כל אחד מהתספריה שלה חמשל לה זה B אז יש שלושה hard link.

## Implementation – inode

- The OS data structure that represents the file
  - Each file has its own (single) inode (= short for "index node")
  - You can think of the inode as the true representative of "the file" or "the file object"
  - Internally, file names "point" to inodes
    - This inode is determined via the path-resolution algorithm (later)
- The inode contains all the metadata of the file
  - Timestamps, owner, permissions, ...
  - Pointers to the actual physical blocks, on the drive

כעת נדבר על inode. שזה אובייקט שנמצא בדיסק שמכיל את כל ה meta data לכל קובץ יש inode וmeta data שתפקידו זה לספק מידע על הקובץ שהוא גדול והרשאות ואיפה נמצא ה data של הקובץ.

## inodes & \*stat

- **stat(2) & fstat(2) retrieve metadata held in inode**

- POSIX promises that at least the following fields are found in the stat structure and have meaningful values

- |                |                                                     |
|----------------|-----------------------------------------------------|
| 1. st_dev      | ID of device containing file                        |
| 2. st_ino      | inode number, unique per st_dev                     |
| 3. st_mode     | specifies file type & permissions (S_ISREG(m), ...) |
| 4. st_nlink    | number of hard links to the file                    |
| 5. st_size     | file size in bytes                                  |
| 6. st_uid      | user ID of owner                                    |
| 7. st_gid      | group ID of owner                                   |
| 8. st_ctime    | last time metadata (=inode) or data changed         |
| 9. st_mtime    | last time data changed                              |
| 10. st_atime   | last time metadata (=inode) or data accessed        |
| 11. st_blksize | block size of this file object                      |
| 12. st_blocks  | number of blocks allocated for this file object     |

- Why do we need st\_size as well as st\_blksize & st\_blocks?

אלו השדות בתוך ה inode. למשל מספר שמוזה את ה device עליו נמצא ה inode. באתה מחשוב יכולם להיות כמה deviceים וכמה מערכות קבצים וצריכים לדעת על איזה inode זה יושב. יודע מה מספר inode שלו שהוא מספר ייחודי על אותו device. רק אחד עם מספר כתשחו על inodedevice כמה שונים יכולים להיות כמה inode. יש את mod שנותן את הרשאות על הקובץ. גודלו, מתי נוצרו, מתי עודכן ה inode. גודל של בלוק ומספר של בלוק, אנחנו לא שומרים בדיסק chunks של meta data בתים או 30 אלא יש גודל מסוים של chunks שיודעים לשמר בקובץ למשל 20 512 בית. יש שני שדות שהם אחד מהם זה גודל של chunk ומה בלוקים מוקצתה לקובץ.

## inodes & \*stat

- **Istat(2)**

- Exactly the same as stat(2) if applied to a hard link
- But if applied to a symlink, would return the information of this symlink (*not to the target of the symlink*)
- In this case, POSIX says that the only fields within the stat structure that you can portably use are:
  - st\_mode which will specify that the file is a symlink
  - st\_size symlink content length (= length of target filepath)
- The value of the rest of the fields *could* be valid, but it is not specified by POSIX
- Notably, it is not specified if a symlink has a corresponding inode
  - Will be discussed shortly

יש כל מיני פקודות stat, lstat שמאפשרות לקרוא את metadata של הקובץ.

## Implementation – directory file

### A simple flat file comprised of directory entries

- For example, it could be a sequence of

```
struct dirent {
 ino_t d_ino; /* POSIX: inode number */
 off_t d_off; /* offset to next dirent */
 short d_reclen; /* length of this record */
 char d_type; /* type of file */
 char d_name[NAME_MAX+1]; /* POSIX: null-term fname */
 /* Must we always use NAME_MAX+1 chars? No! */
};
```

### Importantly, note that

- The name of a file is **not** saved in the inode  
(recall: there can be many names/hardlinks associated with one file)
- Rather, it is stored in the directory file as a simple string, in d\_name
- If the file is a symbolic link
  - The target file can be retrieved from the contents of the symlink

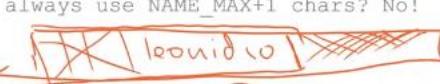
אמרנו שכל ספירה מורכבת מרשימה מקושרת של סטרוקטים. כל סטרuktur מכיל dirent רשימה הbhא ומה גודל של הרשימה הזאתו גם d\_inode את מספר ה inode של הקובץ וגבר אחרון שהוא טיפיס (ספרייה, קובץ רגיל והלאה). שאלה? למה שמים dirent הוא נמצא ב inode, כלומר יודע מה הטיפיס

של הקובץ. הסיבה היא לעשות רקורסיה בצורה עילית אחרת היינו חיבים לסדר קצר כלומר אחרת היינו הולכים ל dirent ו נגישים ממנו ל inode והוא יודע הטיפוס שלו. עכשו שאלת יותר קשה? מה גודל של סטרuktur זהה בערך? 270 בית בערך, בעצם עבור כל entry בספרייה אנו צריכים להקצות 270 בתים. מה היה קורה אם ספרייה מכילה 10 אלף קבצים, למשל אם נרצה למצוא קובץ f1 בתחום אותה ספרייה אני עוברים entry אחד אחרי השני עד שמוסאים directory שבשדה d\_name מופיע f1. לכן אנו לא נרצה לשמר הרבה מידע עבור כל קובץ, חלק מהמידע חיבים לשמר כי אין בקירה אבל איזה מהשדות לעלה נראה מיותר? השם פה תופס הרבה בתים. אז מה עושים בפועל? הוא תופס 255 בתים אבל בכלל שם של הקובץ הוא ידי פחות (מש יותר ממה שנדרים ממוצע שם קובץ זה 15-10 תווים, אבל מן הסתם לא יכולים להגיד דבר כזה). אז מה עושים?

אנחנו מקצים dirent וממלאים השגות שאורך שלהם קבוע ואז ממלאים את השם, נניה שימושים ב15 בתים או כתובים לשדה d\_reclen את ה 15 ולתול הדיסק כתובים רק את ה 15 בתים.

## Implementation – directory file

- A simple flat file comprised of directory entries
  - For example, it could be a sequence of
 

```
struct dirent {
 ino_t d_ino; /* POSIX: inode number *
 off_t d_off; /* offset to next dirent *
 short d_reclen; /* length of this record *
 char d_type; /* type of file *
 char d_name[NAME_MAX+1]; /* POSIX: null-term fname *
 /* Must we always use NAME_MAX+1 chars? No! *
};
```
- Importantly, note that 
  - The name of a file is **not** saved in the inode (recall: there can be many names/hardlinks associated with one file)
  - Rather, it is stored in the directory file as a simple string, in d\_name
  - If the file is a symbolic link
    - The target file can be retrieved from the contents of the symlink

אמרנו שספריית הקבצים זה רשימה מקושרת של סטרקטרים, עכשו מה הבעיה? רואים בהתחלה שמלאים כל השדות שגודל שלהם קבוע. ואז ממלאים את השם נניה leonid, ואז נשאר הרבה מקום לא בשימוש. איך נדע לגשת לאיבר הבא ברשימה מקושרת,

## Implementation – directory file

- A simple flat file comprised of directory entries

– For example, it could be a sequence of

```
struct dirent {
 ino_t d_ino; /* POSIX: inode number */
 off_t d_off; /* offset to next dirent */
 short d_reclen; /* length of this record */
 char d_type; /* type of file */
 char d_name[NAME_MAX+1]; /* POSIX: null-term fname */
};
/* Must we always use NAME_MAX+1 chars? No! */
```

- Importantly, note that

– The name of a file is **not** saved in the inode  
(recall: there can be many names/hardlinks associated with one file)  
– Rather, it is stored in the directory file as a simple string, in `d_name`  
– If the file is a symbolic link

- The target file can be retrieved from the contents of the symlink

נניח כמו שראויים למעלה שבהתחלת יש אובייקט בגודל 15 ואחריו בגודל 20 ואחריו בגודל 35. אז אם למשל נרצה לגשת לאיבר בגודל ב35 הולכים לראשון ובודקים מה יש בשדה `d_reclen` רואים שבו יש 15 ואחר כך איבר שני גם בתול `d_reclen` שלו יש 20 וכן מدلגים 35 ומתקבלים איבר שרצינו. זה טוב אבל יש בעיה, למשל אם נמחק האיבר השני, אז מה שנעשה משתמש ב `offset` אז מה שנעשה זה נעדכן את ה `d_offset` של האיבר הראשון להיות 35 וזה בעצם התפקיד שלו.

זה שיטה די טובה לפחות רשיימה מקוורת ולכזווים גודל של ספרייה, כי היפוש קובץ מהספרייה צריך לעבור על חצי מה `entries` ולעשות זאת צריך להביא אותם מディיסק כדי לקרוא אותם וזה יהיה איטי. לכן הדבר שיכolumn לעשות זה שגודל הספרייה יהיה קטן ככל möglich.

נשים לב שבתרשים למעלה אף יש. לנו `entry` שה `type` שלו זה `soft link`, מה שצרכים לעשות זה למצוא `entry` זהה בספרייה ואז הוא אומר איפה נמצא metadata של ה `soft link` והוא אומר לנו איפה נמצא metadata של ה `data` של ה `soft link` ועכשו ניגשים אליו וקוראים ה `data` שם יהיה רשום לאיזה קובץ ה `soft link` זה מצבע ואז לוקחים מה רשום שם ומה תחילה לעשות היפוש מחדש. אנחנו יכולים לתקן את זה על ידי אופטימיזציה על יד הכנסת את הקובץ המטרה שעליו `softlink` מצבע בשדה `d_name`. ז"א הוא יכול גם את של `soft link` וגם את הקובץ שעליו מצבע.

הערה: זה מבנה של מערכת הקבצים שישב על הדיסק ומיה שקורא אותה זה הדריבר של מערכת הקבצים.

## Path resolution process

- **Resolving a path**
  - Get a file path
  - Return the inode that corresponds to the path
- **All system calls that get a file name as an argument**
  - Need to resolve the path
- **To this end, all these system calls use the same algorithm**
  - Which is oftentimes called “namei” (internally, within the kernel)
- **The namei algorithm consists of O(n) steps, where...**
  - “Atomic name”
    - = a filepath that doesn’t contain a slash (“/”)
    - = a name component
  - n = number of atomic name components that comprise the file path
  - Including the components that comprise the symlinks along the path
    - Recursively speaking

OS (234123) - files

3

מה זה.path resolution? איננו מעבירים מסלול לקובץ, למשל נרצה לפתח קובץ /use/hw/gge.txt

או איך מערכת הפעלה מגיעה לקובץ זה? איך האלגוריתם עובד? קודם כל אנחנו שוברים את המסלול לאוטומטיים ואנחנו מפעילים פונקציה רקורסיבית עליהם.

## Path resolution process

To illustrate, here's a simplistic pseudo code version of a user-mode component-by-component open

```
#define SYS(call) if((call) == -1) return -1

int my_open(char * fname) {
 if(fname is absolute) chdir("/") + make fname relative
 foreach atom in fname do // atoms of "x/y" are "x" and "y"
 if(is symlink) SYS(fd = my_open(atom's target))
 else SYS(fd = open(atom /*checks perm!*/))
 if(! terminal) SYS(fchdir(fd) + close(fd))
 else break
 }
 return fd
}
```

OS (234123) - files

40 2020-12-30

הנה קוד שמתאר האלגוריתם, קודם כל מסתכלים אפ' המסלול מתחילה ב "/" אם כן אנחנו עוברים למספרייה "/" אחרת אנחנו נשארים בספריית העבודה. עכשיו עוברים על כל טחן מאטומים שזה usr, hw, gg.txt פותחים את האטום ובודקים איזה הרשות יש עליו (האם ניצן לפצוח אותו) והאם מדובר באופן מיוחד למשל אם הוא soft link אז מפעילים בצורה רקורסיבית את הפונקציה open על ה soft link כי קודם כל צריך לפתחו מב ש כתוב ב soft link ואם לא אז זה חייב להיות ספרייה (לא יתכן שזה קובץ רגיל, כי אם כן אז מקבלים תקלה) אחר כך נוכנסים לספריית user וסורקים אותה ומוצאים את ה entry המתאים לו hw אם מצאנו אז ממשיכים למצוא בה entry שמתאים לו gg.txt ומסיימים.

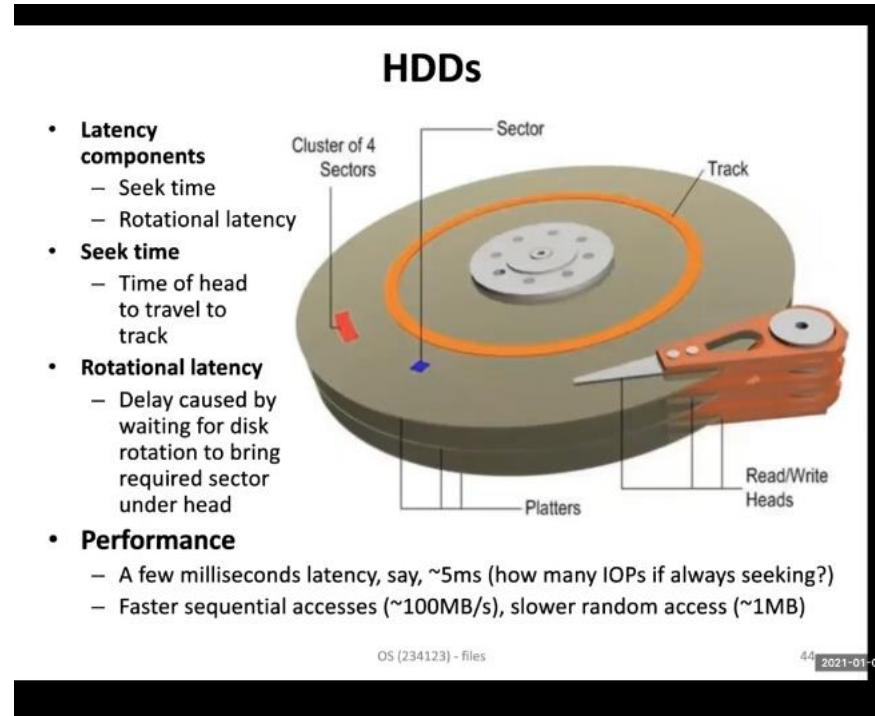
ממבט ראשון זה נשמע שזה אלגוריתל שעבוד בזמן לינארי, קודם כל אם יש לי אטום שהוא soft link אז האלגוריתם כבר לא יהיה לינארי, אפילו בלי ההנחה הזאת האלגוריתם הוא לא לינארי, כי כל פעם סורקים entries עברו כל אטום ולכל שלספרייה מכילה יותר קבצים אז הפעולה של הבדיקה תהיה יותר כבדה, لكن בעצם הביצועים של האלגוריתם תלויים לא רק במספר האטומים אלא גם בגודל הספריות וזה שהוא שcharיך לחתה בחשבון.

משיכים לדבר על מערכות קבצים.

איך נראה חומרה?



היום נכנס לימוש של מערכות קבצים וזה יהיה התוכן של שיעור היום. משחו שלא בחומר הקורס ולא הפקוס של מערכות הפעלה, זה בעצם איך נראית החומרה. אז באופן כללי כך. ראה דיסק קשיח של המחשבים הנוכחיים, מה הסתם הגиск הוא סגור והוא נפתח ניתן לראות אותו.



המבנה של הדיסק זה דיסקיות מגנטיות ולכל דיסקית מגנטית יש מראש קורא. (להסתכל ביווטיב איך הוא עובד) לכל דיסק קוראים plator ומסלול נקרא track ואז הוא קורא. היחידה הבסיסית שאפשר לקרוא זה sector שהיא 512 ביט. בדרך כלל כל קריאה או כתובה היא ברמה של clastor ולא sector שהוא בגודל 8KB4.

## Example – enterprise level (mid 2013)

- **Seagate Cheetah 15K.7 SAS; by spec:**

- “Highest reliability rating in industry”
- Interface: SAS (Serial Attached SCSI; rotates faster and is pricier than SATA’s 5.4K/7.2K RPM HDDs)
- Price: \$260 (@ amazon.com)
- Spindle speed 15K (RPM = rounds per minute)
- Capacity: 600 GB
- Cache: 16MB (DRAM)
- Form factor: 3.5 inch (laptops have 2.5" or even 1.8" HDDs)
- Throughput: min=122 MB/s – max=204 MB/s
- Avg rotational lat.: 2.0 ms
- Typical avg seek time 3.4 ms (read) 3.9 ms (write)  
Typical single track seek 0.2 ms (read) 0.44 ms (write)  
Typical full stroke seek 6.6 ms (read) 7.4 ms (write)
  - “Full stroke” = head moves from outer to inner portion of the disk (max head motion); “single track” = move from track n to n±1
- Error rate < 1 in  $10^{21}$



דוגמא לדיסק ניצן לראות חמעלה והוא יחסול יחסית וקטן ומהיר יחסית, (קצב סיבוב שלו זה 15.7 אלף סיבובים לדקה שווה די מהיר). הדיסקים הרגילים ששמים במחשבים קצב סיבוב שלהם נע בין 5400-7200 סיבובים לדקה.

### (Anecdote: why “3.5 inch form-factor”?)

- 3.5" = 8.89 cm
- But standard dimensions of a 3.5 inch disk are
  - Length : 5.79" ≈ 14.7 cm
  - Height: 1.03" ≈ 2.61 cm
  - Width: 4.00" = 10.16 cm



במחשבים ניידים יש דיסקים אחרים שנקראים 2.5 inch אבל של המחשב נិיח נקרא 3.5 אינץ' וניתן לבדוק שאם מימד של הדיסק הוא 3.5 אינץ', לא רוחב ולא גובה ולא כלום. אבל הוא נקרא מסיבה זו.

**(Anecdote: why “3.5 inch form-factor”?)**

- **3.5” = 8.89 cm**
- **But standard dimensions of a 3.5 inch disk are**
  - Length : 5.79” ≈ 14.7 cm
  - Height: 1.03” ≈ 2.61 cm
  - Width: 4.00” = 10.16 cm
- **Historical reason**
  - Capable of holding a platter that resides within a 3.5” (~90mm width) floppy disk drive

OS (234123) - files

48 2021-01-06 08:36:23

raskin@technion.ac.il

באמצע שנות ה 90 היה כמו שמתאר למעלה היה דיסקית קשיח שגודלו שלו היה 1.44 מיליביט וגודלו שלו היה 3.5 אינץ.

## Which blocks on disk constitute a file?

- **Recall that HDDs are “block devices”**
  - They are accessed in resolution of sectors
    - (Typically 512 bytes per sector, or more recently 4KB)
    - (Sequential access much faster than random access)
  - Sectors are identified by their LBA (logical block address)
    - 0,1,2, ..., N-1
    - Adjacent LBAs imply physical contiguity on disk  
=> Sequential access
- **However**
  - Which sectors together constitute a given file?
  - And how do we find the right sector if we want to access the file at a particular offset?

בקורס שלנו נסתכל על הדיסק כמו שאיך מערכות קבצים משתמשות על דיסק כלומר כמערך גדול של בלוקים כל בלוק בגודל 4 קילו בית או אם הדיסק שלנו זה 4 tera byte

או יש לנו מיליארדים בלוקים. כלומר מערך בגודל מיליארדים תאים, וליצוג זהה קוראים LBA בפועל רוצה לפנות לדיסק היא פונה עד ידי כך שבפועל היא רוצה לפנות לתא 78 של אותו מערך, ודיסק דרייבר אחראי לחתת אותו ומרתגלו אותו למשהו הקשור לחומרה. למשל אם החומרה היא hardDisk אז אמורה לדעת על איזה plator זה נמצא וויאיזה track רוצה לקרוא. אם מדובר ב SSD (שיש בפלאפונים במחשבים) אז החישוב היה שונה לגמרי.

---

## Need to understand...

- **Data structures**
  - On-disk structures that organize the data/metadata
    - Array?, linked list?, tree?,...
- **Access methods**
  - How to map system calls to the above structure
    - open(), read(), write(), ...
- **Unlike virtual memory, above decisions made purely in SW**
  - => Lots of flexibility
  - => Lots of filesystems,
  - Literally, from AFS (Andrew FS) to ZFS (Sun's Zettabyte FS)
- **We'll focus on the simplistic VSFS = Very Simple File System**
  - A toy example to help us understand some of the concepts

באופן כללי יש מלא מערכות קבצים. יש מערכות קבצים שהורדו שהם זמינות לכלום FAT, EXT234 ויש מערכות קבצים שהם לוקאליות ויש כאלה שהן מבודדות. כלומר אנחנו פונים איז בפועל אנו פונים לקובץ שיודב שבמחשב מרוחק באיזשהו server לא לוקאלי למשל AFS

NFS(Network file system ,

בקורס אנו נדונם ב מערכת קבצים VSFS שבעזרתה נוכל להבין איך מערכות קבצים עקרוניות עובדות. למשל FAT ארגון של קבצים יכול להיות שונה לגמרי.

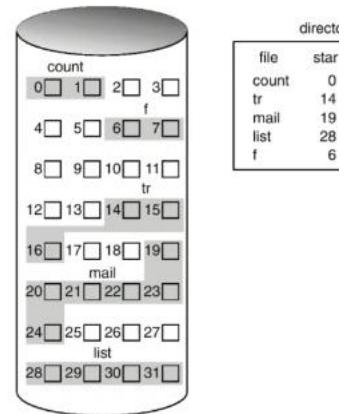
## But before, let's rule out contiguous block allocation

### • Pros

- Simple file representation
- Perfect sequential access when reading a file sequentially
- Minimal seek for random access
  - Just one seek for each contiguous access

### • Cons

- External fragmentation
  - Deletions might leave holes of unusable sizes
- Problem: how to append to a growing file?
  - Need to guess size in advance



לפני שנסתכל על מערכת קבצים VSFS, אנחנו נראה למה הגישה הטריוויאלית לא עובדת ולמה נאלץ להסתבר ולעשות דברים מסובכים. למה לא ניתן להגיד שככל קובץ יש לו שני מספרים 1. באיזה תא במערך מתחילה 2. כמה תאים תופס או מתי מסיים באיזה תא. למשל ניתן לראות לדוגמה יש למעלה 5 קבצים למשל קובץ mail מתחילה בבלוק 19 ומסתיים 6 בלוקים כלומר עד 24. השיטה הזאת היאści פשוטה, אין יותר פשוט ממנה, ואם היינו יכולים להשתמש בה אז היינו מעדיפים שכן השתמש בה. יחד עם זאת יש פה בעיה, לדוגמה ניתן לראות ש 50% מהבלוקים תפוסים בערך ואם ננסה להקצות קובץ בודל של 7 בלוקים אז אנחנו בבעיה כי אי 7 בלוקים ברצף. ולכן השיטה הזאת לא משתמשים בה.

## VSFS – layout on disk

- Assume we have a really small disk of N=64 blocks (0,1,... 63)

- Block size = sector size = 4 KB ( $\Rightarrow$  overall size of  $4 \times 64 = 256$  KB)  
BBBBBBBB BBBB  
01234567 8 15 16 23 24 31 32 39 40 47 48 55 56 63

- VSFS layout

- D = data blocks (56 out of the 64); all the rest is metadata:
- I = inode table (on-disk array, 5 blocks out of the 8)
  - Assume 128 B inode  $\Rightarrow$  At most:  $5 \times 4KB / 128 = 5 \times 32 = 160$  files
- d = bitmap of allocated data blocks (at most 56 bits are on)
- i = bitmap of allocated inode (at most 160 bits are on)
  - (An entire block for 'd' and 'i' is too big for VSFS, as we only need 56 and 160 bits, respectively; we do it for simplicity)
- S = "superblock" = filesystem information

OS [234123] - files

53

از איך מערכות קבצים מארגנות את המידע על הדיסק קויל יודעות לאתר את הקבצים עצם ואת התוכן שלהם. אנחנו נסתכל על מערכת קבצים קטנה (בגודל 64 בלוקים) וכל בלוק בגודל 4 קילו בית לנוכח המערכת היא בגודל של 256 קילו בית. אם נסתכל על 64 הבלוקים, לכל בלוק יש תפקיד מיוחד אז 56 בלוקים אחרים הם בלוקים של data או אם נרצה להקצת קובץ בגודל 10 קילו בית אנחנו צריכים 3 בלוקים. לנוכח 56 בלוקים אחרים צריכים למצוא 3 בלוקים פנויים ולצרף אותם לקובץ, והקובץ ייכשה צרייך לדעת שמידע שלו יושב בשלוש הבלוקים האלה. 5 הבלוקים שמוסמנים ב I זה בלוקים של Inode שמוגדרים במערך והם בגודל של kb20 ונקור ש inode זה 128 בתים לנוכח יש לנו 160 קלומר במערכת קבצים זו אנחנו יכולים לאחסן 160 קבצים, ספריות, Fifo. דבר הבא זה בלוקים שמוסמנים ב d (קטן) לוקחים בתוכו מערך של 56 בתים ומערך זה אומר אם בלוק מידע תפוס או לא אז תא 0 במערך זה מתחילה עם בלוק 8 עד שמסתהים עם בלוק 63 קלומר הוא בגודל 56 ואז תא אומר האם הבלוק תפוס על ידי קובץ או לא. למה צריכים את זה? כי שמנסים להקצת בלוק לקובץ אנחנו עוברים על המערך הזה ובוגרים מי פניו. עכשו הבלוק שמסמן ב I קטן זה אומר בלוק שבתוכו יש בית שבערתו ניתן לדעת איזה Inode פניו ואיזה אחד תפוס רק עכשו המערך בגודל 160 ביטים. ובлок אחרון זה S שהוא super block מכיל מידע על מערכת קבצים כמו meta data של קובץ או הוא יכול meta data של מערכת קבצים, קלומר מכיל איזה מערכת קבצים רצה למשל FAT, VSFS, FAT, אחרי זה יש מידע על גדים והקצאות,

למשל כמו בЛОקים מוקצה לinode, וכמה בЛОקים מוקצה לdata. וכל המידע שמאפשר לנו לעבוד עם instance קונקרטי של מערכת קבצים.

## Partitions, fdisk, mkfs, mount

- A disk can be subdivided into several “partitions” using ‘fdisk’
  - Partition = contiguous disjoint part of the disk that can host a filesystem
- Disks are typically named
  - /dev/sda, /dev/sdb, ...
- Disk partitions are typically named
  - /dev/sda1, /dev/sda2, ...
  - Listing partitions on disk /dev/sda:

```
<0>dan@pomelal:~$ sudo fdisk -l /dev/sda

Disk /dev/sda: 1979.1 GB, 1979120025600 bytes
255 heads, 63 sectors/track, 240614 cylinders, total 3865468800 sectors
Units = sectors of 1 * 512 = 512 bytes

Device Boot Start End Blocks Id System
/dev/sdal * 2048 499711 248832 83 Linux
/dev/sda2 501758 3865466879 1932482561 5 Extended
/dev/sda5 501760 3865466879 1932482560 8e Linux LVM
```

OS (234123) - files

הערה מאד מהרה היא מבחינה איך מרכיבים כמה מערכות קבצים ביחד. עושים זאת דרך פקודה mount. למשל שימושים דיסק או נוצר דיסק חדש. בUNIX בולו כללי יש פעולה mount שמחברת מערכות קבצים, ו umount שמנתקת מערכות קבצים. כל הדברים הללו נראה בתרגול.

## Partitions, fdisk, mkfs, mount

- **Running mount in shell with no args prints all mounts, e.g.**

```
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
tmpfs on /dev/shm type tmpfs (rw)
/dev/sdal on / type ext3 (rw)
/dev/sda2 on /usr type ext3 (rw)
/dev/sda3 on /var type ext3 (rw)
/dev/sda6 on /tmp type ext3 (rw)
```

- **For more details see**

- “Learn Linux, 101: Hard disk layout”
  - <https://www.ibm.com/developerworks/library/l-lpic1-v3-102-1>
- “Learn Linux, 101: Create partitions and filesystems”
  - <http://www.ibm.com/developerworks/library/l-lpic1-v3-104-1>

OS (234123) - files

57 2021-01-06 08:53:34

ניתן לראות למטה שעבור כל התקן שהוא דיסק יש מערכת קבצים שונה. וניתן לראות ש4 אחרונות זה מערכות קבצים שיושבות על דיסק פיזי 1 כי ככלם יושבים ב sda וכולם מאותו סוג ויש פה מערכות קבצים למטה למשל proc שהוא מערכת קבצים שמאפשרת לעשות monitoring למערכת הפעלה.

/proc file system is a mechanism provided, so that kernel can send information to processes. This is an interface provided to the user, to interact with the kernel and get the required information about processes running on the system. ... Most of it is read-only, but some files allow kernel variables to be changed.

## VSFS – finding an inode

- **VSFS layout**

```
|----- The Data Region -----|
SidIIIII DDDDDDDD DDDDDDDD DDDDDDDD DDDDDDDD DDDDDDDD DDDDDDDD
01234567 8 15 16 23 24 31 32 39 40 47 48 55 56 63
```

- **Given inumber (=index of inode in table), find inode block:**

- sector = (inodeStartAddr + (inumber x sizeof(inode\_t)) / blockSize
  - |---- in bytes ---|
  - |--- in bytes --|

- **Once found, the inode tells us all the file's metadata**

- Number of allocated blocks, protection, times, ... , and
- Pointer(s) to where data is stored

- **Pointers could be, for example,**

1. Direct
  - Point directly to all data blocks => |file| ≤ pointerNum x blockSize
2. Indirect (multi-level)
3. Linked list

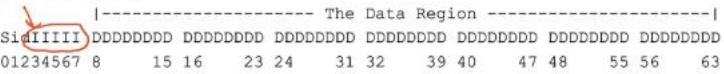
05 (234123) - files

58 2021-01-06

נזהר למערכת קבצים VSFS, אמרנו לפני שבוע שמחפשים קובץ אנחנו עוברים על ספרייה ומחפשים בתוכה את hw.txt ואז מוצאים אותה, ואז מה היא נותנת? הפלט של החיפוש הזה זה מספר ב Inode קלומר מה שמערכת הפעלה עשויה זה היפוש מהשם של הקובץ ל inode. נניח שמערכת הפעלה עשתה היפוש ומזכה שהוא נשמר ב inode מספר 7. אמרנו ש inode שמורים במערך של inode בגודל 160. איך מגיעים אליו? מה שמערכת קבצים עשויה הוא החישוב הבא, היא צריכה לדעת כתוב כלומר באיזה sector נמצא inode הרצוי. ואז היא תקרא את זה ל זיכרון וברגע שהוא נמצא בזיכרון היא ניגשת ל זיכרון וקוראת את המידע, אבל איך היא יודעת באיזה sector זה נמצא. נניח שמערך של Inode מתחילה איפה שמסומן

## VSFS – finding an inode

- **VSFS layout**


  
 The Data Region  
 3 x 4 KB

+

- **Given inumber (=index of inode in table), find inode block:**

- sector = (inodeStartAddr + (inumber x sizeof(inode\_t)) / blockSize
   
 |---- in bytes ---|                           |--- in bytes --|

- **Once found, the inode tells us all the file's metadata**

- Number of allocated blocks, protection, times, ..., and
- Pointer(s) to where data is stored

- **Pointers could be, for example,**

1. Direct
  - Point directly to all data blocks =>  $|file| \leq \text{pointerNum} \times \text{blockSize}$
2. Indirect (multi-level)
3. Linked list

OS (234123) - files

58 2021-01-06 08:58:21

נשים לב שיש שלוש בלוקים של Sid לפני שהמערך מתחילה כלומר הוא מתחילה בכתובת  $3^*4\text{kb}$

וכיוון ש inode הוא בתא 7 לכן לתוכה מוסיפים מסטר ה  $128^*$  inode בכתובת של inode, וכך לדעת באיזה sector זה נמצא מחלקים ב  $\text{KB4}$  לכן סך הכל

$$(3^*4\text{kb} + 128^*7)/4\text{kb} = 1$$

עכשו ניגשים לדיסק וטוענים את sector מסטר 1 לזכרון ועכשו מערכת קבצים יכולה לגשת לזכרון ולשלוף ממש את inode ומשם לדשת לקובץ. בתוך inode יש כל מיני דברים שקשורים לmetadata של הקובץ. למשל הרשאות, times, ומה שמשמעותו היום זה המידע שעוזר לנו לאתר ה data של הקובץ. אז יש 3 שיטות לאיתור,

4. Direct
5. Indirect
6. LinkedList

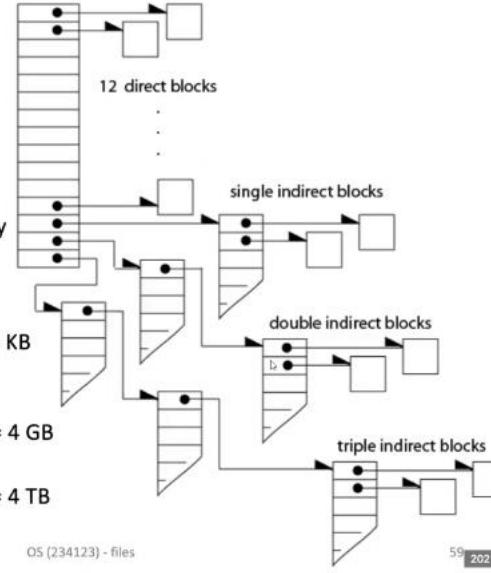
או מתחילה Direct השיטה זו כמעט ולא בשימוש. בעצם Inode יכול מערך של מצביעים, שהם מצביעים ל data blocks של אותו קובץ. נניח שנרצה לתרום בקבצים עד  $4\text{MB4}$  שהם די קטנים והם דורשים 1000 מצביעים שהוא המון! וכל אחד בגודל 4 בתים לכן צריכים  $\text{kb4}$  של מצביעים בלבד!! שמאז אחד זה בוגר די גדול כי רובם

בגדלים נמוכים מזה, אבל מערכת קבצים זאת גם לא תומכת בקבצים גדולים וגם עבור קטעים קטנים צריך inodes עצומים. לכן השיטה הזאת לא פותרת שום בעיה.

לכן משתמשים בשאר, נתחילה indirect או multi level indirect.

### Multi-level index (in the classic Unix FS)

- Assume each block pointer is 4 bytes long, and each block is 4KB
- 12 pointers in inode point directly to data blocks
  - Consumes  $4 \times 12 = 48$  B
- Single-indirect pointer points to a block completely comprised of pointers to data blocks
  - 1024 data blocks
  - $(12 + 1024) \times 4\text{KB} = 4144\text{ KB}$   
 $\approx 4\text{MB}$
- Double-indirect adds
  - $1024^2 \Rightarrow 4\text{KB} \times 2^{20} = 4\text{ GB}$
- Triple-indirect adds
  - $1024^3 \Rightarrow 4\text{KB} \times 2^{30} = 4\text{ TB}$



ננסה להבין מה היתרון. היא מאפשרת בצורה מאוד יעילה לאתר בלוק של קובץ. למשל אם נרצה בלוק 70 אז ניתן לערוך בתא 70 ושם יש את המצביע לבלוק data שרצויים. ככלומר האיתור block של קובץ זה די יעיל. מה שעשו בשיטה הזאת זה להקצתה מערך קטן בגודל 15 תאים ותאים ראשונים 0-11 הם direct וממש מצביעים לבלוקים של הקובץ. ככלומר כל עוד קובץ היא קטן עד kb48 או ממש עובדים עם הצבעות ישירות. אז עבורם ייצוג כזה הוא די יעיל, העשוו עבורם נרצה יותר, אז יש מצביע מספר 12 שהוא מצביע לblock בגודל kb4 אבל הבלוק הזה הוא לא של data אלא מצביעים. ככלומר כולם מחולק לkb4 וכל תא בו מצביע לblock של data, אז עד עכשווי 12 בלוקים ראשונים מוצבעים בצורה ישירה ובלוק מספר 12 נדרש לגשת אליו דרך מצביע מספר 13 במערך ואז בבלוק שמספר אלי אידקס 0 מצביע לבלוק 12 עד בלוק 1035 ככלומר כל עוד צריכים  $4\text{Kb} \times 1035 = 4\text{MB}$

אנו מוסדרים עד עכשו. אם נרצה מעבר לזה אז משתמשים ב double indirection. אז מה שעושים זה מצביע 13 במערך הוא מצביע ל block של מצביעים והוא לא מצביע לבlokים של data כמו single direction שראינו קודם, אז כל תא פה מצביע לבlok של מצביעים ואז הרמה אחרונה מצביעת ל block של data. אז אם נרצה בלוק מס' 1036 עושים כמה שביצור,

### Multi-level index (in the classic Unix FS)

- Assume each block pointer  $P$  is 4 bytes long, and each block is 4KB
  - 12 pointers in inode point directly to data blocks
    - Consumes  $4 \times 12 = 48$  B
  - Single-indirect pointer points to a block completely comprised of pointers to data blocks
    - 1024 data blocks
    - $(12 + 1024) \times 4\text{KB} = 4144\text{ KB} = \sim 4\text{MB}$
  - Double-indirect adds
    - $1024^2 \Rightarrow 4\text{KB} \times 2^{20} = 4\text{ GB}$
  - Triple-indirect adds
    - $1024^3 \Rightarrow 4\text{KB} \times 2^{30} = 4\text{ TB}$
- 

OS (234123) - files

59

לכן ניתן ליזג בסך הכל כנל בתא 13 מצביעים לבלוק של kb1 מצביעים כי הבלוק בגודל kb4 וכל כתובת זה 4 בתים. ומכל תא בבלוק זה מצביע לבלוק של kb1 מצביעים וכן (kb<sup>21</sup>) מצביעים ועוד kb1 מצביעים של single indirect ו עוד 12 מצביעים אז עד לנו בעצם

$$4\text{Kb}(1\text{kb})^{2+1\text{kb}+12}=4\text{GB}$$

לכן כן עוד קבצים שלנו עד GB4 אנו מוסדרים אחרית ניגשים לרמה הרביעית.

$$4\text{Kb}(1\text{kb})^{3+(1\text{kb})^{2+1\text{kb}+12}}=4\text{TB}$$

עכשו כל עוד קובץ שלנו לא מעל TB4 אנו מוסדרים.

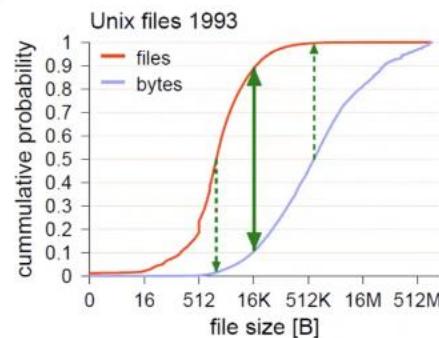
הערה. איך מערכת הפעלה מוצאת את ה inode שנמצא בדיסק. הרצתה קודמת אמרנו שקוראים תוכן של הספרייה, אמרנו שיש מיפוי של שם הקובץ inode. כולם רק על המעבר של הספרייה יש לנו inode שיש לו אותה מערכת הקבצים עושה החישוב

שראינו ו יודעת באזזה סקטור נמצא ואז פונה לדיסק דרייבר ו מבקשת ממנו להביא ל זיכרון את הסקטור והוא מעלה אותו מדיסק ל זיכרון ואז מערכת קצים יכולה לגשת ל זיכרון ו לקרוא ה inode ממנה ואז בשיטת indirect הולכת לבlok. נשים לב ש תוכן הספרייה שבו מוצאים המצביע של inode נמצא על הדיסק, וシステム הפעלה עולה היא צריכה לדעת איפה נמצאת root. כל היררכיה הזאת של מערכת קבצים מתחילה ממנה ומערכת הפעלה אמרה לדעת איפה נמצאת, ומשם היא יכולה להגיע לכל אחד מהספריות אחרות. אם ה inode די גדול למשל קבצים בגודל 4 גיגה אז ה inode הופך ל 4 מגה בית שהוא מטורף מבחינת זבוז זיכרון, ואפנרצה לעבור על קובץ אז צריך לgesht הרבה פעמים לדיסק כדי לטעון inode, אז ישפה יותר מדי גישות לדיסק בשביל לקרוא inode בלבד.

אעשה קצת סדר לעצמי. הכל מתחילה מספרייה ראשית ו אמרנו שספרייה ניתנת לחישוב עליה כרשימה מקווערת של dirent structs ובעצם, היא מכילה inode וגם את השם של הקובץ \_name של inode מכיל טבלת ה direct שבעצם מצביע על בלוקים בזיכרון, בלוקים אלו הם מכילים קבצים גם ושוב כל אחד יש לו dirent שבחכו יש את טבלת ה direct, ובעצם כך מורכבת מערכת קבצים.

## Why is hierarchy imbalanced like so?

- Data from
  - 12 million files from over 1000 Unix filesystems
- Y axis
  - CDF = cumulative distribution function = how many items [in %] are associated with the corresponding X value, or are smaller
- ‘Files’ curve
  - Percent of existing files whose size is  $\leq x$
- ‘Bytes’ curve
  - Percent of stored bytes that belong to a file whose size is  $\leq x$



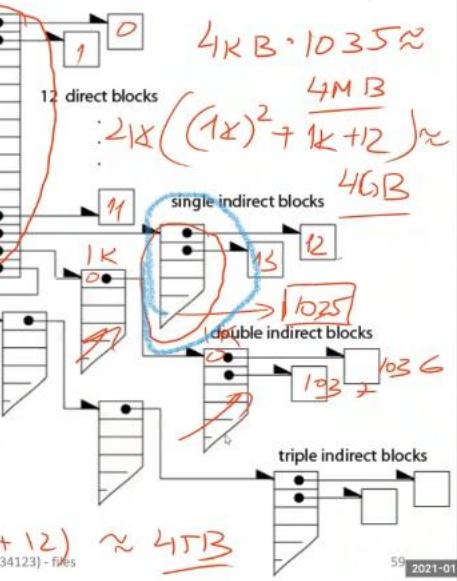
ברור לנו ש עבור ייצוג כזה של מערכת קבצים עבור קבצים קטנים עד 48 קילו בבית, מבחינת מיציאת הבלוק של data הביצועים פה הם הכי טובים שיכולים להיות. אנחנו טוענים inode לזכרון ועכשו הגישה ידי מהירה, למה 48 קילו בית מה מיוחד בו? כי ניתן לראות שהתפלגות של בתים וקבצים (cdf) שאנו רוצים לבדוק אכן שני גרפים מתנהגים אחד לשני, למשל נניח שלוקחים הרבה מערכות קבצים ועם הרבה קבצים כל אחת. מה אחוז הקבצים שגודל שלהם לכל יותר 16 בית, נניח שמקבלים שזה 10% וזה מה שאומר הגרף האדום בעצם אומר מה אחוז הקבצים שהם באותו גודל, מה הגרף הכהול אומר בדיקן? הוא אומר עבור נקודה בוא נkeh למשל כל הקבצים שגודל שלהם א בתים ונספר כמה בתים יש בקבצים האלו ונחשב אחוז הבתים מכלל הבתים שיש במערכות קבצים. אז ניתן לראות ש עבור 16 קילו בית זה מהו 90% מקבצים ורק 10% מבתים. וזה לא אמר להפתיע אותנו כי רוב הבתים יהיו מרוכזים בקבצים ענקיים אז הניסוי הזה עשו לפני הרבה שנים וגם היום נקבל אותה תוצאה כמעט. לכן נסהה לחת ביצועים טובים לקבצים עד 48 קילו בית ובדרך כלל רוב הרבעים הגיעו אליהם יכולה להיות אקראית, ככלומר לא ניגשים לבית 10, 11, 12 אלא קל הזמן יש קפיצות ובמקריםгалו גישה מהירה לבלוק היא קרייטית. מה עם קבצים גדולים למשל zip, סרטים למיניהם, איך ניגשים אליהם? כמעט אף פעם לא ניגשים אליהם בצורה אקראית אלא סדרתית. נניח שניגשים לבלוק 1366 וטענים לזכרון את הבלוק 1367, 1368 ואז ניגשים לבלוק 1336 ואז הבלוק 1367 יודעים איפה נמצא ואז בעצם ניגשים לכל הבלוקים

## Multi-level index (in the classic Unix FS)

- Assume each block pointer is 4 bytes long, and each block is 4KB
- 12 pointers in inode point directly to data blocks
  - Consumes  $4 \times 12 = 48$  B
- Single-indirect pointer points to a block completely comprised of pointers to data blocks
  - 1024 data blocks
  - $(12+1024) \times 4KB = 4144$  KB  
= ~4MB
- Double-indirect adds
  - $1024^2 \Rightarrow 4KB \times 2^{20} = 4$  GB
- Triple-indirect adds
  - $1024^3 \Rightarrow 4KB \times 2^{30} = 4$  TB

$$4KB ((1x)^3 + (K)^2 + 1K + 12) \approx 4TB$$

OS [234123] - files



59 2021-01

אחר לכך מקרים קודמות הבלוק אחורי נמצאו בזיכרון מה שמוסמן בירוק

## Multi-level index (in the classic Unix FS)

- Assume each block pointer is 4 bytes long, and each block is 4KB
- 12 pointers in inode point directly to data blocks
  - Consumes  $4 \times 12 = 48$  B
- Single-indirect pointer points to a block completely comprised of pointers to data blocks
  - 1024 data blocks
  - $(12+1024) \times 4KB = 4144$  KB  
= ~4MB
- Double-indirect adds
  - $1024^2 \Rightarrow 4KB \times 2^{20} = 4$  GB
- Triple-indirect adds
  - $1024^3 \Rightarrow 4KB \times 2^{30} = 4$  TB

$$4KB ((1x)^3 + (K)^2 + 1K + 12) \approx 4TB$$

OS [234123] - files

59 2021-01

נגישים מבלוק בירוק שהוא כבר בזיכרון מקרים קודמות ממנו נגישים לדיסק להביא בלוק מסוים אחד ואז עוד פעם במצבה הכי עילית נגישים לבלוקים מתוך ראשוני הבלוק שמוסמן בשחור וזה שאנו חמייד נגישים במצבה סדרתית שהאפשרות prefetch

מאוד יעיל לנו בסרטים לא מרגינאים שופ בעיות בגישה לדיסק. מה יוצא דופן מכל הסיפור הזה? אם עושים database שבו יש מלא גישות אקראיות, אם זוררים במבנה נתונים זה היה אחד החסרונות של עצי AVL או במקום השתמשנו בעצי B+ עם מקדם די

גודול, עושיפ את זה כי אפשר לעשות prefetching לכל העז פרט מעלים, ואז מהפשים בזיכרון עד שmaguiim לעלה וברגע שmaguiim לעלה קוראים רק את הולה מדיסק, אז זאת הסיבה כי אנחנו עבר קבצים גדולים יודעים לעשות קריאה יעה בצורה סדרתית ולא קריאה יעה בגישה אקראית.

## Extents

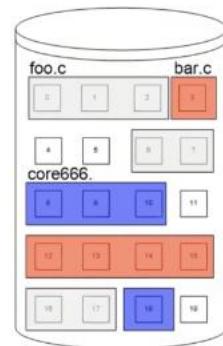
- **Motivation**

- Hierarchical block structure could mess up sequential access
- Dynamic append/truncate/delete I/O ops might hinder sequential access (= every leaf block might reside someplace else)

| Catalog |       |        |        |
|---------|-------|--------|--------|
| foo.c   | (0,3) | (6,2)  | (16,2) |
| bar.c   | (3,1) | (12,4) |        |
| core666 | (8,3) | (18,1) |        |

- **Solution: extent-based allocation**

- Extent = contiguous area comprised of variable number of blocks
- inode saves a list of (pointer, size) pairs



## Extents

- **In ext4**

- Current default FS for Linux
- With 4KB block, size of a single extent can be between: 4KB ... 128MB (32K blocks)
- Up to 4 extents stored in inode
- Hierarchical if more than 4
  - Indexed by an “Htree” (similar to Btree but with constant depth of 1 or 2)
  - Allows for efficient seek/offset search

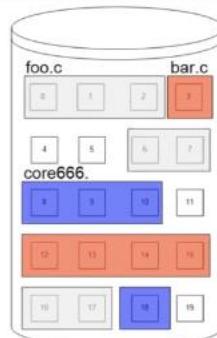
| Catalog |       |        |        |
|---------|-------|--------|--------|
| foo.c   | (0,3) | (6,2)  | (16,2) |
| bar.c   | (3,1) | (12,4) |        |
| core666 | (8,3) | (18,1) |        |

- **Supported by most current filesystems**

- ntfs, ext4, hfs+/apfs, btrfs, reiser4, xfs, ...

- **Comments**

- ext4 = extended filesystem 4 (name unrelated to “extent”)
- ext4 is backward compatible: ext2 and ext3 can be mounted as ext4

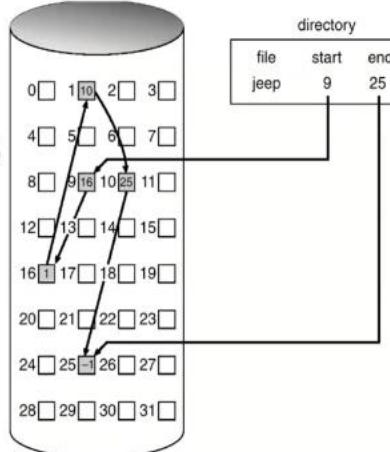


בקשר לקבצים גדולים, שמדובר הקבצים גדולים הגישה היא סדרתית לנן לאחר כל בלוק לא צריך לעשות זאת בצורה יעה כי אנחנו כבר בזיכרון או יודעים אם כל בלוק נמצא.

מה שחשוב לעשות למשל שעושים `cpizun` או שעושים צריך לעשות בצדה דיבר עליה לקרוא **data** מהdisk. ואחד החסרונות של מערכת קבצים זה בעצם שכל בלוק נמצא איפשהו בdisk, זאת אומרת שמתכוילים על disk, הגישה לדיסק היא כל פעם במקום שונה (גישה אקראית לדיסק) והוא דיבר איטית, הדיסק מספק מידע בסדר גודל של MB לשניה. אז מה יכול לשפר ביצועים? גישה אקראית למשל אם יכולים לפניות לdisk ולהגיד לו לקרוא מבלוק 17 למשל 1000 בלוקים או הקצב הקיראה/כתיבה יגיע MB200 בשניה. אז אם הינו יכולים להגיד שעובד קבצים גדולים הבלוקים מסודרים ברצף ויכולם לקרוא אותם ברצף אז יכולים לשפר ביצועים, זה עושים דרך Extent זה עוזר להוסיף לקובץ לא רק בלוקים בודדים אלא רצף של בלוקים. דוגמא לכך זה מערכת קבצים EXT4 שזה לכל קובץ אפשר להוסיף 4 רצפים וכל Extent הוא מblk (בלוק אחד) עד 32 אלף בלוק כלומר Mb128, קלומר תיאורית כל קובץ של חצי גיגה בניוי מחצי רצפים שכל אחד זה MB128 כמובן לא תמיד ניתן להגיע לזה כי לא תמיד יש רצף של בלוקים פנויים בdisk. הגישה של Extent היא דיבר מודרנית ומשתמשים בה הרבה היום.

## Linked list block allocation

- **A file could be a linked list**
  - Such that every block points to the next
  - Each directory entry points to 1<sup>st</sup> and last file blocks of file
    - (Need last for quick append)
- **Severe performance overhead**
  - Because seek operations require linear number of ops
  - Each op is a random disk read
- **Optimization**
  - Move pointers out of blocks
  - Put them in external table that can be cached in memory
  - Rings a bell?



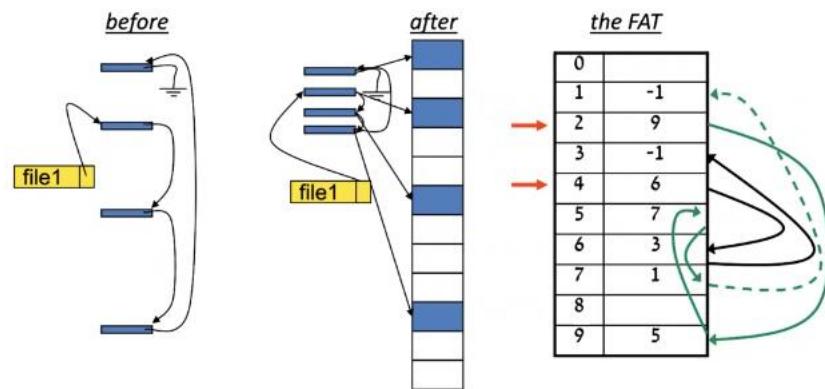
נדבר על רשימה מקוורת, מה זה רשימה מקוורת באופן כללי? נניח שיש לנו jeep שמתהיל בבלוק 9 ובתוך בלוק 9 שמורים 4 בתים ושמם אינדקס של הבלוק הבא שהוא 16 אחר כד 1 והלאה, בעצם זאת השיטה של רשימה מקוורת. יש פה יתרון שאין

מגבלה לגודל של קובץ. וכמוות מצביעים שמקצים זה תמיד לפני גודל של קובץ, שזה 4 בתים מתוך 4 קילו בית (גודל הבלוק) וגם יעיל לגישה סדרתית יחסית. אבל נניח שנראה לראות סרט ולגשת קצת קדימה, ונניח שהוא בגודל MB400 ואנחנו רוצים לкопז לאמצע שזה MB200 אז צריך לדלג על 50 אלף בלוקים, כי בשביל לדעת בלוק 50 אלף כי צריך לדעת מה הקודם.

## FAT filesystem

- **FAT = file allocation table**
  - The DOS filesystem, still lives
  - Named after its main data structure

| name |     | start block |
|------|-----|-------------|
| foo  | ... | 2           |
| bar  | ... | 4           |



השיטה של רישימה מקוושרת היא בשימוש של מערכת קבצים FAT. אז בעצם היא שואלה למה אנחנו עושיםسلط בין מצביעים לבין **data**. בדיק אודה בעיה של עץ AVL מול עצי B+. למה צמתים מכילים גם **data** וגם מפתחות ולא להפריד? וזה בדיק מה שעושים ב-FAT, לקחו את ברשימה המקוושרת ומוציאים אותה החוצה ויש טבלה שנראית FAT שהיא אחת לכל מערכת קבצים. ובעצם מגדרה את הרשימות הקשורות. למשל למעלה בתמונה ניתן לראות את הטבלה The Fat ונניח שיש קובץ שמחילה בבלוק 2 נקרא **foo**, הולכים לטבלה לתא 2 ורואים שיש שם 9 הולכים לתא 9 ואחריו כתוב 5 ואז הולכים ל5 ושם כתוב 7 ושם כתוב 1 ואז אחרי אחד -1 וזה סימנו (קיבלו מספר שלילי). נניח שיש לנו מערכת קבצים 4 גיגה ביבט זה אומר שיש 1 מגה בית כניסה כלומר גודל הטבלה 4 מגה בית. מה שעושים זה טוענים הטבלה לזכרון, ואז חיפוש למשל תא 10 אלפיים של קובץ אז לא נצטרך לגשת לדיסק ולקחת בלוק ובתוכו למציא הבלוק אחריו, לנן אומנם עושים 1000 קפיצות אבל כולם בזכרון, אז נניח שעושים 10 אלפיים קפיצות בזכרון וכל קריאה היא סדר גודל של 200ms לנן להגיע לבlok 10 אלפיים צריך 2 ms זה כולם ולא נרגיש את זה בעצם. הבעיה הגדולה של FAT זה עמידות שלו בפני נפילות, נניח שקורא משהו לטבלה, אז כל מערכת הקבצים

הルכה לפח, וכן מה שעושים זה שומרים אותה בכמה עותקים וגם אז היא נחשית למערכות קבצים הפחות יציבות ולשזר מידע מידע FAT בהרבה מקרים בלתי אפשריים. בעיה אחרת זה שמערכת קבצים בגודל 4 GB זה לא מעניין, אז אם נסתכל על מערכת קבצים של TB1 וננסה לחלק את זה ב-4 KB גודל של כל בלוק, ו-4 בתים לכל כניסה אז יש לנו GB4 גודל של הטעלה, אז מה הבעיה? אנחנו לא יכולים לטען אותה לזכורן כולה, אז מה שקרה במערכת קבצים גדולים לא משתמשים ב-FAT, ואז קפיצות נוצרת לעשות אותן דרך גישות לדייסק ולא זיכרון. בכל זאת מערכת קבצים FaT היא סוג של בסדר, ולאט לאט היא הולכת ונעלמת.

## Motivation

- **Problem #1**
  - Sometimes disks fail
- **Problem #2**
  - Disk transfer rates can be much slower than CPU performance
- **Solution – part #1: striping**
  - We can use multiple disks to improve performance
  - By *striping* files across multiple disks (placing parts of each file on a different disk), we can use parallel I/O to improve the throughput
- **But striping makes problem #1 even worse**
  - Reduces reliability: 100 disks have 1/100th the MTBF (=mean time between failures)
- **Solution – part #2: redundancy**
  - Add redundant data to the disks (in addition to striping)

סיימנו הדיוון על ייצוג של קבצים, ובעצמן נדבר קצת על יציבות. נניח שרוצים לבנות מחשב או איזשהו server ורוצים לשם עליו כמה דיסקים, ונניח שיש לנו שתי אופציות.  
אופציה 1 - לשם דיסק אחד של TB8  
אופציה 2 - לשם 8 דיסקים של TB1  
מה ההסרונות ויתרונות של כל אופציה, למשל באופציה 2 ניתן לגשת במקביל לכמה דיסקים ולשפר ביצועים זהה שיקול ראשון והוא של ביצועים. יש עוד שיקול זהה אם דיסק אחד מתקלקל הולך לנו רק TB1 של ביצועים במקום TB8.

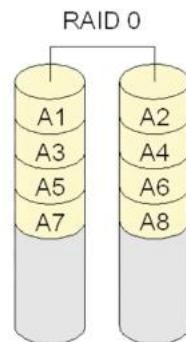
## RAID (Redundant Array of Inexpensive Disks)

- **Individual disks are small and relatively cheap**
  - So it's easy to put lots of disks (10s, 100s) in one box/rack for
    - Increased storage, performance, and reliability
- **Data plus some redundant information are striped across the disks in some way**
  - How striping is done is key to performance & reliability
  - We decide on how to do it depending on the level of redundancy and performance required
  - Called "RAID levels"; standard RAID levels:
    - RAID-0, RAID-1, RAID-4, RAID-5, RAID-6
- **Proposed circa 1987**
  - By David Patterson, Garth Gibson, and Randy Katz (@ Berkeley)

נשכח מדיסקים, נניח שאורך חיים של מנורה להישרפ זה אלף ימים. נניח שמקימים פקולטה שיש בה 1000 מנורות תוך כמה זמן אמורים לצפות לשריפה למנורה ראשונה? יום אחד, תוך יום אחד אנוינו ב מוצר מוגבר שמנורה שנשרפה. למה זה קשור לדיסקים? (דיסקים גם כל זה דבר שמתקלקל, יש לו אורך חיים די קצר) עכשו אם נשימוש בהרבה דיסקים החיים שלהם מתקצר במעט, אז איך משתמשים בדיםקים רגילים לפחות כל מיני בעיות.

## RAID-0

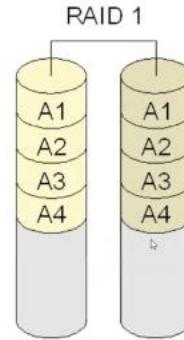
- **Non-redundant disk array**
  - Files striped evenly across  $N \geq 2$  disks
  - Done in **block resolution**: in principle, technically, a block can be as small as 1 byte; but is a multiple of drive's sector size in most standard RAID levels taught today (0,1,4,5,6)
- **Pros**
  - High read/write throughput
    - Potentially  $N$  times faster
    - Best write throughput (relative to other RAID levels)
  - Faster aggregated seek time, that is
    - The seek time of one disk remains the same
    - But if we have  $N$  independent random seeks, potentially, we can do it  $N$  times faster
- **Con**
  - Any disk failure results in data loss



נסתכל בכל מיני שיטות איך לעשות שימוש במבנה דיסקים. יש לנו בעצם שתי שיטות קיצונית וכל מיני דברים ביניהם. כל הגישות שך איך להשתמש בהרבה דיסקים נקראות RAID שהוא קיצור ל redundant array of expensive disk. שיטה ראשונה היא לעשות פיזור בלוקים בין הדיסקים, למשל בלוק 1 נכתב לדיסק אחד ובלוק 2 לדיסק שני, זה נותן לנו ביצועים יותר טובים, כי נניח שנרצה לכתוב לבלוק 1 יblk 2 או גיבש לדיסקים נפרדים ועושים כתיבה קריאה באופן מקביל, لكن שיפרנו ביצועים פי 2 בממוצע, עוד משהו שאם קונים 2 דיסקים של TB2 כל אחד, אז נצלילה לשמור TB4 של data כלומר אין לנו overhead. החיסרון הוא שאם אחד מהם נופל, אז אין לנו יותר שום מידע, אז דיסק שני אין לו שום משמעות בלי הדיסק שנפל.

## RAID-1

- **Mirrored disks**
  - Files are striped across half the disks
  - Data written to 2 places: data disk & mirror disk
- **Pros**
  - On failure, can immediately use surviving disk
  - Read performance: similar to RAID 0 (Why?  
How about write performance?)
- **Cons**
  - Wastes half the capacity
- **Related: nowadays, in the cloud / data center**
  - Replicating storage systems are prevalent
  - Often 3 replicas of each “hot data chunk” (chunk sizes are usually in MBs, to amortize seeks; hot = used a lot)
  - Replicas get “randomly” assigned to disks to balance the load
  - That is, not using classical RAID 1 disk pairs; rather, every disk is striped across many (potentially all) other disk



גישה אחרת RAID-1 היא משתמש בשני דיסקים וכל מה שכותבים לדיסק אחד כותבים לדיסק שני, גם מבחינת ביצועים שעושים קריאה אפשר לקרוא במקביל, לגבי כתיבה, במקרה הגורע נשאר אותו דבר אבל גם בפועל אנחנו קצת מושפרים כי לא שיש לנו אפשרות לעשות כתיבות בצורה בלתי תלולה מקראיות, אז אם יש הרבה קרייאות אז גם כתיבות מתעכבות, וכך בכלל שיש מערכת זאת לנוכח לאזן כמה קרייאות בין הדיסקים ככלומר כל דיסק עמוס פחות וכך ניתן לעשות כתיבות יותר מהר כי לא מעוכבות על ידי קרייאות. החיסרון היא מהיר, ככלומר שילמנו על TB4 ובפועל שומרים רק TB2 של data. בתעשייה מוכנים לשלם פי 2 חברות גדולות כמו גוגל או כו אף המידע הוא קריטי, ככלומר משלמות>If של כסף לשמור מידע שלנו, **ואָא** לא בצדקה הזו, נניח שהיברנו 15 דיסקים ביחד, ובכל אימיל שמקבלים משכפלים 15 פעמים בדיסק, אבל אם המחשב נשחרר פיזית או יכול להיות דברים פהות נוראים, למשל בעית החסל ומהשבר לא עובד, או אסון טבעי. וכך לשמר אותו מידע ב-700 פעמים באותו מקום זה חסר כל ערך שכן מה שהחברות עושות משכפלות את המידע ונמצא מקומות שונים בעולם, כך שאם יש בעיה בינוי יורך אז יכולים לשחזר המידע בברלין.

## Reminder: parity/xor

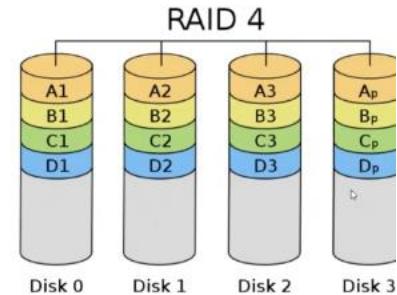
- **Modulo-2 computations**
  - 0 = even number of 1s
  - 1 = odd number of 1s
- **When one sequence bit is lost**
  - Can reconstruct it using the parity
- **In storage systems**
  - Parity computed on a block level

| # | bit sequence | parity/xor |
|---|--------------|------------|
| 1 | 000          | 0          |
| 2 | 001          | 1          |
| 3 | 010          | 1          |
| 4 | 011          | 0          |
| 5 | 100          | 1          |
| 6 | 101          | 0          |
| 7 | 110          | 0          |
| 8 | 111          | 1          |

נרצה לעשות איזון בין מחיר גבוהה, לבין גם ביצועים ו redundancies או יש כמה שיטות כדי לעשות איזון בצורה אחרת.

## RAID-4

- **Use parity disks**
  - Each block (= multiple of sector) on the parity disk is a parity function (=xor) of the corresponding blocks on all the N-1 other disks
- **Read ops**
  - Access data disks only
- **Write ops**
  - Access data disks & parity disk
  - Update parity
- **Failure => “degraded read”**
  - Read remaining disks plus parity disk to compute missing data
- **Pros**
  - In terms of capacity, less wasteful than RAID-1 (wastes only 1/N)
  - Read performance similar to RAID-0 (How about write performance? Next slide...)



מה RAID-4 עושים? נניח שיש לנו 4 דיסקים ונkeh 3 בלוקים ראשונים מדיסק 1,2,3  
ונעשה XOR קלומר

A1 xor A2 xor A3

, זה נקרא  $p_A$  שהוא משלים כמוות האחדים במספר זוגי ואז יש לנו דיסק ששומר את ה parity's האלו התוצאה זו נמצא בדיסק 4. אז אם הדיסק 2 התקלקל אנחנו יכולים לשחזר את כל המידע שם כל,

$$A2 = Ap \text{ xor } A1 \text{ xor } A3$$

או שיחזרנו את כל המידע שהוא שם, ומערכת זו עוזרת לו להתמודד עם נפילה של דיסק אחד, אז היתרונו היא שיכולים לסייע נפילה של דיסק אחד מתוך ה3. מבחינת ביצועים נראה שאנו מצליחים כי אם מפזרים בלוקים בין הדיסקים אז עומס על כל דיסק יהיה נמוך, אבל כל הבעיה היא בכתיבות, אז אם במקרה עושים כתיבה ל  $A1, A2, A3$  אז קל כי עושים כתיבה במקביל ומחשבים את ה parity במקביל. נניח שעושים שינוי של  $c_1$  בלבד,

או צריך לשנות ה parity החדש בצורה צזו

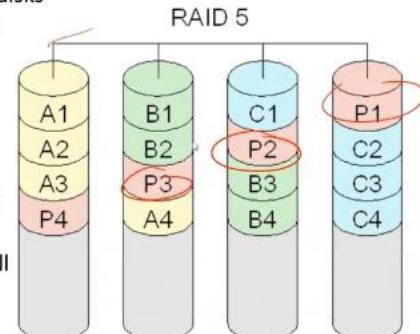
$$cp' = cp \text{ xor } c_1 \text{ xor } c_1'$$

כאשר  $cp$  זה parity ישן,  $c_1$  זה AFTER השינוי.

כלומר עבור כל כתיבה צריך לקרוא  $cp$  ישן ואת  $c_1$  ישן ואז לעשות חישוב בזיכרון (מעט חינם) ואז לכתוב  $cp$  ואת  $c_1$  כל כתיבה צריכה לעשות על הדיסק גם קרייה וגם כתיבה. זה אומר לפחות בפקטור של 2 פוגע במערכת.

## RAID-5

- Similar to RAID-4, but uses block interleaved distributed parity
  - Distribute parity info across all disks
  - A “stripe” is a concurrent series of blocks ( $A1B1C1, A2B2C2, \dots$ )
  - The parity of each stripe resides on a different disk
- Pros
  - Like RAID-4, but better because it eliminates the hot spot disk
  - E.g., when performing two small writes in RAID-4
    - They must be serialized
  - Not necessarily so in RAID-5
    - => better performance



בשביל לטפל בבעיות של RAID 4 אז ניתן לראות שהשיפור נעשה על ידי כל שימוש את parity כל פעם בדיסק אחד ולא רק בדיסק אחד, ככלומר מפוזרים את ה parity וכן גורמים לאיזון עבודה בין הדיסקים

תקשורת.

קצת מידע שהיפשתי.

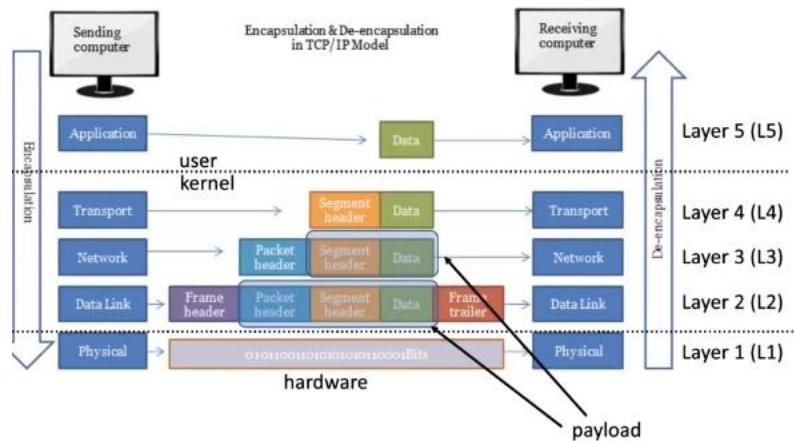
קודם כל בשבייל לדעת איך מחשבים מתקשרים ביניהם צריך לדעת מה הדרישה, כל מחשב מרכיב אליו networking driver שתפקידו זה רק לכתוב ולקראן מתחום server. עכשו מה זה server? אנו יכולים לעשות server המקומי על ידי כך שיתוף של מערכת קבצים בין כמה מחשבים בבית למשל ואז יש לנו server המקומי, בעולם האמתי יש לנו מיליון משתמשים וצריך שיהיו מחוברים ביניהם, איך זה נעשה? זה נעשה על ידי כך שיש server ענקי שיכול לתרום בכל מערכות הפעלה אפשריות וגם ניתן לכתוב אליו ולקראן ממנו, ואפשר לחבר אליו מיליון מיליארדים של אנשים. אז למשל שולחנים הודיעו דרך אינטרנט מה שקרה בפועל זה שקדם כל מבקשים ממערכת הפעלה לגשת לnetwork דרייבר, מערכת הפעלה נכנסת לשגרת גרעין, ניגשת לדרייבר דרך אבל ה `fdt` וכותבת אליו, הדרייבר הזה יודע לתקשר עם server, והserver מבקש מחשב אחר גישה אליו דרך פסיקה כדי לכתוב למשל ואז מה שכחכנו יהיה מוצג על המסך של המחשב של משתמש 2 וזה בעצם כל מה שקרה בצורה כללית.

## Protocol definition

- **Communicating parties are**
  - Host machines (computers) & processes
- **A network communication protocol is a set of rules defining**
  - Format & order of messages sent & received
  - Action taken upon message transmission & reception
- **All network communication activity**
  - Is governed by protocols

נתחילה מהגדיר מה זה פרוטוקול בכלל? זה סוג של הסכמאות איך עושים דברים, ככלומר אוסף של חוקים. ככלומר מגדרים מה הסדר של העברת מידע ומה הפורמט, ומה הדברים שאמורים לקרות מהתוצאה של קבלה של מידע זה. מידע באינטרנט עוברת ב pocket זו"א אוסף של בתים שבדרך כלל pocket נניח יש לה 1500 בתים בערך ובעצם שרוצים להעביר קובץ גדול בראש צריך לחלק אותו לchunks קטנים ולשלהם כל chuck. וכל chunk זה pocket וכל אחת עד פורמט מסויים, רוצחים לוודא שהיא מגיעה לייעד, וגם באיזה סדר נשלהו.

## Network protocol stack consists of layers



בעם אם נסתכל על התקשרות, נראה שפיטוי של תוכנה מתבצע על ידי שכבות. אז איך מפתחים מערכת הפעלה? פעם הייתה מערכת הפעלה מונוליתית, כלומר קוד די גדול שמתפל בכל הדברים, לפחות לאט התברר שגישה זו די בעייתית. למשל נניה שנרצה לעשות שינוי ב file system, או השינוי דורש לעבור על כל הקוד ולשנות כל מיני דברים ואולי בגל שינוי או scheduling לא עובד. אז מה שעשו זה לחלק מערכת הפעלה ל מודולים, ויש מודול שמתפל במערכת קבצים, אחד בזיכרון ואחד ב scheduling, וכל אחד נותן שירות למודול אחר. למשל אם נרצה לעשות שינוי ב file system ורוצים להשתמש במשהו שקשרו לזיכרון, או משתמש במודול שקשרו לזיכרון בלי צורך לעדכן שאר מודולים אלא רק מודול הרלוונטי. שמסתכלים על network, יש מודולים שונים שנקראים layers שכל אחד יש לו תפקיד מסויל. וננה שוננסה להבין זה הפקיד של כל אחד מהם, בסופה של דבר צריך להבין שאם למשל נסתכל על אפליקציה browser, הוא שולח מידע מבחרתו request, תביא לי אתר כלשהו, ואם שולחים המידע הזה דרך WIFI אז מה שמשדרים זה ביטים מועברים דרך תוווק קלומר אחדים ואפסים.

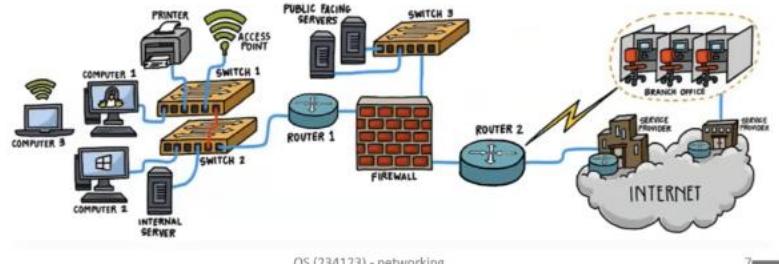
## Application-layer (L5) protocols

- When the protocol is determined by the app
  - There are many examples
  - Here are a few
- Standard (protocol determined by multiple organizations)
  - HTTP, HTTPS (web browsing)
  - SMTP, IMAP, POP (email)
  - VoIP (voice)
  - iCalendar (scheduling)
  - NFS (distributed filesystem)
  - SSH (secure shell)
  - Bitcoin (cryptocurrency)
- Proprietary (single organization; still, can be open)
  - Microsoft Exchange (mail & scheduling)
  - Skype, Zoom (mostly video conferencing)
  - WhatsApp, Telegram (mostly text messaging)

נתחיל משתי שכבות עליונות, נסתכל על המודל העליון, בו יש אפליקציות, זה לא אפליקציות של browser, explorer או דברים כאלה אלא HTTPS, HTTPS והלאה, כל אפליקציה קובעת באיזה פורמט רוצה להעביר את ה data. האם הוא מוצפן או לא? כמה בתיים? מה הסדר? והלאה. בעצם כל אפליקציה אם נסתכל על המידע ש IMAP מנסה להעביר ונשווה את זה למידע VoIP רוצה להעביר זה מידע שונה לגמרי, פורמט שונה לגמרי, מטרה שונה לגמרי. ויש כמויות מאוד גדולות של פרוטוקולים שרצים ברמה 5, ויש מהל סטנדרטים שימושיים בהם יום יומ לmdl HTTP, SSH אפשר לראות ולדעת מה הפורמט והוא מתועד היטב ולא חסוי, יש אפליקציות שהבעלים שלהם לא רוצים להסביר ולתעד מה הם מסווגות, כי הם רוצים להסתיר את המידע, למשל, zoom, skype אין לנו יכולת להסתכל על הביטים ולהבין מה קורה שם, המידע הוא מוצפן וגם לורמת לא ידוע.

## Lossiness

- When considering protocols, we should be aware that
  - Networks are inherently lossy
  - What's sent won't necessarily reach its destination
- For example, because
  - Bits specifying destination get flipped due to electrical problems
  - Network elements (routers, switches, APs, endpoints) malfunction
  - Network cables get severed



ששולחים מידע המידע הזה עובר גם בראשת בינוי על תשתיות מסוימת, למשל תשתיות קוית, WIFI והמידע עובר דרך הרבה התקנים. למשל טלפונים לwebserver של גוגל, זה לא שמחשב שלנו מחובר דרך קבל שיווצר ממחשב שלי ומחובר ישירות לתוכה webserver של גוגל. המידע שהשולחים עובר דרך הרבה התקני תקשורת בראש, דרך router של בזק למשל שנמצא בבית לעוד כל מיני router בתחום שיגיע לאותם שירותי של גוגל. המונן דברים יכולים לקרות למידע זה, למשל להשתבש או למכת ליאבוד, או יכול להגיד בסדר שונה, למשל שלנו שני שני pockets והם הגיעו לפימסדור הפוך. אז יש הרבה מאוד דברים שנצטרכן לטפל בהם על מנת לאפשר תקשורת נכונה. מה זה אומר תקשורת נכונה? זה כבר תלוי באפליקציה, יש كانوا אפליקציות שאכפת להם זה שלא, יש גם אפליקציות שרוצות שימושה יודעת שהגיעו לפיקסל והן מניחות שmagiut כפי שהלכו, ויש כאלה שלא סומכות על אף אחד.

---

## Transport-layer (L4) protocols

- **Protocols in this layer**
  - Provide host-to-host communication service for processes
  - Implemented by the OS (typically)
- **There are many, but only 2 are widely used**
  - TCP & UDP
    - Implemented by all OSes
    - Account for the vast majority of internet traffic

שכבה אפליקציות משתמשת בשכבה מס' 4 שזאת שכבת ה프וטוקולים שנונת שירות לשכבה מס' 5, ושכבה 3 נותנת שירות ל-4 ו-2 נותנת ל-3 ו-1. מה זה שכבה פרוטוקולים? היא מכילה כל מיני פרוטוקולי תקשורת ויש הרבה כאלה אבל כמעט רוב הtraffic מועבר על ידי שני פרוטוקולים נפוצים, TCP, UDP.

### TCP (transmission control protocol)

- **The protocol that cares...**
  - It cares about data loss & reordering  
(acronym should've been "Transmission that Cares Protocol" 😊)
- **Provides**
  - Stream of bytes abstraction, namely
  - It ensures that all bytes arrives, in order, to the receiving app
- **Said to be "connection-oriented"**
  - A communication 'session' between the two parties must be negotiated/established before data transmission can begin
- **How does TCP implement its nice properties?**
  - Here's an *oversimplified* explanation,  
to provide intuition,  
in a nutshell...

למשל HTTP משתמש ב TCP

נתחיל מ TCP שהוא מנסה לספק אבטחה מעלה סטריניג של מידע. למשל שולחים נתונים והם מגיעים לפי הסדר שלנונו. ככלומר הוא חייב לטפל ב pocket שהולכות לאיבוד, גם אם שניים החלפו בסדר או הוא מטפל בזוה. וגם עובד ברמת connection, כלומר שני מחשבים רוצים לדבר זה עם זה, הם פותחים connection וואז בעזרתו מעבירים מידע. כמה תהליכיים יכולים לפתח כמה connections ז"א לתוכן B,A, יכולם לפתח כמה connection ביניהם.

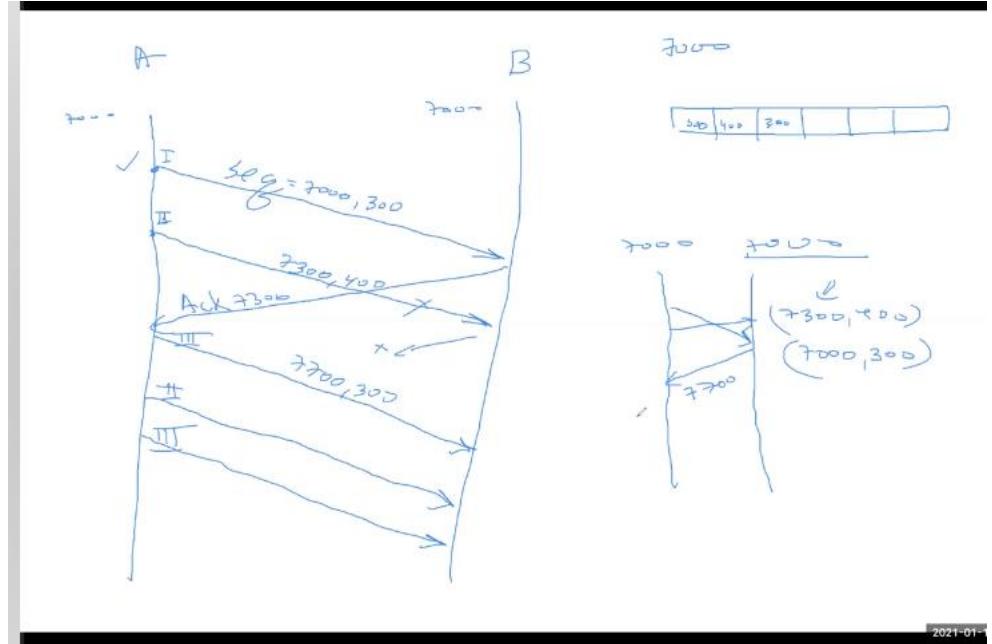
איך הוא מצליח למש את כל הדברים האלה?

## TCP (transmission control protocol)

- **Sender**

- Split bytes into contagious chunks (called “segments”)
  - Each chunk specifies which bytes [from ... to]
- Keep chunks around
  - Until receiver acknowledges receiving them
- Resend chunks
  - That haven’t been ack-ed for some time
  - That the receiver says it’s missing
- When noticing sent chunks don’t reach receiver
  - Slow down transmission rate
  - Carefully speed up otherwise
- Congestion control: changing rate response to drops/acks
  - Tries to address buffering problem

– [https://en.wikipedia.org/wiki/TCP\\_congestion\\_control](https://en.wikipedia.org/wiki/TCP_congestion_control)

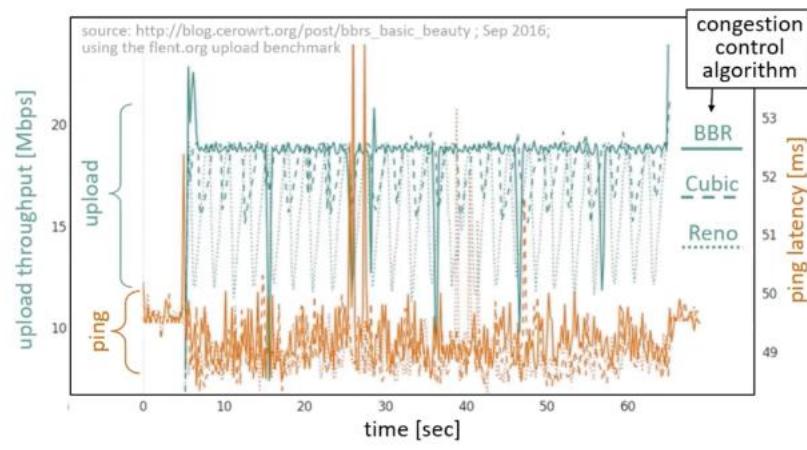


יש לעלה הסבר, במקומות אנהנו נציג להבין.

נניח שיש לנו שני מחשבים B,A וכל אחד רץ עליהם תהליך, נניח שיש לנו מידע ארוך, מחלקם אותו ל pockets של מידע. לכל אחד יש גודל, ויש איזה מספר סידורי שני תהליכי חיים לדעת, נניח שזה 7000 ונניח ששניהם מכירים אותו, שנוצר ה handshake TcP. ש מקים תקשורת שני תהליכי האלו מתחלפים בכל מיני pockets handshake. ומסכימים על המספר הזה. נניח שתתהליך A שולח pocket ועליה כותב 7000 והיא בגודל של 300, הוא לא חייב לה痴ות, הוא יכול להמשיך לשולח, הוא שולח את ה pocket הבאה, שתיהה 7300 (נקרא seq כלומר 7000 ועוד גודל של pockets ראשונה) ושולח 400 ביטים של chunk. עכשו עברו TCP לכל connection מוגדר להיות מספר בתים שתהייו בתתהליך השידור, נניח שזה 700, הפ כבר נשלח ולא יודעים אם התקבלו או לא אז בשלב ה זה A נעצר. B מקבל את ה pocket ראשונה ושולח בחרזה אישור (ack) שהוא מכיל מספר אחד שזה seq + גודל של pocket כלו רמאשר שקיבל הכל עד 7300, A מסתכל ורואה ש B אישר לנו ה pocket עברה בהצלחה. עכשו A יכול לשולח ה pocket הבאה אבל עדין מהכח לאישור של pocket שנייה כי לא קיבל עליה אישור, ונניח שלא קיבל, למשל כי המידע הלך לאיבוד, או האישור עליה הלך לאיבוד. אבל כך וכך A לא יודע שהתקבלה ה pocket, אז עבר זמן מסוים שנקרא time-out ואומר שה pocket הלך לאיבוד ושולח אותה שוב אפשרי גם יחד עם pocket מס' 3 (תלו依 מימוש). בתרזה הזו ניתן לשדר pocket שוב ושוב, כדי לוודא

שכל ה pocket מגיעה. מבחן B היא מקבלת אותם בלי סדר. למשל אם B קיבל קודם כל 7300,400 אחר כך 7000,300 שהוא מקבל את זה והוא יודע שזאת pocket שהגיעה לא לפि הסדר והיא שותק, לא מאשר אותה, לא מעביר אותה לשכבה אפליקציה אלא מזריר אותה בבטן, שגיעה 7000 אז הוא מעלה אותה למעלה לשכבה אפליקציה והוא מאשר עד 7700 קלומר עד בית 700 קיבל את כל המידע מתחילה המספר שנקבע שזה 7000. לנכון TCP מבטיח ש pockets יעבר את ורק לפि הסדר.

### TCP (transmission control protocol)



TCP “sawtooth” behavior with various congestion control algorithms; occurs because drops are used to sense congestion. (BBR proposed by Google in 2016.)

על מנת להבין כמה ש TCP בעיתתי. צריך לשדר כמה שיוור מהר כמו כל פרוטוקול, אז בכלל שהוא חייב לדאוג לסדר, אז אם הוא סתם משדר pockets אז מה שקרה הוא משדר הרבה pockets, קלומר אל הוא שידר 1 מגה בית, וגילה שהידע הילך לאיבוד אז הוא משדר עוד פעם מהתחלה ועוד פעם, אז הוא רק מעmis על הרשות יותר וייתר, אז אם TCP סתם וישדר pockets שהוא יכול להוציא מ host הוא רק יגרום לאיבודים, ובמוקם להגיע לביצועים טובים, הוא יוכל לביצועים גראפים מאד, אז מה שעושים זה שבTCP מנסים להבין איזוב קצב לשדר, אז מה שעושים, זה משגרים בקצב נמוך ואז עולה וזה מתייצב על קצב שרשט מאבדת מעט pockets כפי שראה בתמונה (לא נראה איך עושיםה את זה). אם יש לנו זמן שבו pockets הולכות לאיבוד, אז הוא צדועשה צעד אחרונה ומshedר בקצב יותר נמוך וייתר עד שמתיצב. בתוך TCP יש מלא אלגוריתמים ומימושים (נקראים congestion control כלומר בקרת עומס). מוצג למעלה 3

אלגוריתמים כאלה Reno, Cubic, Reno במשל BBR נתן לראות שהוא מגיע לקצב גבוה ואז נראה היה איבוד של pocket וממש צנחה וлокחבלו יחסית הרבה זמן עד שה חוזר בחזרה אז אם נסתכל טוב על אלגוריתם זהו אז במשמעותו בסדר גודל של 16. אלגוריתם אחר Cubic הנפילות فيه יותר קטנות ורואים שאולי יש טיפה פחות נפילות ככלומר לא ממהר ליפול, אז הוא רץ בסביבת 16.5, ואלגוריתם BBR נתן לראות שהוא שמעט נפילות עמוקות, אבל נפילות עמוקות יהיו ממש עמוקות והה recovery שלו הוא מאד מאד מהיר, ככלומר הממוצע יהיה קרוב ל-18. ניתן לראות שאך אחד מהם לא אידיאלי.

## UDP (user datagram protocol)

- **The protocol that doesn't care**
  - Best effort service
- **Provides**
  - Datagram (as opposed to data-stream) abstraction
    - Datagram = chunk of bytes (still called segments here...)
  - Chunks might get lost or be delivered out-of-order
    - But per-chunk bytes integrity is supported (with checksum)
  - Apps decide if they're okay with that
- **Compared to TCP**
  - Simpler, lower latency, no congestion control (so can blast away)
- **Said to be “connectionless”**
  - Each chunk handled independently of others
  - No negotiation to establish communication ‘session’

הפרוטוקול UDP.

נזכר ש TCP מבטיח שידור dat הולך לאיבוד או נניח ששודר פאטקה 1, פאטקה 2, פאטקה 3 ועוד פאטקה ב הולכת לאיבוד וג התקבלה אז לא שולחים את ג לאפליקציה אלא מוכנים עד ש send ישלח פאטקה בשוב ואז שולחנים בו ג ביחד. למשל זה חשוב ב HTTP שטוענים אתר זה חשוב כי רוצים לראות את כל האתר ולא חלק ממנו, שמעבירים קובץ נרצה לקבל את כל הקובץ ולא כל חלק בנפרד. ולא יעוזר לנו להביא אותו בחלקים. יש לא מעט אפליקציות שעבורם גם סדר וקבלת כל היל

קריטית, ויש כאן שלא מוכנית לספק איבודים של מידע, מה הם? קודם כל סוג ראשון זה אפליקציות שרצות לדאוג בעצמם גם לסדר וגם לזה שהכל יגיע ויש לא מעט כאלה שלא משתמשות ב TCP כי יש לו הרבה בעיות, הוא נוח שנותן לנו מלא הbhות, וגם מטפל בכל מיני בעיות ברשות אבל יש לו הרבה בעיות, אז חלק מאפליקציות שעכורות חשוב הביצועים משתמשות ב UDP כי בדרך כלל מעביר מידע בקצב יותר גבוה. ויש סוג אחר של אפליקציות שלא דואגות בעצם לשום דבר אבל פשוט אם מידע לא מגיע אז אחר כך לא מעוניין אותם, למשל אפליקציות של streaming, שידור אונליין, רדיו. אפשר לחשב על כך שימושו מדבר ואז pocket הולך לאיבוד, ואז צריך להוכיח עד שעוד פעם הגיע, אז אם משאנו ידבר מה שקרה ואז הולכת לאיבוד וקופא ומחכים עד שעוד אחת מגיעה ויש עכשו הרבה כאלה שmaguit ביחיד והוא קופץ ל 5 שניות קדימה אז מدلגים. אז מה ש אפליקציות הזו אומרות בסדר, נחלק השידור ל pocket קטנים ואם אחד הולך לאיבוד אז לא יהיו מורגש כי חסרים קטיעים קטנים. אם ננסה להשווות שיחת וידיאו מול סרט יוטיוב, למה סרט יוטיוב יכול להשתמש ב TCP ושיחת וידיאו לא? מה הסרט יוטיוב עושה הוא מביא מראש כמה pockets, והוא מנגן הסירטון הוא יכול להביא מראש עוד ועוד, לעומת זאת בשיחת וידיאו אי שאלה הביא מראש דברים, כי הכל צריך להיות ב real time ואז אם משאנו הולך לאיבוד לא יוכל להגיד עד שימושו יביא את זה. לכן יש לא מעט אפליקציות שתכונות של TCP לא עוזרות להם, אז משתמשת ב UDP שהיא שואה וזה מאמין כי לשדר המידע.

## Domain & host names

- Computers are associated with hierarchical human-readable domain names, sometimes referred to as host names
  - [www.cs.technion.ac.il](http://www.cs.technion.ac.il), [csa.cs.technion.ac.il](http://csa.cs.technion.ac.il), [csm.cs.technion.ac.il](http://csm.cs.technion.ac.il)
    - Each of these is a name of a *single* host machine
    - So they're indeed "host names", but...
  - [www.google.com](http://www.google.com), [www.amazon.com](http://www.amazon.com), [www.cnn.com](http://www.cnn.com), [www.ynet.co.il](http://www.ynet.co.il)
    - Each of these is backed by multiple host machines, and...
  - [hagit.net.technion.ac.il](http://hagit.net.technion.ac.il), [benny.net.technion.ac.il](http://benny.net.technion.ac.il)
    - Each of these identifies a website in host net.technion.ac.il
    - So "domain" is a more general term
  - [https://en.wikipedia.org/wiki/Domain\\_name](https://en.wikipedia.org/wiki/Domain_name)
- In this lecture, we'll typically use the term host name
  - And usually assume that it corresponds to a single host machine

כעת יורדים למטה לשכבה 3 (שכבה ה  $\text{ip}$ ) נניח שרצוים להגיע לאתר של הפוקוטלה, מה עושים זה פותחים broswer וכותבים כתובת של הפוקוטלה. מה זו הכתובת הזו? זה נקרא **domain name**. זה בעצם שם שאנו מודבקים למחשב או אוסף של מחשבים. מחשבים אלו לא יודעפם לעבוד עפ' שמות אלא מספרים, אז אוקי נניח שיש לנו השם הזה, מה הלאה? איפה מחשב שלנו יודע איפה השירות web של טכניון יושב. שלב ראשון, צריכים לתרגם את השם של השירות לכתובת  $\text{ip}$ . עושים את זה בעזרת DNS כלומר יש לנו מחשב שתפקידו לחת את השם ולתרגם אותו לכתובת  $\text{ip}$ . מי שעשה את התרגום זה יכול להיות **router**, או יודע מי לפנות כדי לעשות את התרגומים. נני הצלחנו לתרגם שם  $\text{ip}$ , אז מה זה  $\text{ip}$ ? בדרך כלל יש שני סוגי  $\text{ip}$ . פורמט ישן זה כתובת  $\text{ip}$  בנוייה מ-4 בתים ששה מעט מאוד. ויש פורמט חדש שם כתובת  $\text{ip}$  היא 128 ביטים. אבל רוב servers לא תומכים בזה.

## IP addresses

- Presented as four 8-bit decimal octets separated by dots

– For example, the IP of [csa.cs.technion.ac.il](http://csa.cs.technion.ac.il) is

| Notation               | IP address                                                                                                                                                                                                                                                                                                                |          |          |    |   |          |          |          |          |
|------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------|----------|----|---|----------|----------|----------|----------|
| Decimal                | 2219057153 (not terribly convenient)                                                                                                                                                                                                                                                                                      |          |          |    |   |          |          |          |          |
| Binary (8x4 = 32 bits) | <table style="margin-left: auto; margin-right: auto;"> <tr> <td style="text-align: right;">132</td> <td style="text-align: right;">68</td> <td style="text-align: right;">32</td> <td style="text-align: right;">1</td> </tr> <tr> <td>10000100</td> <td>01000100</td> <td>00100000</td> <td>00000001</td> </tr> </table> | 132      | 68       | 32 | 1 | 10000100 | 01000100 | 00100000 | 00000001 |
| 132                    | 68                                                                                                                                                                                                                                                                                                                        | 32       | 1        |    |   |          |          |          |          |
| 10000100               | 01000100                                                                                                                                                                                                                                                                                                                  | 00100000 | 00000001 |    |   |          |          |          |          |
| Dot-decimal            | 132.68.32.1 (a bit more convenient)                                                                                                                                                                                                                                                                                       |          |          |    |   |          |          |          |          |

- The protocol that implements IP addresses is... IP

- (Recall that IP = internet protocol)
- It's the network-layer (L3) protocol
- On top of which TCP & UDP (L4) are implemented
  - IP identifies host machines
  - TCP/UDP identify processes' individual communication channels on hosts, as we'll see next

כתובת  $\text{ip}$  זה מספר של 32 בתים, ואפשר לייצג אותו על ידי מספר דיסימלי אבל אי אפשר לזכור אותו. בדרך כלל בשבייל מכונה זה 32 ביטים אבל בשבייל אנשים זה לקווים אותם ומחלקיים לבתים וכאל בית רושיםם בצורה נפרדת. ויש נקודות לצורך נוחות כדי לדעת שזו בית הבא. למשל לדוגמה מצוין כתובת של שירות [csa.cs.technion.ac.il](http://csa.cs.technion.ac.il). אז בעצם שפוניים

למחשב פונים לפיקותה זו, עוד לא יודעים איל מгиיעים אליו בעזרת כתובות זו אבל יש לנו כתובות זה ואיכשהוא הנס קורה.

## Sockets

- **The parties that communicate across the network are**
    - Apps that run on hosts (e.g., browsers)
  - **They communicate through**
    - Socket file descriptors, created via the `socket()` syscall, instead of `open()`
    - A sockfd constitutes a communication endpoint
  - **read()-ing and write()-ing through socket fds**
    - Translate to receiving & sending data (duplex)
  - **There's also more specific system calls (see their man)**
    - `ssize_t send (int sockfd, const void *buf, size_t len, int flags)`
    - `ssize_t recv (int sockfd, void *buf , size_t len, int flags)`
  - **And their scatter-gather versions**
    - `ssize_t sendmsg (int sockfd, const struct msghdr *msg, int flags)`
    - `ssize_t recvmsg (int sockfd, struct msghdr *msg , int flags)`

בשביל להבין את הדיון צריך להבין מה זה socket. שאנו בטור משתמשים רוצים לקרוא מידע מהקובץ או לקרוא, אנחנו פונים למערכת הפעלה, ואז אומרם לה זה שם הקובץ ואז תכני אותו לעבודה. ואז יש לנו מספר זהה מספר בעצם ב-FDT. ואני בטור משתמשים נרצה לקרוא מכניסה מספר 17. מערכת הפעלה יודעת באיזה קובץ מדובר. מבחן תקשורת רוצים לעשות אותו דבר,

רוציה להגדיל מערכות הפעלה לייצור תקשורת עם מחשב מרוחק ואז תחזיר מספר integer בדיקות כמה בקבצים ועכשוו בעצם ששולחים מידע ומקבלים מידע נרצה להשתמש במספר זהה לכותב אליו ולקראו ממנו. זה בדיקות מה שמערכת הפעלה עשויה, יוצרת socket, ונותנת לuser את מספר socket ועכשוו משתמש שרווצה לכתוב הוא מעביר כפרמטר את מספר ה socket בנוסף לזה חיש קרייאות מערכת הפעלה שמיועדות ל networking, למשל send, receive שמקבלות כפרמטר את מספר ה socket, וזה שמאפיין את ה connection מבנית המשתמש. איך היא מראה בתוך מערכת הפעלה? זה שהוא שמכיל 5 שדות.

7. פרטוקול
8. IPs קיזור ל ip source
9. Ports קיזור ל port source
10. של המחשב שמתהברים אליו ip desktop יש גם כלומר IPd
11. גם כלומר port desktop portd

זה בעצם מה שיש ב socket, והוא אמור לאפיין

## Ports

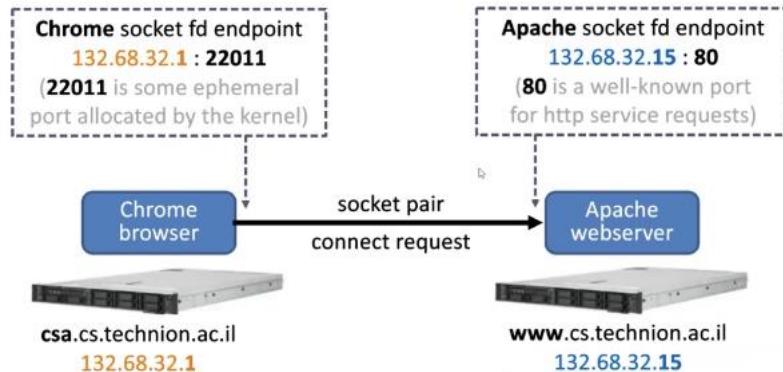
- **Simultaneously, on the same host, there can be**
  - Multiple communicating processes, each utilizing multiple sockfds
  - So IP addresses aren't a sufficient identifier for transmitted data chunks
- **To avoid ambiguity, each sockfd is associated with a**
  - Port, unsigned 16-bit integer that uniquely identifies the sockfd
  - Every transmitted chunk is associated with IP address + port
- **Ports can be either**
  - Ephemeral = dynamically allocated by the kernel, or
  - Well-known = standard, predetermined, known values
    - Ports ≤ 1023 are “reserved” (for privileged processes)
- **For example, http & https traffic flows via ports 80 & 443**
  - <http://www.google.com> ⇔ <http://www.google.com:80>
  - <https://www.google.com> ⇔ <https://www.google.com:443>
- **See more well-known ports in**
  - [https://en.wikipedia.org/wiki/List\\_of\\_TCP\\_and\\_UDP\\_port\\_numbers](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers)

עכשו מה זה port?

זה מושג ומרטואלי להלוטין, מערכת הפעלה יש לה טבלה של ports יש טבלה נפרדת ל TCP, ובבלה נפרדת UDP שהוא בגודל k64, כי כתובות של port זה 16 ביט לנכון  $2^{16}$  ביט גודל הטבלה. כל פעם שאנו פותחים תקשורת, נניח יש לנו שני מחשבים ותהליך 1 במחשב 1 רוצה לדבר עם תהליך 2 במחשב 2. אז מה שקרה זה המערכת הפעלה צריכה לחתם במחשב 1 להקצתו אחד ה port הפנויים למשל 2079 ובמחשב שני גם מקצת port נניח 93 ואז ה socket שיוצר זה בין ip של מחשב אחד למחשב שני לבין port 2097,93 אז שהוא משווה מערכת הפעלה בחורה עבור socket שיוצרה. יש socket סטנדרטים למשל אם רצח לחייב שרת http או חיבבים להשתמש ב 80, port, כל הצד השני יודע לפנות אליו. יש port שיכולים לבחור בצורה אקראית, למשל באלנו

להקם שרת AMAZON על port מספר 9999. ה-port ימ הסטנדרטים עד 1023 וכל מה שמעל יכולם להתחלף ואפליקציות יכולות להשתמש בהם איך שרצו.

## Example



בעצם שיצרים תקשורת, ונניח שיש לנו webserver, ויש לו איזשהו ip ובסגול שרצים להקים webserver נשתמש ב port מספר 80. עכשין המחשב שרצה להתחבר יש לו ip שלו והוא יודע לאיזה server רוצה לפנות. איך הוא יודע? הוא פונה ל DNS (מה שתרגם לכתובת ip) וմבקש ממנו לפנות ל server זהה, והוא מוחזיר ip של מחשב השני. אז אנחנו רוצים לפנות ל ip הזה שסומן בכחול למעלה. אז בעצם נרצה ליצור תקשורת TCP, למה? כי רוצים לשרת פרוטוקולים http מ 132.68.32.1

ל

132.68.32.15

עכשו מבחינת ה ports, אנחנו יודעים ששרת web נמצא על port מספר 80, אם היינו אנחנו שרת שהקמנו או צריך לדעת על איזה שרת שמננו את ה server הזה. אז יודעים שרצים ליצור תקשורת על port מספר 80, מה עם port שלנו? אנחנו פונים למערכת הפעלה, ואומרים לה ליצור socket עם הפרמטרים האלה, והיא בוחרת מכל ה ports של ה TCP שיש לה אחד פניו למשל 22011 ויוצרת מזה socket, אז מ ip

132.68.32.1

ל

132.68.32.15

ו port

22011

ל

80

ל

ואז פונה למחשב המרוחק ואומרת שרוצה ליצור תקשורת עם פרמטרים האלה, בצד השני נוצר socket הפוך קלומר מקו

132.68.32.15

ל

132.68.32.1

ו port ו מוקם

80

ל

22011

או בעצם מכל אחד מצדדים נוצר socket ובעזרתם שניהם יכולים לדבר אחד זה עם זה. זה לא משנה מי יצר אותו, וכל אחד בעזרת ה socket יכול להעביר מידע.

## Our echo application-layer protocol

|   | Simplistic echo client                                 | Simplistic echo server                                    |
|---|--------------------------------------------------------|-----------------------------------------------------------|
| 1 | Connect to the server using TCP                        | Accept connection from a new client                       |
| 2 | Write (= send) a short byte-sequence message to server | Read (= receive) up to N characters from client           |
| 3 | Read (= receive) from server at most N characters      | Write (= send = echo) these characters back to the client |
| 4 | Print the received characters                          | Print client's host name                                  |
| 5 | Exit                                                   | Go to 1                                                   |

از אנו רוצים להסתכל על אחד ה pattern של עבודה. יש לנו server ויש client שמתחבר אליו, זה לא תמיד המצב, לא תמיד עובדים עם client server. אז צריך לאריך לserver ליצור תקשורת, ואז server אפשר, ואז יכולים להעביר מידע אחד לשני, שנייהם יכולים לעשות write, read. ואז סוגרים התקשרות ונגמר הסיפור. אנחנו נתבונןecho server למשל הם סימנו לעבוד וסוגרים את התקשרות ובזה נגמר הסיפור.

מה זה echo server? נניח שקם client server ואז יוצר תקשורת עם ה server, ושולח hello buffer נניח hello, ואז הוא מקבל hello ומחזיר אותו למשתמש ומשתמש מדפיס מה שקיבל מserver.

```

/*
 * Implement the protocol's server side. The host of this
 * server and the given port must be known to the client
 */
void echo_server(uint16_t port)
{
 const int N=256;
 char buf[N];
 int k, clifd, srvfd = tcp_establish(port);

 for(;;) {
 DO_SYS(clifd = accept(srvfd, NULL, NULL));
 DO_SYS(k = read (clifd, buf , N));
 DO_SYS(write (clifd, buf , k));
 print_peer("client from:", clifd); // client's host
 DO_SYS(close(clifd));
 }
}

```

OS (234123) - networking

ניתן להסתכל על המימוש echo server יש לנו בהתחלה tcp establish מכין את ה server לעבוד בתור server, וזה אומרת לנו מודיעים למערכת הפעלה לתת לנו port מספר x ואז כל pocket שmagiu ל x להעביר את זה אלינו. בשלב זהה ה server שלנו מוכן לקבל בקשות התחברות מה client. ואז עולמים פועלות accept שהיא פעולה חוסמת, ככלומר server אומר למערכת הפעלה שהוא ממתין לבקשת התחברות, ולא עושה כלום אלא ממתין וברגע שיש בקשה התחברות מבקשת העיר אותה ואז היא תטפל בה, אז היא בעצם חוסמת ומחזירה את המספר של ה socket שבעזרתו ניתן לעשות תקשורת. גם tcp establish מחזיר מספר של socket, אבל ה socket כאן לא מלא אלא חלקי, כי בתור server לא יודעים מי יפנה לנו לכן הוא יכול פרטוקול ומה ה ip שלו ואיזה port מאזין או לקבל בקשות תקשורת. Accept לוקחת את זה ויוצרת socket חדש שאילו מעתקה את ה השדות של ה socket קודם ומוסיפה לשם את ה port של ה client. עכשין מה שנעשה ב server זה read מתוך ה socket המלא לתוך buffer של זה בתים. ואז היא מחזירה כמה בתים קראנו ואז עושה write של socket ה client. ואז בסוף סוגרים ה socket, ומacha לעוד בקשות התחברות מ clients אחרים או אותו .client

```

/*
 * Implement the protocol's client side. The given host
 * and port must identify the server
 */
void echo_client(const char *host, uint16_t port)
{
 char buf[256], msg[] = "hello\n";
 int k, fd = tcp_connect(host, port);

 DO_SYS(write(fd, msg, strlen(msg)));
 DO_SYS(k = read (fd, buf, sizeof(buf)));
 DO_SYS(write(STDOUT_FILENO, buf, k));
 DO_SYS(close(fd));

 exit(EXIT_SUCCESS);
}

```

## מה echo client עושה?

קודם כל מפעילים `tcp connect` שמקבלת למי רוצים להתחבר, ואת השם של הclient ואת ה `port`, על איזה `port`? למשל נרצה להתחבר ל `http` של `www.cs.technion.ac.il` הפקולטה והיא מחזירה `socket` שביצרתו נוכל לתקשר, ואז כתובים לתוכו `massage` ואז קוראים מותך `socket` לתוכו `buffer`, ואז מקבלת מהזירה מספר התווים שנקראו בפועל ואז כתובים על המסך `k` תווים מותך `buffer`, ובסוף סוגרים את ה `.socket`

## Creating TCP socketfd: both sides

| server                  | message                        | client                        |
|-------------------------|--------------------------------|-------------------------------|
| srvfd = socket(...)     |                                | sockfd = socket(...)          |
| bind( srvfd , port )    |                                |                               |
| listen( srvfd )         |                                |                               |
| <i>loop:</i>            |                                |                               |
| clifd = accept( srvfd ) | request service<br>(transport) | connect( sockfd , host+port ) |
| read(clifd )            | request<br>(application)       | write( sockfd )               |
| write( clifd )          | response<br>(application)      | read( sockfd )                |

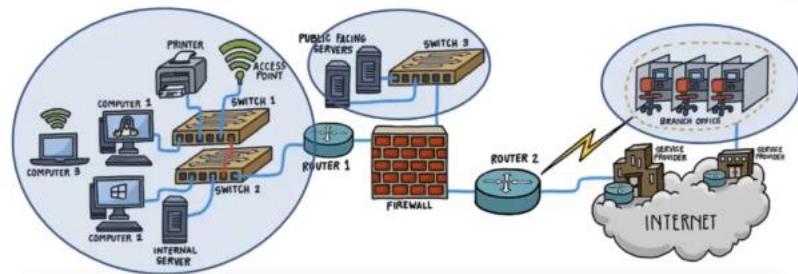
נרצה לדבר מבחן אלגוריתמיקה מה עושים. מה שעשוה client, מפעיל פונקציה socket שמייצרת socket שהוא לא מלא ולא מכיל הרבה דברים, יכול בגודול את ה ip שלו. אי אפשר להשתמש בו כי הוא לא יודע כלום, מערכת הפעלה לא יודעת לאן להתחבר, ולא נתנה לו שום port. מתי זה קורה? בclient אחרי שייצרנו ה socket ריק הזה אנחנו מפעילים פונקציה connect שמקבלת ה host, port של ה server. ובעצם מלאת את השדות החסרים בעזרתו ידעת מה ה host, port, ip של ה server, והוא בוחרת בצורה אקראית את port אחד מהפנויים וצריך להגיז באיזה פרוטוקול אנחנו עושים connect, או עבור client מייצרת את ה socket המלא שבחרתו נוכל להעביר מידע, ומה שוקרה שם מתחת לשטיח המערכת הפעלה, חלק מ connect זה שמערכת הפעלה במחשב שלנו צריכה למכת למחשב של ה server ולודא שהוא מוכן, כי יכול להיות שהוא לא רוצה, מה ה server עשו באותו זמן, ה connect לא יצליח עד ש server סיים תהליך אתחול שלו ויסכים לקבל את ה connection. ה server גם מתחליל socket שלא מכיל כלום, אחר כך עושה bind שבו מסpter ל server. ומערכת הפעלה באיזה פרוטוקול ואייזה port צריך להאזין, מעכשוו כל ניסיון התחבורות ל port מספר 70 גיע אליו (server) בשלב זהה לוקחת את ה socket שנוצר ומוסיפה אליו כל הפרטיהם של ה server. זה שזה host, protocol, port, ip של ה server. ודבר אחרון שעושים זה listen שואמר הדבר הבא, נניח שהתקבלה בקשה לצורך תקשורת, אז

מתחלים לטפל בה בזמן שטפלים בה לפני שהסכמנו לטפל בבקשת הbhא, מגע עוד בקשות, אז שמים אותם בתור, וזה חייב להיות עם גודל סופי, אנחנו צריכים לדעת מה הגודל שלו, ככלומר כמה בקשות תקשורת לא אושרו שיכולות להיות בתור ולא שאושרו הם לא בתור, אז במקרים מעבירים למערכת הפעלה עברו ה socket הזו גודל של תור בקשות זה x. וזה בעצם מה שקרהנו לו `tcp_establish`.

דבר הבא זה `server` שהוא מוביל לקבל בקשה התחברות הbhא, ואז נוצר `socket` חדש ושולח ל`client` ואומר לו שבקשה שלך מאושרת, ותשמש ב`socket` זה לעשות תקשורת. בדרך כלל יוצרים חוט חדש שטפל בתקשרות וחוט ראשי חזרה לעשוות `accept` לבקשת bhא של התקשרות.

## LAN (local area network) connectivity

- **LAN examples**
  - Computers in your home
  - CS server room
  - Computers where you work
  - Computer in a data center (may include multiple LANs)
- **Devices in the LAN are connected**
  - With wires, or wirelessly, or both



עכשו מבחן הטופולוגיה של הרשת, אם נסתכל על איך הרשת בנוייה אז יש הרבה מאד מרכיבים, זה לא מחשב `client` ו `server`, בפרט הם יכולים להיות לא מחוברים, אז יש מה שנקרא **LAN**  
(Local area nerwork)

זה רשת מחשבים פרטית, למשל ביתית כלומר יש לנו מחשבים שמחברים על ידי כבליים, wifi, הם נמצאים ביחד באותה רשת ביתית. ושרוצים למשת לגש למחשב למחשב שלנו בעבודה, אין לנו בעצם חיבור פיזי אלו רשתות נפרדות. מה שמחבר בין ה

lan זה routers, מה זה routers? איך בזאת מחשב מתחבר לרשת? כרטיס רשת כלומר חומרה שמאפשרת להתחבר. Router זה התקן שיש לו לפחות שני כרטיסי רשת כלומר מחובר לשתי רשתות, ו יודע לשלוח מרשת אחת לשניה, אז בעצם בשביב של pocket תעביר מעבר מקום למקום היא חייב לעבור כמו routers בדרך.

## MAC (media access control) address

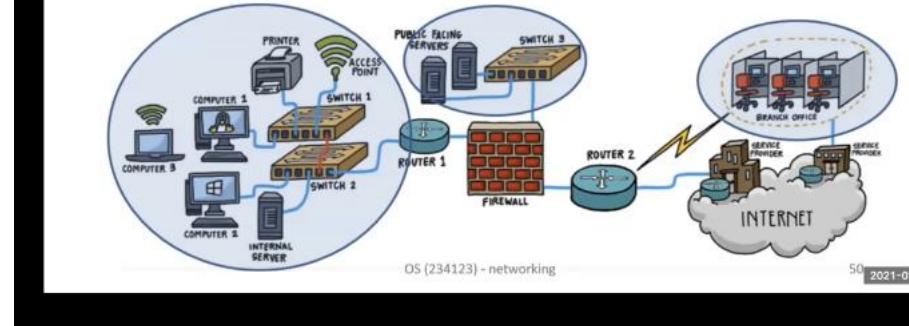
- **A unique 48-bits number**
  - Identifying each Ethernet component
  - Burned in ROM of NIC / Switch / AP
  - Used to switch Ethernet frames to their destination within the LAN
- **Notation**
  - 6 pairs of hex digits, e.g.,
    - C8:5B:76:EA:B8:A0
  - Of which the device manufacturer is allocated 3, exclusively, e.g.,
    - CC:46:D6 – Cisco
    - 3C:5A:B4 – Google
    - 00:9A:CD – HUAWEI
  - So manufacturer ensures MAC uniqueness

לכל כרטיס רשת יש מספר MAC שהוא כתובה שאותו הוא נולד והוא אמור למות. כלומר הוא לא אמור להשתנות. כתבות MAC זה בעצם 48 ביטים, שגורם להרבה בעיות ביצועים. ואין הסבר ממה נובעת הבחירה. חלק ממנה מופיע את היצורן של כרטיס רשת וחלק זה משווה פרטיו לכל כרטיס, כלומר לא אמור להיות שני כרטיסים עם אותו כתובה.

## IP & Routers

- **Problem**

- LAN hardware components typically “speak” L2 (Ethernet)
  - A “language” that works exclusively within the LAN
- How then, can node@LAN-A send messages to node@LAN-B?
  - Technically, switches@LAN-A can’t talk to switches@LAN-B
  - And in addition, note that...

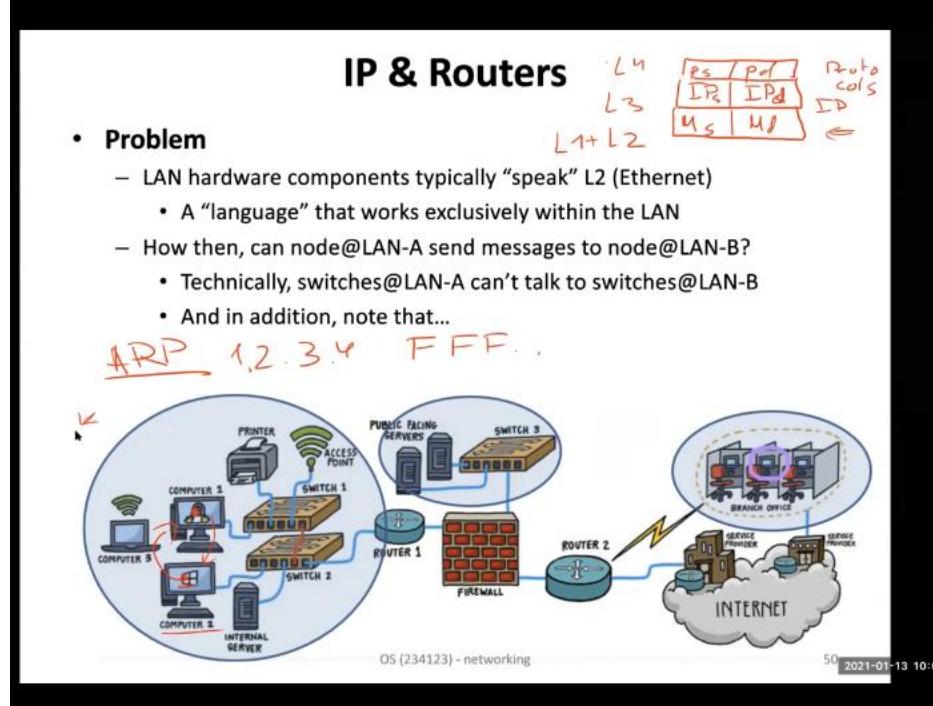


נמסה להסתכל על התמונה ולהבין איך הכל עובד, למה צריך MAC, ip וכל הדברים האלה.

מתחילה מדוגמה פשוטה, שבה מחשבים מנסים לדבר אחד על השני באותה רשת. יכול להיות שמחשב ראשון יש לו MAC של מחשב שני. ואז שולחים מכתבת MAC2 ל MAC1. נזכיר שיש לנו שכבה L4 שיש בה port של source ו port של distention.

ושכבה L3 שהיא שכבה ip ושכבה L2 ו L1 שביהם יש MAC של source ו MAC של distention. גם בתחום הרשות הlokality אנו יודעים את שם המcona ואו פונים DNS והוא אומר לנו את ה IP של המcona, למשל 1234 ואז שולחים חבילה שנקראת ARP שבה שמים כתובת IP של כתובת הזאת אותו מ MAC שלנו ל Mac של FFF ושולחים אותה לכולם, לכל מחשב שנמצא בתקשורת, וכל אחד מקבל החבילה ARP זאת חבילת ARP כלומר מישו מחפש מחשב ואז מקלף ומסתכלים על כתובת IP של מחשב 1 ומחשב 2, ומהפש 1234 נניח שהוא 1235 או זה לא חבילת לכן זורק אותה ולא עונה, נניח שאנו מחשב 3, ואז הוא מקבל חבילה ומקלף ורואה 1234 או הוא מגלה שמחפשים אותו ועכשו מחזיר תשובה לאותו מחשב ש MAC שלו יודע, כי שולחה החבילה הוא מילא את ה MAC של המחשב ששלה ולא שלו, חcn מחזיר תשובה את כתובת MAC שלו כך ועכשו מחשב 1 יודע לפנות למחשב 2 ולהעביר חבילות כי יש לו

את ה MAC ו IP. עכשו מה קורה במצב יותר מסובך שבו שולחים חבילה ממוחשב 1 למוחשב שני כאשר שניהם מרוחקים ממש למשל כמו בתמונה

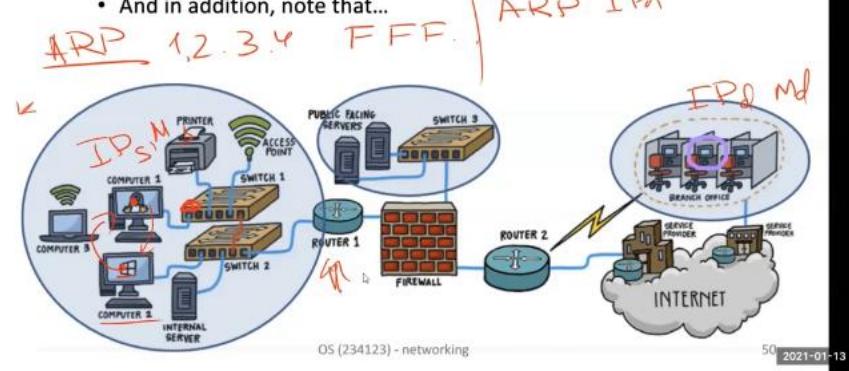


מה שקרה עכשו, זה שמחוב מסומן באדום יודע את ה ip או פונים ל DNS וمبקש מה כתובת ip של ה server, لكن אנו יודעים את ה destination ip, لكن צריך לדעת מה ה MAC שלו כי הוא לא נמצא באותו רשת, אבל גם אם יודעפ איז איך הגיע אליו MAC Source? נניח ש למחוב ורוד יש MAC distention ולי יש MAC Source, אז נשלח חבילה שמיודעת לMAC distention אז לאף אחד אין את ה MAC זהה, אז איך החבילה הגיע למחשב מרוחק. אז מתחילה כך, שולחים חבילה ARP שמכילה ip של המחשב מקבל אותה ומכלף אותה ומסתכל על ip והוא מוצא שלא שיין distention, אז מי מסתכל בזורה אחרת?

## IP & Routers

- Problem

- LAN hardware components typically “speak” L2 (Ethernet)
  - A “language” that works exclusively within the LAN
- How then, can node@LAN-A send messages to node@LAN-B?
  - Technically, switches@LAN-A can’t talk to switches@LAN-B
  - And in addition, note that...



הrouter שמסומן בחץ אדום, קיבל את החבילה ומקלף את שכבה L2, L1 ומסתכל על קו, כמובן הוא לא שלו, אז הוא הולך לטעלת ניוט שלוי ובודק אם יודע מה לעשות אליו ובטבלאות אלו יהיה כתוב لأن להעביר אותה לאיזה רשת. אז router שולח ARP למחשב 1 ואומר שהחbillah לכתובת dist ip dist ip משלחה אותה ל 1.1.1.1, אז אז החbillah ראשונה שיוצאת זה מ

Ip source -> ip dist

MAC source -> MAC1 (router MaC)

עכשו החbillah מגיעה ממחשב אחד(router).

אז עכשו router רוצה המשיך לשולח אותה ולא יודע لأن, כתוב אליו לרשף אחריו אבל מה לעשות אליה? אז יוצאת חbillah חדשה שאומרת אני רוצה לשולח ברשף הזאת חbillah ל dist ip או מי יודע מה לעשות עם זה? אז router מתעורר ואומר שהוא יודע (לכל router יש שני MAC לפחות), אז חbillah יוצאה מ

Ip source -> ip dist

MAC 2 (router MAC) -> MAC3 (router MaC)

עכשו חbillah שלישית עם header MAC dist MAC4 יצאם שונים

או זה הפקיד של כתובות, כתובות MAC זה למצוא מחשב ברשות פרטיה, וכתובה ip בשבייל להעביר חבילות דרך ה router ברשותות שונות.

## The “TCP/IP” protocol stack – recap

- **L1 = physical-layer**
  - How bits transmitted (EE level)
- **L2 = link-layer**
  - Ethernet (frequently)
  - Logical communication between hosts across a LAN, with MACs
- **L3 = network-layer**
  - IP, which provides global address space
  - Logical communication between hosts across the WAN, with IPs
- **L4 = transport-layer**
  - TCP & UDP, stream abstraction
  - Logical communication between processes across the WAN
- **L5 = application-layer**
  - Numerous protocols that utilize L4

# מערכות הפעלה

## גורג סלמאן

### תוכן העניינים

|    |                                       |
|----|---------------------------------------|
| 4  | תרגול 1                               |
| 12 | תרגול 2                               |
| 14 | תהליכיים בלינוקס:                     |
| 19 | קריאה המערכת ( <i>waitpid()</i> )     |
| 19 | קריאה המערכת ( <i>exit()</i> )        |
| 21 | קריאה המערכת <i>execv</i>             |
| 21 | קריאות המערכת <i>:getppid, getpid</i> |
| 23 | מבנה הנתונים לניהול תהליכיים.         |
| 23 | מ吒ר התהליכים:                         |
| 24 | ניהול קשרי משפחה בגרעין:              |
| 25 | רשימת התהליכים:                       |
| 27 | תרגול 3                               |
| 27 | תזכורת.                               |
| 31 | טיפול בסיג널ים:                        |
| 32 | סיכום:                                |
| 33 | <i>FD (File descriptors)</i>          |
| 34 | <i>FDT (File descriptor table)</i>    |

## מערכות הפעלה

|    |                                |
|----|--------------------------------|
| 34 | ערוצי קלט פלט סטנדרטים         |
| 37 | קריאה נתונים מקובץ             |
| 38 | כתיבת נתונים לקובץ:            |
| 39 | <i>.File – objects</i>         |
| 39 | שילוב קלט/פלט בין חוטים.       |
| 40 | שילוב קלט/פלט בין תהליכיים.    |
| 41 | <i>file – object</i> שחרור     |
| 43 | 43 - מה ראיינו עד עכשו.        |
| 43 | <i>pipes</i> בLinux            |
| 44 | יצירת <i>pipe</i> חדש          |
| 47 | כמה צריך לסגור קצויות מיותרים? |
| 47 | קריאה המערכת ( <i>dup()</i> )  |

52

תרגול 4

|    |                                            |
|----|--------------------------------------------|
| 54 | : <i>Linux boot sequence</i>               |
| 55 | : <i>BIOS (basic output/Output system)</i> |
| 56 | : <i>MBR( master boot record)</i>          |
| 57 | . <i>GRUB(GRand Unified Bootloader)</i>    |
| 59 | טעינת גרעין Linux:                         |
| 61 | מודולים ( <i>modules</i> ).                |
| 62 | יתרונות השימוש במודולים.                   |
| 62 | חזרה.                                      |
| 65 | טעינת המודול.                              |
| 65 | העברת פרמטרים למודולים.                    |
| 66 | גישה לנתוני גרעין.                         |

## מערכות הפעלה

|    |                                         |
|----|-----------------------------------------|
| 69 | התקני תווים פיקטiviים.                  |
| 72 | שדות חשובים ב <i>file operations</i> .  |
| 74 | IMPLEMENTATION DRIVERS באמצעות מודולים. |
| 80 | רישום דרייבר חדש.                       |
| 80 | הקצת מספר ראשי.                         |
| 82 | מה הולך לקרות שmericits.                |

|     |                                        |
|-----|----------------------------------------|
| 86  | <b>תרגול 5</b>                         |
| 86  | מבוא לזמן תהליים.                      |
| 89  | . <i>FCFS</i> אלגוריתם                 |
| 90  | <i>SJF</i> אלגוריתם                    |
| 90  | <i>SJF</i> אופטימלית של                |
| 91  | <i>SRTF</i> אלגוריתם                   |
| 98  | זמן תהליים בלינוקס.                    |
| 99  | אלגוריתם זמן של תהלי זמן-אמת           |
| 101 | הוספת תהליים לתור הריצה                |
| 102 | чисוב <i>time slice</i> .              |
| 103 | האבולוציה של זמן תהליים בלינוקס.       |
| 104 | <i>Completely Fair Scheduler (CFS)</i> |
| 105 | מבנה הנתונים של <i>CFS</i> :           |
| 106 | מה קורה במערכת עםosa?                  |
| 107 | יציאה וזרה מהמתנה.                     |
| 107 | עדיפות.                                |
| 108 | קצת נוסחות                             |

## 4 מערכות הפעלה

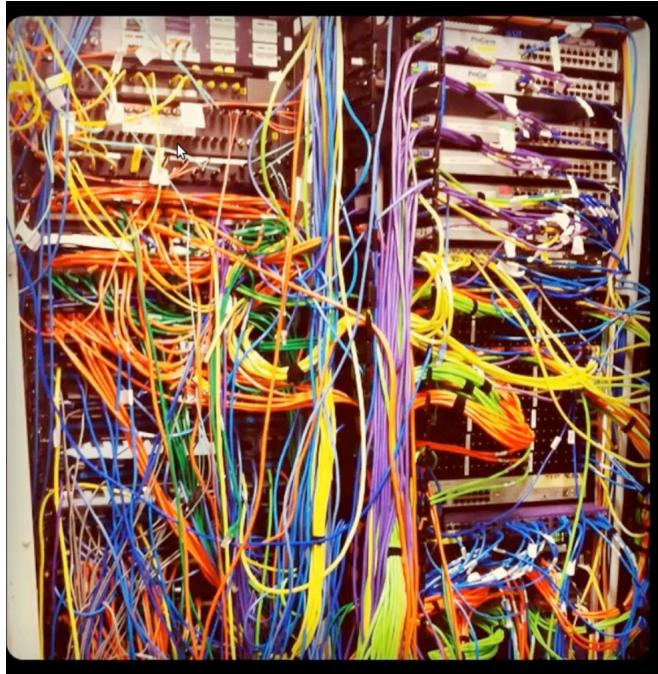
### תרגום 6

|     |                                 |
|-----|---------------------------------|
| 109 |                                 |
| 113 | מהי החלפת הקשר?                 |
| 113 | יתרונות וחסרונות של החלפת הקשר. |
| 114 | שני סוגי של החלפת הקשר          |
| 115 | החלפת הקשר כפואה (= הפקעה).     |
| 116 | הפקעה ( <i>preemption</i> )     |
| 120 | היכן נשמר האזכור של התהלייר?    |
| 124 | שלבי החלפת ההקשר                |
| 132 | <i>:do-fork</i>                 |

|     |                            |
|-----|----------------------------|
| 138 | . חוטים.                   |
| 140 | חוטים ( <i>threads</i> )   |
| 141 | חוטים יכולים לשפר ביצועים  |
| 143 | תקשורת בין חוטים           |
| 144 | <i>pthreads</i> ספריית     |
| 158 | קטע קרייטי                 |
| 158 | מנעולים                    |
| 159 | מנעולים ב- <i>pthreads</i> |
| 162 | . מימוש של מנעל.           |

### תרגום 1

## מערכות הפעלה



תקייר. אנו רואים מערכת מאוד מורכבת, ולא רואים מי נגד מי. כדי להסביר איך חלך עובד אנו אמורים לדעת איך שאר החלקים עובדים. נכון, קשה להסביר איך כל רכיב עובד בנפרד.

### מהי מערכת הפעלה?

היא שכבת תוכנה שמקשרת בין הישומים של המשתמש לחומרה.

### יישור קו: מה זה מעבד ?

(1) ה**קלט** של המעבד הוא פקודות מכונה (asmבל) ומבצע אותן אחת אחריה שנייה בצורה סדרתית.

(2) כל מעבד מורכב מספר **LIBOT core** חישוב עצמאיות שרצות במקביל.

(3) תדר אפייני של המעבד הוא  $3GHz$  וזה גישה טיפוסי לרגיטרים זה  $sas1$  (רגיטרים + מטמוניים).

(4) רגיטר זה ייחית זיכרון מאוד קטן שנמצא על המעבד. (למשל ב- $Riscv$  יש 32 רגיטרים).

## מערכות הפעלה

6

מה זה זיכרון ?

(1) מאחסן מידע ונדיפה. (ברגע שנכבה את במחשב כל המידע ייעלם).

(2) נפח אופייני  $8Gb$  וזמן גישהש טיפוסי  $as00.1$ .

הערה. טעות נפוצה היא שלכל ליבת יש זיכרון משלה. זה לא נכון, כל הליבות נגשות אותו זיכרון.

מה זה דיסק ?

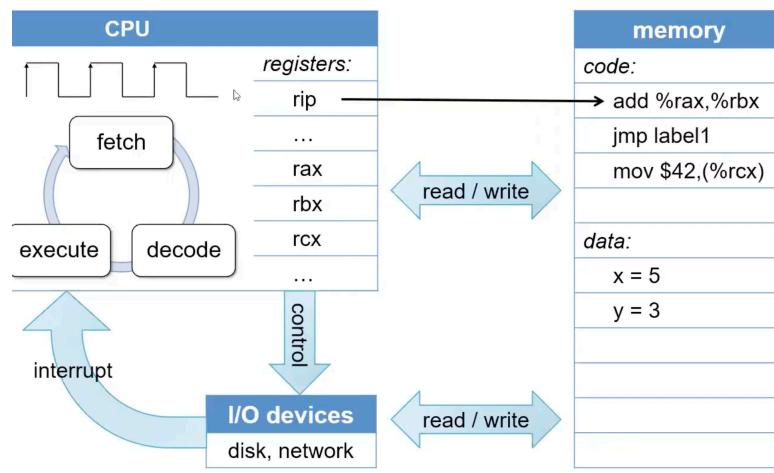
(1) זיכרון ודיקס לא אותו דבר.

(2) דיסק הוא אמצע לנדיניות כלומר כלומר מאחסן מידע אמיד ולא ייאבד שנכבה המחשב.

(3) הבעייה היא שדיסק יותר איטי.

(4) נפח אופייני  $1TB$  וזמן גישהש טיפוסי  $1ms$ .

המעבד לא יכול לעבוד עם מידע דיסק. הוא רק יכול לעבוד עם זיכרון. לכן, יעביר את המידע מדיסק לזכרון ויעבד אותו בזכרון.



האם הזיכרון נחשב כהתקני קלט פלט ? לא, דוגמא להתקני קלט פלט  
הן: מקלדת, עכבר, מסך ..

תפקידו של מערכת הפעלה:

(1) לחלק את משאבי חומרה בצורה יעילה והוגנת בין המשתמשים.

(2) תונן כדי שהוא לא משתמש בהרבה חומרה לעצמה אחרת לא  
עשינו שום דבר.

## מערכות הפעלה

להציג למשתמשים אבטרכזיות וממשקים (API) כדי להקל פיתוח אפליקציות.

להגן על המידע של המתמשים מפני משתמשים או תוכניות אחרות זדוניות.

לשמר על מידע המשתמשים מפני פילות חומרה. (למשל בעבר אם היוו עובדים על קובץ *word* ונתקע היינו מאבדים אותו).

עיקרון הוירטואלייזציה. הבעה היא חלוקת משאבים. נניח לヨיסי יש מעבד יחיד וזכרון יחיד. אבל יויסי רוצה להריץ הרבה הרבה אפליקציות בו זמנית. על איזה פקודות יעבד המעבד?

פתרון. וירטואלייזה של משאבים פיזיים.

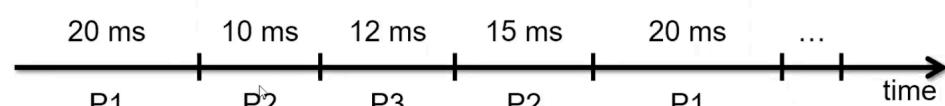
- מערכת ההפעלה מציגה למשתמש גרסאות וירטואליות של המעבד והזיכרון.

- למשרונות יש מעבד פיזי אחד מערכת הפעלה תציג אשלייה למשתמש ולתוכניות כאילו יש רובה מרחב זיכרון והרבה מעבדים.

- אם יש לי זיכרון אחד לכל אפליקציה יהיה מרחב זיכרון וירטואלי משלה. וכך לא יוכל לגשת לזכרון אחת של השניה.

- צריך להוסיף תקורה. למשל אם אני עטב בIBM ומקבל שכר  $x$  בשעה יש את תקורה של ללכמת לעבודה ולהזoor ולא מleshimim לי עליה אז זמן זה הוא תקורה.

אין אפשר לנקח מעבד יחיד ולהציג הרבה מופעים וירטואליים שלו הטריך הוא לחלק את זה בציר זמן. למשל,



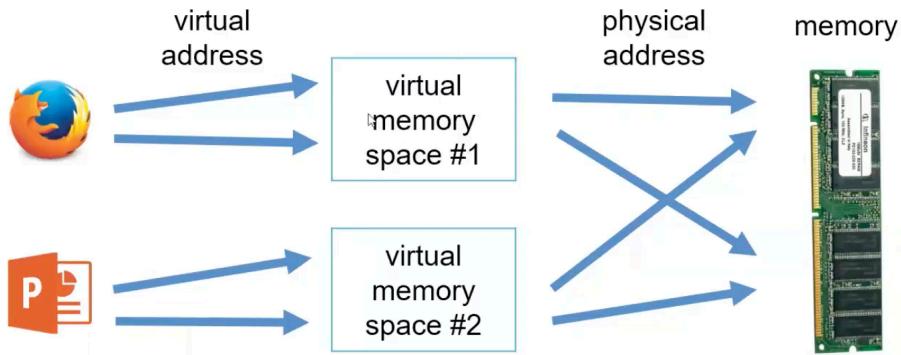
המעבד

נותן לתחילה  $p_1$  זמן של 20ms ועבור תחילה  $p_2$  זמן של 10ms ועבור תחילה  $p_3$  זמן של 12ms.

## מערכות הפעלה

- מערכת הפעלה מחלקת את הזמן של המעבד הפיזי בין התהליכים השונים.
- הגאים מחייבים מהירות בין התהליכים: מריין תהליך אחד לפרק זמן קצר. (מילוי-שניות) אז משאה את ביצועו וועבר להריצת תהליכי אחד אחר וכך הלאה.

איך ממשכים את וירטואלציית המעבד?



- מערכת הפעלה ממפה את הכתובת הווירטואלית לכתובת הפיזית.

אתגרים במימוש עיקנון הווירטואלציה.

בעיה. (1) נניח שiosis לוח על כפטור המערכת והדליך המחשב. מערכת הפעלה היא הראשונה שרצה. כלומר, נטענת לזיクリון, מזאה את רכיבי החומרה, מתחילה את בני הנטונים שלה. לאחר שלב האתחול מערכת הפעלה מעבירה את השלים על המעבד הפיזי לתהליכי  $m_1$  כדי שירוץ ישירות על המעבד. אבל מה אם  $m_1$  ימשיך לרווץ ולא יחזיר את השלים למערכת הפעלה. נזכיר שיש  $d_{m_1}$  רגיסטר אחד הוא מצביע לקוד של מערכת הפעלה או של התהליכים.

בעיה. (2) תהליך  $m_1$  עלול לגשת לקבצים שאסור לו לגשת אליהם.

בעיה. (3) תהליך  $m_1$  יבצע פעולה איטית מאוד (כמו קריאה. כתיבה מהdisk) שלא רצה על המעבד?

## מערכות הפעלה

- פערון לבעיה 2. רמות הרשאה. המעבד יגדיר שתי רמות הרשאה.
- *user-mode* : רמת הרשאה נמוכה לשימוש תוכניות רגילות.
  - *kernel-mode* : רמת הרשאה גבוהה לשימוש מערכת הפעלה. בכל רגע נתון יהיה ברמת הרשות אחת ויחידה.
  - רמת ההרשאה הנוכחית – *cpl = current-privilege* ( *cpl = level* ) נשמרת ברגstor כלשהו של המעבד.
  - אם *user-mode* אז *cpl = 0* ואם *kernel-mode* אז *cpl = mode*

פקודות מיוחסות.

- המעבד יחלק את הפקודות לפקודות מיוחסות ולא מיוחסות.
- המעבד יבדוק בלבשב ביצוע הפקודה האם היא מיוחסת. אם כן, אבל המעבד במצב *user-mode* תיווצר חריגה. החriger תעביר את השליטה לגרעון, והוא ירוג את התהליין.

איך נשנה רמת ההרשאה?

- במלחים פשוטות: איך משתמש יג'ז לדיסק כדי לקרוא קובץ.
- ניסיון ראשון: המשתמש יעלה את רמת ההרשאה, וזה יקרא לפונקציה של מערכת הפעלה שתיגש לדיסק. וזאת בעיה כי כל משתמש יהיה לו גישה למערכת הפעלה.

ה策ה טובה יותר: המשתמש יקרא לפקודת מכונה מיוחדת אשר תבצע שתי פעולות בעת ובוונה אחת.

(1) תעלה רמת ההרשאה.

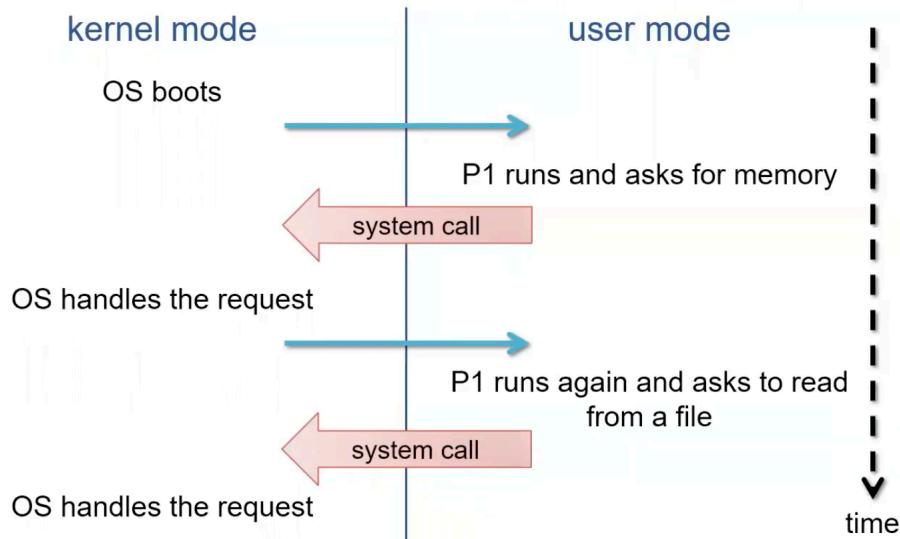
(2) תעבור לבעור פונקציה של מערכת הפעלה.

הפקודה המיוחדת הזאת נקראת *syscall* שהיא מעלה רמה מה הרשותה וגם נתונה ל`ldt` להציג לקוד מערכת הפעלה. זה קורה למשל שימוש רוצה לקרוא מדיסק, להקצות זיכרון, כי הוא לא יודע לעשות את זה אז הוא יעשה *syscall* כזכור קרייה למערכת.

## מערכות הפעלה

10

### תרחיש לדוגמה

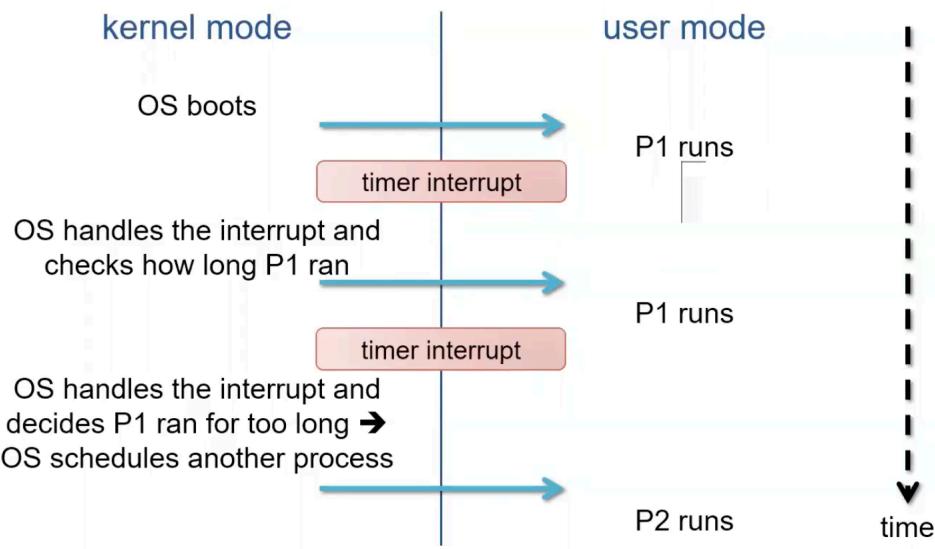


פתרון לבעיה (1) פסיקת שעון.

- לכל תהליך נתן זמן מסויים לרוץ.
- לינוקס מפקיעה את המעבד מהתליכף אחד לטובות תהליך אחר, בעזרת התקן חומרה מיוחד - השעון.
- מערכת הפעלה מבקשת מהשעון לשולח פסיקה במרווחי זמן קבועים כדי להعبر שליתה למערת ההפעלה.
  - כל הפסיקות, בפרט פסיקות שעון, מטופלות ב – *kernel mode*
- במהלך הטיפול בפסקה, מערכת ההפעלה מחליטה האם להליכף את התהליך הנוכחי שרצ כרגע על המעבד.

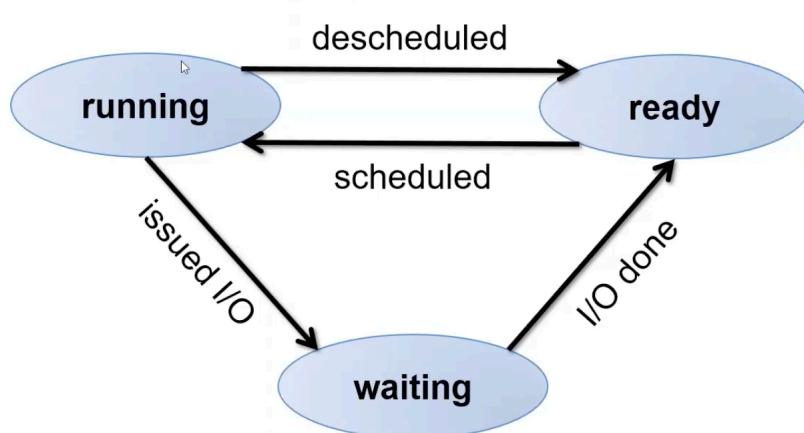
## מערכות הפעלה

### תרכיש לדוגמה



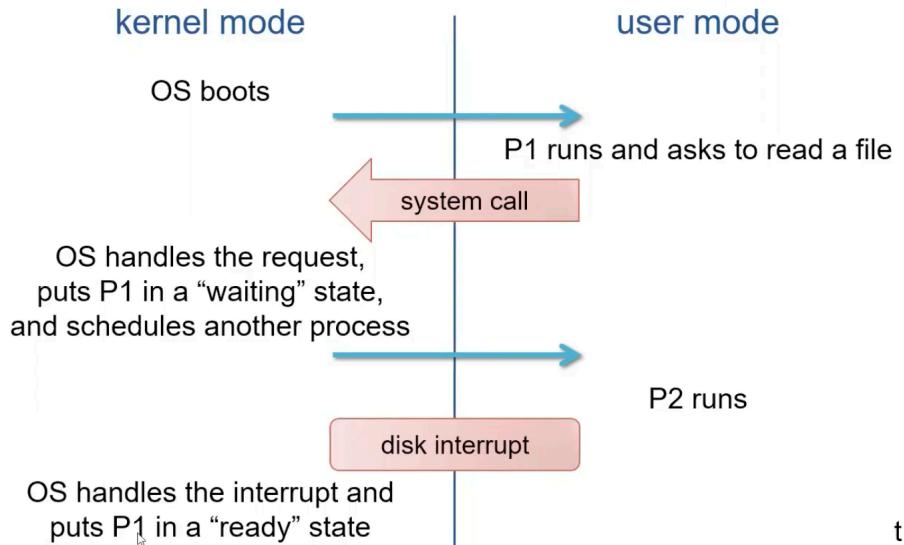
פתרון לבעיה (3)

- תהליך יכול לבקש ממערכת הפעלה שירות  $O/I$  לדוגמא: קריאהقتיבת מדיסק או כרטיס רשת.
- גישה להתקני  $O/I$  איטית מאוד: סדר גודל של מספר מילישניות.
- בזמן ההמתנה להתקני  $O/I$  התהליך לא רץ והמעבד חסר פעולות.
- מערכת הפעלה תסגור את התהליכים במערכת לשולשה מצבים אפשריים.



## מערכות הפעלה

12

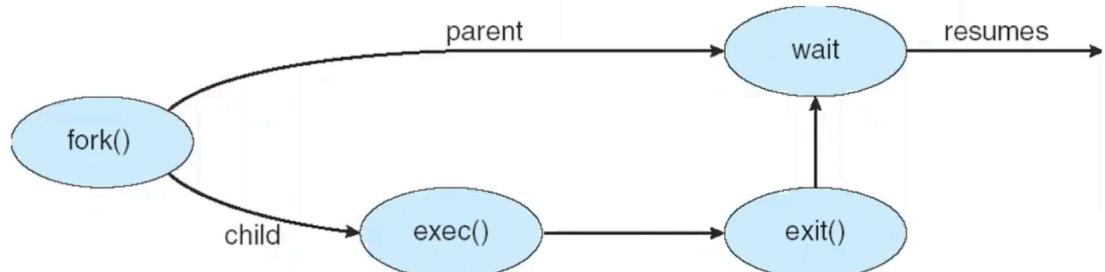


תרגומן 2

*TL; DR*

- תכנית יא אוסף פקודות תהליך הוא ביצוע של אותן פקודות.
- בשעה ראשונה: נלמד איך קוד משתמש יכול ליצור תהליכים חדשים, לבדוק מה מצבם ולהמתין לסיום שלהם.
- הינו מצבים שיש כוריאת מערכת *malloc* הינו מצבים שיש כוריאת מערכת *syscall* שמייצרת תהליך חדש. קודם כל, המשק בלינוקס, אם תהליך רוצה ליצור תהליך חדש הוא משכפל את עצמו דרך *fork*.
- יש לנו אחד שני תהליכים אחד מהם הוא תהליך האבא ואחד מהם תהליך הבן.
- הבעייה היא שאנו רוצים ליצור תהליך חדש ולא תהליך שעושה בדיקות כמו תהליך האבא ואותו קוד. לכן, מה שעושה תהליך הבן זה *exec* שזה טוען תוכנית חדשה לתחליך שנמצא כרגע. ואז שהוא מסתים תהליך האבא ממשיך.

## מערכות הפעלה



מהו תהליך?

- תהליך הוא ביצוע סדרתי של תוכנית.
- מערכת הפעלה נותנת לכל תהליך אשליה שהוא לבד במערכת כדי להקל על פיתוח אפליקציות וכדי לספק לתהליכיים הגנה זה מזה.
- תהליכיין יכולים לתקדר ביניהם.
- מספר תהליכיין רצים בו זמנית על המעבד: מערכת הפעלה מחליפה בין התהליכים במהירות ויוצרת אשליה שהם רצים ביחד.
  - בהמשך נראה איך לינוקס ממשת את החלוקת בין התהליכים.
- כל תהליך צריך משאבים. למשל: זמן מעבד, זיכרון....
- בהמשך נראה איך לינוקס מחלקת את זמן המעבד בין התהליכים
  - למעשה זה אלגוריתם הזמן של לינוקס.

## מערכות הפעלה

14

|  | רשות   | 0%     | 1%       | 56% זיכרון | 4% CPU | PID                                     | שם                       |
|--|--------|--------|----------|------------|--------|-----------------------------------------|--------------------------|
|  |        |        |          |            |        |                                         | אפליקציות (5)            |
|  | 0 Mbps | 0.1 MB | 294.3 MB | 1.3%       |        |                                         | (3) Firefox 🍏 <          |
|  | 0 Mbps | 0 MB   | 75.4 MB  | 0%         | 7096   |                                         | Microsoft PowerPoint 📃 < |
|  | 0 Mbps | 0 MB   | 44.5 MB  | 0%         | 12952  |                                         | Microsoft Word 📄 <       |
|  | 0 Mbps | 0 MB   | 24.5 MB  | 0.2%       | 12872  |                                         | מנהל המשימות 🖥 <         |
|  | 0 Mbps | 0.1 MB | 36.8 MB  | 0.3%       | 13988  |                                         | סיר 📱 <                  |
|  |        |        |          |            |        |                                         | תהליכים ברקע (78)        |
|  | 0 Mbps | 0 MB   | 1.0 MB   | 0%         | 4604   | ...32) Adobe Acrobat Update Service 📁 < |                          |
|  | 0 Mbps | 0.1 MB | 106.8 MB | 0.3%       | 11284  | Antimalware Service Executable 🗑️ <     |                          |
|  | 0 Mbps | 0 MB   | 3.9 MB   | 0%         | 9768   | Application Frame Host 📁 <              |                          |

פחות פריטים ↑

:Windows

תהליכיים בלינוקס:

- לכל תהליך בלינוקס יש מזהה הקורי *id* –
  - מספר שלם בן 32 ביט, ייחודי לתהליכי.
  - ברוב מערכות לינוקס משתמשים רק ב 15 הביטים תחתונים, וכך ניתן ליצור עד  $32K$  תהליכי. מנהל המיצוקות יכול להגדיר מספר גבוה של תהליכי.
  - ערכי ה *pid* ממוחדרים מתחלכים שסימנו לתהליכי חדשים.
- עם עלית המערכת, הגרעין יוצר את התהליך *idle* שמספרו = 0
  - ככל רגע נתון יש תהליך אחד במערכת.
  - נקרא ליריצה כאשר אין תהליכי מוכנים ליריצה ומבצע פקודת *hlt*, המכנית את המעבד למצב שינה.
- התהליך *idle* יוצר תהליך *init* שמספרו 1
  - מפעיל ייזוא את שאר התהליכיים בהמשך.

## מערכות הפעלה

15

קריאה למערכת (*fork()*):

*pid\_t fork();*

- פועלה: מעתיקה את תהליך האב לתהליך הבן ווחזרת בשני תהליכיים.
- קוד זהה (ומייקם בקוד).
- זיכרון זהה (משתנים וערכיהם, גם במחסנית וגם בערימה).
- סביבה זהה (קבצים פתוחים, ספריית עבודה נוכחת).
- אבל, תהייף הבן הוא תהליך נפרד מהתהליך האב, אך יש לו *pid* משלהו.
- פרמטרים: אין.
- ערך מוחזר:
  - במקרה של כישלון: 1 - לאב.
  - במקרה של הצלחה: לבן מוחזר - ולאב מוחזר *pid* של הבן.

לפני (*fork()*)

```
parent

int main() {
 int x = 0;
 pid_t p = fork();
 if (p == 0) {
 x = 1;
 } else {
 x = 2;
 }
}
```

עכשו קראנו ל *fork* ויש לנו שני תהליכים:

## מערכות הפעלה

### אחרי `(fork())`

|                                                                                                                                                    |                                                                                                                                                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>parent</b><br><pre>int main() {     int x = 0;     pid_t p = fork();     if (p == 0) {         x = 1;     } else {         x = 2;     } }</pre> | <b>son</b><br><pre>int main() {     int x = 0;     pid_t p = fork();     if (p == 0) {         x = 1;     } else {         x = 2;     } }</pre> |
|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|

הערה. הרגיטר `rip` מצביע לאותה פקוד אצל האבא והבן כלוומר פקודה אחת אחרי שעשינו `fork` לאבא.

## קוד האב וקוד הבן מסתעפים

|                                                                                                                                                             |                                                                                                                                                          |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>parent</b><br><pre>int main() {     int x = 0;     pid_t p = fork();     if (p == 0) {         x = 1;     } else {         <b>→</b> x = 2;     } }</pre> | <b>son</b><br><pre>int main() {     int x = 0;     pid_t p = fork();     if (p == 0) {         <b>→</b> x = 1;     } else {         x = 2;     } }</pre> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|

מה יהיה ערכו של x  
בסיומו של דבר?  
1 או 2?

## מערכות הפעלה

17

חשיבות להבחן שבסוף ריצת התוכנית יהיה שני עותקים של x ים כלומר לא יהיה רק אחד שהוא ים ערך 2 או 1. הרי לכל תהליך יש מרחב זיכרון משלהו.

- שאלת: מה מופיע הקוד הבא אם `fork` נכשלת? ואם היא מצליחה?

```
int main() {
 fork();
 printf("hello");
 return 0;
}
```

הדפסה לא מתאמת למסך.

אם `fork` נכשלה נדפיס פעם אחת כי הבן לא ידפיס. ואם כן מצליחה, יש הרבה תשובות אפשריות כיוון שמערכת הפעלה מחליפה בין תהליכיים. לכן התשובה לא מוגדרת.

*hellohello* <sub>(1)</sub>  
*hheellollo* <sub>(2)</sub>  
*helhellolo* <sub>(3)</sub>

הסיבה: שני התהליכים ניגשים בצורה לא מתואמת למשאב משותף - המסלול.

קריאה מערכת (`.wait()`)

```
pid_twait(int *wstatus);
```

- פועלה: ממתינה עד אשר אחד מתהליכי הבן יסיים.
- פרמטרים:
  - `wstatus` - מצביע למשתנה בו יאחסנו פרטיים על תהליך הבן שהסתיים.

## מערכות הפעלה

18

- במקרה של *wstatus* יוכל ערך הסיום של הבן) הערך שהעביר כารוגומנט ל *exit* (ערך הסיום מופיע בבית השני מתוך ארבע בתים ה *wstatus*. כדי ללחוץ אותו יד לנצל את המacroו (*\*wstatus >> WEXITSTATUS(\*wstatus)*  
.8) $\square 0xff$
- במידה ולא מעוניינים בסטטוס הבן שהסתיים, אפשר להעביר *NULL*.
- ערך מוחזר:
- אם אין בניים או שכל הבנים כבר סיימו ובוצע *lkm()* יוחזר מיד הערך 1.
- אם יש בניים שסיימו ועודין לא בוצע עבורם *wait* (כלומר הם במצב *zombie* - נראה בשקופית הבאות) - יוחזר מיד ה *pid* של אחד הבנים הנ"ל.
- אחרת, המתנה עד שהבן כלשהו יסתיים.

איך תהוו אבא יכול לחשות לסיום כל תהליכי הבן שלו:

*while(wait(NULL)! = -1);*

איך אז מתקנים הקודם דרך *wait*:

## מערכות הפעלה

```
19 int main() {
 pid_t pid = fork();
 if (pid > 0) {
 // parent waits for child
 wait(NULL);
 }
 printf("hello");
 return 0;
}
```

קריאה למערכת (*.waitpid()*)

*pid\_t waitpid(pid\_t pid, int\*wstatus, intoptions);*

- הפעולה: המתנה למספר בן ספציפי שמספרו *pid*.
- קריאות למערכת *wait*, *waitpid* ו-*clone* חוסמות.
- כלומר, חוסמת את התקדמות התהיליך עד להתרחשות תנאי מסויים.
- באנגלית: *blockingsystemcalls*.
- הארגומנט *options* מאפשר לשנות את התנהנת *options* לקריאת מערכות לא חוסמת.
- אם *options == WNOHANG* תחזיר מיד, כאשר ערך החזרה 0 משמעו שאף תהליך בן עוד לא סיים, ואילו ערך חזרה חיובי הוא ה-*pid* של תהליך בן שישים ונמצא עדין במצב *zombie*.

קריאה למערכת (*.exit()*)

*void exit(intstatus)*

## מערכות הפעלה

20

- פועלה: מסיימת את ביצוע התהילין הקורא ומשחררת את כל המשאים שברשותנו. התהילין עובר למצב *zombie* עד שתהילין האב יבקש לבדוק את סיוומו ואז יפונה לחנותין.

הערה. נשים לב שהיא מסיימת את התהילין עם ערך *status*. היא מסיימת התהילין אך לא ממשידה אותה לגמר אלא משארה אותו במצב ביניים קלומר חי וחייב מות *zombie* והוא ממתים שימושו יעשה לו *wait* שהוא אבא של אותו תהילין.

- פרמטרים:

- ערך סיום המוחזר לאב אם יבודק את סיוומו של התהילין.

- בפועל ניתן להעביר להורה רק 8 ביטים תחתוניים בתור ערך סיום. ולכן קריאות המערכת תעביר (*status* █ 0xffff).

- ערך מוחזר:

- לפי ה *man* קראית המערכת *exit* לא יכולה להיכשל.

למה צריך לסימן תהילין ענ *exit*? אנו רגילים במבוא לעשوت גריד *return 0*.

...ונכון?

• *main()* נקראת ע"י *\_\_libc\_start\_main()* שאוספת את ערך החזרה של *main* וקוראת ל-*exit()*.

```
int __libc_start_main(...){

 exit(main(...));
}
```

ית.

- מה קורה לתהיליך "יתום" *orphan* קלומר תהיליך שסימן לאחר שאביו כבר סיים בלוי לקרוא ל *wait*.
- התהיליך הופך להיות בן של *init*.
- התהיליך *init* ממשיך להתקיים לאורך כל פעולות המערכת.

## מערכות הפעלה

21

- אחד מתפקידיו העיקריים - המתנה לכך בנית כדי לפנות נתוניהם לאחר סיום.

קריאה המערכת *.execv*

```
int execv(const char* filename, char* const argv[]);
```

- פועלה: טוענת תוכנית חדשה לביוץ במקום התהיליך הקורא.

• פרמטרים:

- מסלול אל הקובץ המכיל את התוכנית לטעינה.  
filename -  
- מערך מצביעים למחוזות המכיל את הפרמטרים  
argv -  
argv[0] == filename עבור התוכנית. האיבר הראשון מקיים דהיינו מכיל את שם הקובץ של התוכנית. והאיבר השני הפרט  
האחרון מכיל NULL.

• ערך מוחזר:

- במקרה של כישלון: -1 .

- במקרה של הצלחה: הקירה איננו חוזרת איזור הזיכרון (קוד,  
מחסנית, ..) של התהיליך מאותחלים עבור התוכנית החדשה  
שמהיה להתבצע מההתחלת.

מה ידpix הקוד הבא?

• מה ידפיס הקוד הבא?

```
int main() {
 char *argv[] = {"date", NULL};
 execv("/bin/date", argv);
 printf("hello");
 return 0;
}
```

קריאה המערכת *:getppid, getpid*

- קריית מערכת המזירה לתהיליך הקורא את ה *pid* של עצמו.

## מערכות הפעלה

22

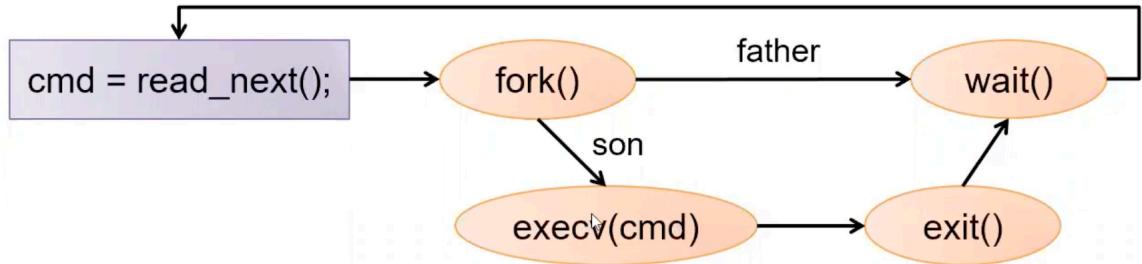
- קריית מערכת המחזירה את *PID* של תהליך האב של התהיליך הקורא.
- שאלה: מה המשמעות של  $getppid() == 1$  עבור תהליך משתמש טיפוסי?
  - תשובה: תהליך האב הוא *init*. קורא למשל אם תהליך הבן יתום.

## דגם לשימוש בתהילכים - *Shell*

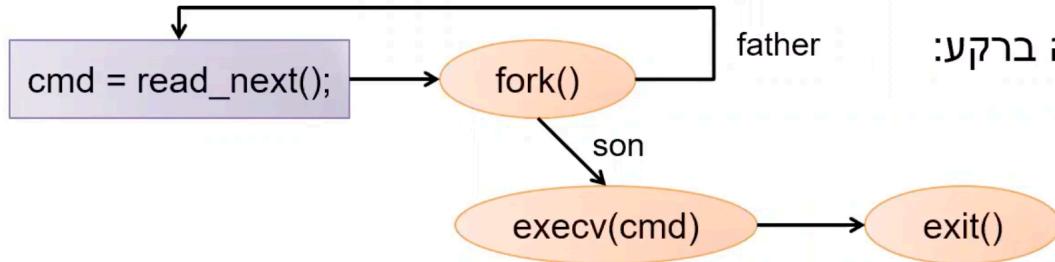
- ממשק שורת פקודה *command line*.
- ייעוד עיקרי: לקבל פקודות ולבצע אותן באופן סדרתי.
  - *shell* מייצר תבליך בן עבור כל פקודה על מנת לבצע אותה.
- כל פקודה ניתנת להריצ' בחזית *foreground* או ברקע *background*.
- הריצה בחזית: האב (*shell*) ממתין לסיום הבן לפני קריית הפקודה הבאה.
- הריצה ברקע: האב (*shell*) עובר מיד לקריאת הפקודה הבאה.
- ייעוד נוסף: להציג קבצים ותיקיות על מנת לסייע במערכת.

## אOPEN פועלות shell בLinux

- הרצה בCHILD:



- הרצה בFather:



מבנה הנתונים לניהול תהליכיים.

- גרעין Linux משתמש את קריית מערכת שראינו ... ... במערכות מבני נתונים הבאים.

- מתאר התהליך *PROCESS DESCRIPTOR*
- במערכת הפעלה אחרות נקרא *PCB* = *PROCESS CONTROL BLOCK*
- שומר בתוכו גם קשרי משפחה.

רשימת התהליכים.

*PID* → *PCB*

- תור התהליכים המוכנים ל裏יצה - *Run Queue*
- תואר המתנה (התוויה הבינוני או רוק) - *Waiting queue*

מתאר התהליכים:

- בכל תהליך בLinux קיים מתאר תהליך (*process descriptor*) שהוא אובייקט מטיפוס *task\_struct* המכיל את:

## מערכות הפעלה

24

- מזהה התהילין
- עדיפות התהילין
- מצביעים למתאר תהליך האב ו "קרוב משפחה" נוספים.
- מצביע לטריות אזרוי הזיכרון של תהליך.
- מצביעי למתאר תהליכיים נוספים.
- מסוף אליו התהיליך מתקשר.
- ועוד ..

מימוש קריית המערכת : *getpid*

- מוגדרת בקובץ *kernel/timer.c*

```
long sys_getpid(void) {
 return current->pid;
}
```

.

ניהול קשרי משפחה ברעין:

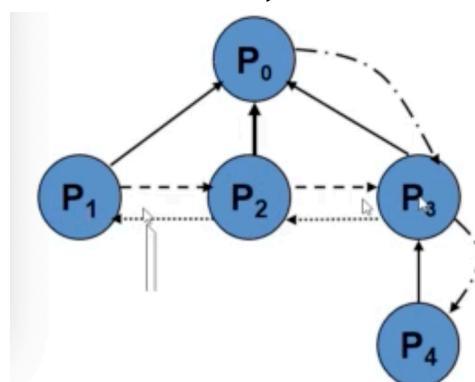
- "קשר משפחה": בין תהליכיים נשמרים באמצעות מצביעים הנשמרים ב-*PCB*.

*real\_parent*: מצביע לאב המקורי.

*parent*: מצביע לאב בפועל.

*Children*: מצביע לרשימה מקוישת של הבנים.

*sibling*: צביעה לרשימה מקוישת של אחיהם (תהליכיים הנוצרים על ידי אותו תהליך אב).



## מערכות הפעלה

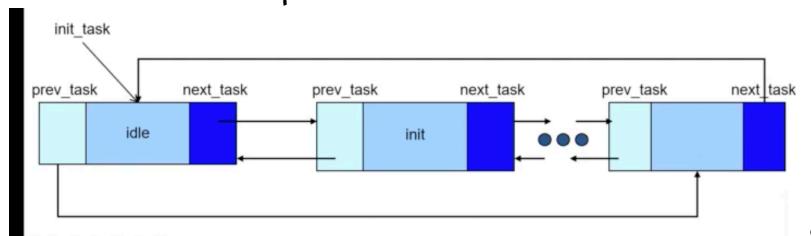
25

הערה: יש בלבול בין *real\_parent*, *parent* שומר מצביע *real\_parent*, *parent* שני מצביעים *real\_parent*, *parent* הבדל הוא האם מדברים בתכנית או לא ואין קשר ל*init*. יש כאן שמלבלים שכאשר תהליך יהיה יותם איז האב שלו *parent* היה *init* והאב בפועל יהיה האב לפני שהוא היה נוכן. מה שנזכר זה שנייהם יהיו *init*. שאנו מרים זיהוי מסוים אז *debugger* יהיה *init*.

אוטו *real\_parent* של *real\_parent*.

רשימת התהליכים:

- מתארו על התהליכים מחוברים ברשימה מקוורת כפולה מעגלית (*proceslist*) באמצעות השדות *next\_task*, *prev\_task*.
- ראש הרשימה הוא המתאר של התהליך *idle* (מוחבע על ידי *init\_task*)
- שאלת: למה הגיוני לעשות רשימה מקוורת כפולה?



איזה קריית מערכת נוספת איבר *PCB* לרשימת התהליכים?

- *() fork*, כי יוצרת תהליך חדש.
- שימושו לרב: *() execv* אינה יוצרת תהליך חדש ולכן אינה נוספת איבר לרשימה.

איזה קריית מערכת מוחקת איבר *PCB* מרשימה תהליכים?

- *() wait* כי היא מוחקת תהליך שהסתיים.
- שימושו לרב: *exit* אומנם מסיימת תהליך אבל אינה מוחקת אותו לغمרי (התהליך עובר למצב *zombie*).

## מערכות הפעלה

26

מצב התהליין:

- מצב התהליין נשמר בשדה *state* במתאר התהליין.
    - משתנה בגודל 32 ביט המפקיד כמערך ביטים: בכל רגע נתון, בדיקת אחד מהביטים ב-*state* דלוק בהתאם למצב התהליין באותו זמן.
  - המצב האפשריים לתהליין בלינוקס הם
    - *Task\_running* - התהליין רץ או מוכן לריצה.
    - נאמר כי תהלייף יוזם לריצה בטוחה קצר.
  - *Task\_zombie* - ריצת התהליין הסתיימה, אך תהליין האב שלו התהליין עדין לא ביקש מידע על סיום התהליין באמצעות קריאה *condemand* של *wait*.
  - מתאר התהליין הוא הדבר היחיד שנשאר ממנו.
- *TASK\_INTERRUPTIBLE* - המתנה רדומה.
- התהליין ממتنן לאיורע כלשהו אך ניתן להפסיק את המתנה התהלייף ולחזרו למצב *TASK\_RUNNING* באמצעות שליחת סיגナル כלשהו לתהליין.
  - זהו מצב המתנה הנפוץ. متى נמצא תהלייכים במצב זה?
- \* דוגמא: 1. תהליין אב הממתין לסיום הבן (קריית *.wait*)
- \* דוגמא: 2. דפדפן *web\_browser* מחקה לקבלה נתונים מהרשת (דף *web*) אבל אפשר לקטוע המתנתו על ידי סגירת חלון היישום, שגורמת לשילוח אותן לסיום התהליין.
- *TASK\_UNINTERRUPTIBLE* - המתנה עמוקה:
- התהליין ממتنן לאיורע כלשהט אך לא ניתן להפסיק את המתנה התהליין באמצעות שליחת סיגナル לתהליין.
  - מצב המתנה נדיר.
- *Task\_stopped* - ריצת התהליין נעצרה בצורה מבוקרת על-ידי תהליין אחר (בדרכן כל *Debugger* או *Tracer*).

## מערכות הפעלה

27

- התהליכים המוכנים ל线索 (מצג *task\_running*)  
 נשמרים במבנה התונים הקרווי *runqueues* (תור הריצה).
- לכל מעבד יש מבנה *runqueues* משלהו:  
*struct rq runqueues[NR\_CPUS];* •

תרגול 3

תזכורת. בתרגול ראשון קודמים דיברנו על תפקיד מערכת הפעלה יש לנו אפליקציות משתמש שרצות ויש את רכיבי חומרה מעבד, דיסק וחיכון ומערכת הפעלה היא מתווכת ביניהם ועם עיקנון הווירטואלציה שמערכת הפעלה נותנת לכל תהליך שהוא התהליך היחיד שרצ במערכת ועושה את זה על ידי חלוקת המעבד בזמן כל תהליך לוקח זמן מהמעבד וכיון שאנו איטים נראה לנו שככל תהליך רץ ברציף. כל תהליך מקבל זיכרון ובגלל הזיכרון הווירטואלי כל תהליך מרגיש שיש לו את כל הזיכון לעצמו. תרגול אחרון דיברנו על קריאות מערכת, למשל *execve*, *fork*, *wait*. ולמדנו איך תהליכי נוצרים על ידי שכפול עצם וזה הבן שנוצר יוצר לעצמו זיכרון חדש על ידי שינוי.

תהליכי בלינוקס: אמרנו שלכל תהליך חשוב שהוא בלבד במערכת כל אחד יש לו רגיסטרים שלו מרחב זיכרון שלו ולא מודיע שיש עוד תהליכי אחרים רצים במקביל. אמרנו אבל שיתר נוח שתהליכיים לתקשר ביניהם למשל פיסבוק, כל יותר לפצל את כל האפליקציה ולא כתבו אותה בתוכנית אחת *c*, *cpp* גדולה. אלא משתמשים בהשה שנקרא — *microservices* ביחיד ומתקשרים ביניהם ומחליפים מידע. איך עושים את זה בלינוקס? למשל, יש לי תהליך אחד ורוצה להעביר מחזרות לתהליך שני מה יעשה?

(א) אפשר דרך קבצים (מערכת קבצים). למשל בתמונה ה затה רואים שיש *more*, *less*, *ano* יודעים שלא מדפיס כל הקבצים שיש בתיקיה הנוכחיות. למשל אם נרצה לחבר ביניהם אנו נעשו הבא. נקח *less* ואת פלט שלו נכתב לקובץ זמני דרך אופרטור > (של באש) ויש

## מערכות הפעלה

28

גם אופטור  $<$  שלוקח קובץ וושאפֵך את התוכן שלו לתוך הקולט *more*.

(2) אפשר לחבר שני תהליכי באותה שורה באמצעות | שבעצם מפנה מיד של מה *that* כותב דוarily  $\backslash more$  במקום  $\& more$ . *more* במקומם למסן. *more* במקומות שיקבל במקלדת הוא יקבל מ *ls*.

| pipes                           | מערכת הקבצים                                                               |
|---------------------------------|----------------------------------------------------------------------------|
| <code>&gt;&gt; ls   more</code> | <code>&gt;&gt; ls &gt; temp</code><br><code>&gt;&gt; more &lt; temp</code> |

(3)

איזה שיטה היא עדיפה?

מבחינת מהירות *pipe* יותר מהירים הם לא עוברים דרך ה-*disk*, לעומת שיטה בעד ימין שצריך לכתוב קובץ לדיסק ולקראת קובץ מדיסק והוא איטי בסדרי גודל מזיכרונו. לכן עדיף להשתמש ב *pipe* ולא מערכת קבצים. אנו נראה איך *pipe* ממומש. גם יתרון מבחינת מקום, צריך למחוק קובץ זמני ולהקצות. אבל אם נרצה גיבוי אז קובץ *temp* יעוז לנו אבל השם שלו מرمז אז נראהה תקשורת הרגילה אנחנו לא נצטרך אותו לאחר מכן נראה עדיף ב *pipe*.

- מבחינת מערכת הפעלה, המימוש הוא באמצעות קבצים. אנו רגילים לחשב על קבצים כמו מיידע ישוב על דיסק אבל בلينוקס יש קונספט שונה זה – *every – thing – is – a – file*. לכן מעכשו, מה שנקרה לו קבצים זה לא רק קבצים רגילים שיושבים על דיסק אלא גם חלק קלטפלט למשל עבר, מסן יהיו קובץ. גם ערוצי תקשורת ייעודים כמו *pipes*, *sockets* (*socketests*) מקשרים בין מחשבים לאינטרנט שהם ממושים באמצעות קבצים יהיו קובץ.
- נתחיל באמצעות פרימיטיבים שלא ממושים דרך קבצים, ובגלל שהוא כל כך מינימליsti הוא ממומש בפרד משאר אמצעי תקשורת זהה *signals* – אותו מספרים שלמים בין 1 – 31.

## מערכות הפעלה

29

- סיגנל יכול לעבור מתהליך  $A$  למתהילך  $B$  וגם יכול לעבור בין מערכות הפעלה למתהילך.
- מתהילך מבקש שירותים מממשק הפעלה זה נקרא *syscall*.
- סיגנלים ממומשים בתוכנה בלבד בלי קשר לחומר כלומר לא קשורים למעבד ספציפי שרצים עלייו, והם נשלחים באופן אסנכרוני ויכולים להגיע בכל נקודה בזמן. דהיינו, מתהילך  $A$  שולח סיגナル  $S$  למתהילך  $B$  אין בהם קשר לפניה שליחת סיגナル כלומר סיגナル יכול לעבור מ $A$  ל $B$  בכל רגע בקוד של המתהילך  $B$ . כלומר המתהילך  $B$  מרים פקודה אחרי פקודה ופותאום מקבל סיגナル שגורם לבצע קוד אחר.
- סיגנלים ≠ פסיקות. הבלבול (*Interrupt*) גורם למתהילך לקטוע את ביצועו שלו ולבצע שגרת טיפול שהיא ברמת מערכת הפעלה וסיגナル הוא במצב משתמש אליו גורם למתהילך לקטוע את ביצועו ולבצע קוד אחר שגם הוא במרחב המשתמש.

|                 |    |
|-----------------|----|
| #define SIGHUP  | 1  |
| #define SIGINT  | 2  |
| #define SIGQUIT | 3  |
| #define SIGILL  | 4  |
| #define SIGTRAP | 5  |
| #define SIGABRT | 6  |
| #define SIGIOT  | 6  |
| #define SIGBUS  | 7  |
| #define SIGFPE  | 8  |
| #define SIGKILL | 9  |
| #define SIGUSR1 | 10 |
| #define SIGSEGV | 11 |
| #define SIGUSR2 | 12 |
| #define SIGPIPE | 13 |
| #define SIGALRM | 14 |
| #define SIGTERM | 15 |
| ...             |    |

בלינוקס יש 31 סיגנלים,  
לכל אחד שם ומספר  
שלם בין 1—31

Idan Yaniv

המשתמש לחץ על CTRL+C –  
מתהילך shell ישלח SIGINT למתהילך  
שרץ בחזיות.

מתהילך ביצע פקודה לא חוקית – מערכת  
הפעלה תשליך למתהילך SIGKILL.

מתהילך ניגש לכתובת לא חוקית בזיכרון –  
מערכת הפעלה תשליך למתהילך SIGSEGV.

סיגנלים בלינוקס:

## מערכות הפעלה

30

למשל *SIGSIGV* ברגע שניגשים נכתובה לא חוקית בזיכרון אז יצר חריגה אבל בסוף הלירה מערכת הפעלה החליטה לא להרוג את התהיליך אלא להרוג את התהיליך.

קריאה המערכת : *kill*

*int kill (pid\_t pid, int sig)*

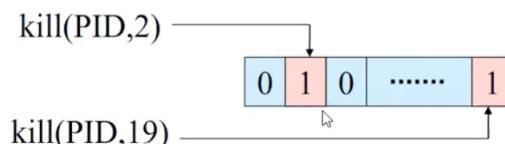
היא לא הורגת תוכנית היא פשוט שולחת סיגナル זה חשוב מאוד כי השם מבלב. וסיגנלים שונים מטופלים באופן שונה.

- שולחת סיגナル שמספרו *sig* לתהיליך המזוהה על ידי *pid*.
- אם  $0 == sig$  הפעלה רק בודקת שהטיליך *pid* קיים מבלי לשלוח *sig* (שימושי לבדיקת תקיפות *pid*).
- ערך מוחזר: 0 בהצלחה,
- 1- בכישלון (למשל אם אין תהיליך מזוהה על ידי *pid*).

העברה סיגナル בשני שלבים:

(1) רישום: מערכת הפעלה רושמת ב *PCB* של תהיליך היעד שיש לו סיגナル ממתיין (*pending signal*).

(2) הרישום מתבצע במערך בינארי בין 31 ביטים, וכך לכל היותר סיגナル אחד ממתיין בכל מספר.



(3) טיפול: בכל עם שהטהיליך חוזר מ מצב גרעין לertz משמש, מערכת הפעלה בודקת אם יש סיגנלים ממתיינים ומטפלת בהם.

(4) בסיום הטיפול מערכת הפעלה תאפס את הבית המתאים במערך.

(5) במידה ויש כמה סדר הטיפול מתחילה המערכת לסייעו.

## מערכות הפעלה

הטיפול בסיגנליים נראה שיכול לדוחות מממש רחוק וזה לא נכון, אלא קורה באופן דחוף כי גרעין חוזר הרבה מנצח גרעין למשתמש וזה קורה ברגע פסיקות שעון. וזה ממש לא הרבה זמן.

טיפול בסיגנלים: תהיליך יכול לטפל בסיגnal במספר אופנים, לדוגמה:

- סיום התהילה בתגובה לסיגן - *terminate*.

- התעלמות מהסיגל והמשר הביצוע הרגיל. - *ignore*

- **TASK\_STOPPED** - עצירת התהליין במעב stop •  
בשלית debugger).

- TASK TOPPED - המשך ביצוע תהליך שהיה במצב *continue*.  
בד"כ בשליטת debugger).

- "תפיסת הסיגנל - Catching signals - הפעלת שגרת משתמש מיוחדת (signal handler) בתגובה לסיגנל.

שגרת הטיפול בסיגナルים: תהלייך יכול להתקין שגרת טיפול בסיגナル (*signal handler*) בשיטת קבלת סיגナル מסויים. השגרה מבוצעת ב-*user mode*, שתיקרא בעת קבלת סיגナル מסויים. ניתן להתקין שגרת טיפול בהקשר של התהלייך שקיבל את הסיגナル. חדש לתקין שגרת טיפול חדש לכל סיגナル פרט ל-*SIGSTOP* ו-*SIGKILL*.

*used*. המשתמש מגדר את שגרת הטיפול והיא מטופלת ב-*user mode*. כזכור מוגדרת בקוד המשתמש.

המבנה *PCB*: ב-*signal\_struct* שומר מבנה בן 31 תאים ובודק שמרוות פעולות הטיפול בכל סיגנל. אופציותות לטיפול:

- בצע את טיפול ברירת המחדל בסיגנל זה. (מה SIG\_DFL •

שמערכת הפעלה מציעה).

- - SIG\_LGN הטעם מהסיגר.

- קישור ל-*signalhandler* שהוגדר ע"י המשתמש. (מצבע ייעוד).  
לפונקציה של המשתמש לטפל בסיגנל.

## מערכות הפעלה

32

קריאה המערכת *signal* : פעולה: משנה את אופן הטיפול בסיגナル שמספרו *signum*.

- פרמטרים: *signum* - מספר בתחום 1-31 פרט 6-*SIGKILL(9)* ו-*SIGSTOP(17)*.

- או *SIG\_DFL* - מצביע לפונקציית משתמש או *handler* • *SIG\_LGN*

- ערך מוחזר:

- בהצלחה, ערכו הקודם של ה-*signalhandler* (פונקציה

*./SIG\_DFL/SIG\_IGN*) קודמת

- בכישלון, *SIG\_ERR*

### דוגמה

```
>> gcc signal.c
>> a.out &
[1] 3189
Waiting...
>> kill 3189
Hi
Bye
[1]+ Done a.out
>>
```

```
#include <stdio.h>
#include <signal.h>

void catcher1(int signum) {
 printf("Hi\n");
 kill(getpid(), 22);
}

void catch22(int signum) {
 printf("Bye\n");
 exit(0);
}

main() {
 signal(SIGTERM, catcher1);
 signal(22, catch22);
 printf("Waiting...\n");
 while(1);
}
```

דוגמא:

סיכום: סיגנלים נשלחים בשני שלבים: רישום, טיפול.

מערכות הפעלה לתחלין:

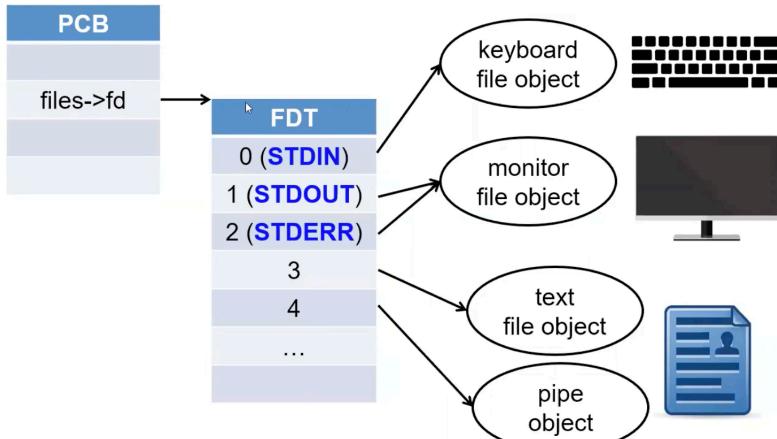
- תחelin ☐ יצר חריגה הדורשת את התערבות מערכת הפעלה.
- מערכת הפעלה מודיעה לתחelin ☐ על האירוע ע"י רישום סיגナル PCB-ב-*ל-לן*.
- במעבר ממצב גרעין לקוד משתמש של תחelin , ☐ יטופל הסיגנל.

תחelin לתחelin:

## מערכות הפעלה

33

- תהלין פונה למערכת הפעלה שתרשום סיגנל לתהליין.
- מערכת הפעלה רושמת סיגנל ב-*PCB* של תהלין.
- במעבר ממצב גרעין לקוד משתמש של תהלין, יטופל בסיגナル.



קלט/פלט של תהליכיים:

למה נוח שהכל כמו קבצים:

כי להדפיס לקובץ או מסר נראה אותו דבר, כי יש *write* ו מבחינה משתמש כתיבה למסר או קובץ זה אותו דבר בקריאה מערכת. אם נחשוב שלכל אחד יהיה קיראת מערכת מיוחדת למשל לעכבר, מסן, מקלדת אז יותר מסבך.

### *FD (File descriptors)*

- כל פעולות קלט/פלט של תהליך בlynoks מבוצעות דרך "קבצים":
- קבצים "רגילים" לאחסון מידע (`/usr/file.txt`) נמצאים בדיסק.
  - התקני חומרה גם כן מייצגים קבצים, אבל נמצאים בזיכרון.
- למשל, העכברים המחברים למחשב מיוצגים כ-`/dev/input/mouseN` -

- גם ערוצי תקשורת כמו *pipes* מיוצגים ע"י קבצים שנמצאים בזיכרון.

הקשר בין תהלין לבין קובץ שהוא ניתן נשמר, ברמת המשתמש, ע"י מספר שלם שנקרא (*filedescriptor*) (*FD*). לדוגמה: קראית המערכת (*open()*) מחייבת *FD* כניסה חדשה בטבלה

## מערכות הפעלה

34

שמצבייה לקובץ. המשמש מעביר את ה-*F*-*D* (שזה הכניסה החדשה בטבלה) לקריאה מערכת כמו *read()*, *write()* כדי לקרוא ולכתוב לקובץ.

### *FDT* (*File descriptor table*)

- ברמת הגרעין, *F*-*D* הוא אינדקס לכניסה בטבלה הנקראת (*FDT*).  
.filedescriptorable(*FDT*)  
- כל תהליך יש *FDT* משלהו, המוצבעת ע"י השדה →  
        *PCB* ב-*fd*  
• כל כניסה ב-*FDT* מצביעה על אובייקט ניהול של קובץ פתוח  
        (*file object*).  
- אובייקט ניהול נגיש ומותזק ע"י גרעין מערכת הפעלה  
         בלבד.  
- מכיל מספר שדות \*  
        \* למשל את "מחוון הקובץ" (seekpointer) המצביע  
        למקום הנוכחי בקובץ כלומר, מהין לקרוא/לכתוב  
        את הנתונים הבאים.  
• כל הקבצים הפתוחים של כל התהליכים במערכת נשמרים גם הם  
    בטבלה גלובלית המנוהלת ע"י הגרעין - ה-*G*-*FDT*.  
        .*GFDT*

### ערוצי קלט פלט סטנדרטיים

ערכי ה-*F*-*D* הבאים מוקשרים להתקנים הבאים כברירת מחדל:

- 0 - ערוץ הקלט הסטנדרטי (*STDIN*), בדרכן-כללי מקשר למקלדת.
- פועלות הקלט המוכנות, כדוגמת *scanf()* ודומותיה, קוראות  
    למעשה מהתקן הקלט הסטנדרטי
- 1 - ערוץ הפלט הסטנדרטי (*STDOUT*), בדרכן-כללי מקשר  
    למסך. פועלות הפלט המוכנות, כדוגמת *printf()* ודומותיה,  
    כותבות למעשה מהתקן הפלט הסטנדרטי

## מערכות הפעלה

35

- 2 - עורך השגיאות הסטנדרטי (*STDERR*), בדרכו-כלו גם הוא מקשר למסך.

## דוגמת קוד

```
>> cat main.c
#include <stdio.h>

int main() {
 printf("Hello World!\n");
 FILE* f = fopen("file.txt", "w");
 fprintf(f, "Hello World!\n");
 return 0;
}

>> gcc main.c
>> strace ./a.out
...
write(1, "Hello World!\n", 13) = 13
open("file.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
write(3, "Hello World!\n", 13) = 13
...
```

מאתורי פונקציות libc אלו  
מסתתרות קראות המערכת  
open(), read(), write()

strace הוא כלי המאפשר לעקוב אחרי  
קראות מערכת במהלך התוכנית

1

הסביר: איזה קראת מערכת מסתתרת מאחוריו הקוד? ראשית אנו מדפיסים למסך ואחרי זה פותחים קובץ שנקרא *file.txt* לכתיבה. ובבבלים *fopen*\* או *FILE\** וואז מדפיסים לתוך הקובץ *HELLO..*. מתחת, מסתתרות הרבה קראות מערכת למשל מתחת *print*, *printf*, *syswrite* וגם *open* מתחת *fopen*. ותחת *fopen* מסתתר *open* ותחת *printf* מסתתר *write*. ניתן לראות זה על ידי *strace*. למשל ב *write* הראשונה מתאימה *printf* איך יודיעם? כי רואים שארגוןعط הרាលון זה 1 והשלישי שזה 13 מייצג את מספר הבתים. השלישית מתאימה ל *fprintf* איך

## מערכות הפעלה

36

יודעים? כי רואים שארגוומינט הראשון זה 3 והשלישי שהוא 13 מייצג את מספר הבטים.

### פתיחה קובץ לגישה



```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *path, int flags);
int open(const char *path, int flags,
 mode_t mode);
```

פתיחה קובץ לגישה.

- פועלה: פותחת את הקובץ המבוקש (לפי *path*) לגישה לפי התכונות המוגדרות ב-*flags* ולי הרשות המוגדרות ב-*mode*.
- ערך מוחזר: במקרה של הצלחה - ה*FD* הקשור לקובץ שנפתח.
- האינדקס המוקצה בטבלה הוא האינדקס הפנוי הנמוך ביותר ב-*FDT*. במקרה של כישלון 1.

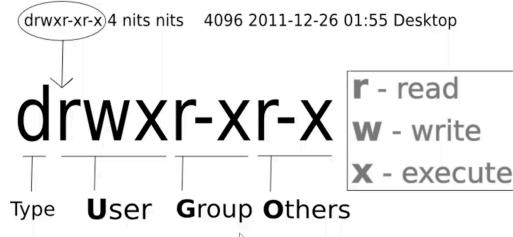
פרמטרים:

- מסלול לקובץ (או התקן) לפתיחת *path*. לדוגמה:
  - "file1" - לציין הקובץ *file1* בספרית העבודה הנוכחית.
  - "/usr/hw/file1" - לציין קובץ כתובות אבסולוטית.
- *flags* - תכונות לאפיקן פתיחת הקובץ. חיב להכיל אחת מהאפשרויות הבאות:
  - *O\_RDONLY* - הקובץ נפתח לקריאה בלבד.
  - *O\_WRONLY* - הקובץ נפתח לכתיבה בלבד.
  - *O\_RDWR* - הקובץ נפתח לקריאה ולכתיבה.
- ניתן להוסיף תכונות אופציונליות באמצעות ()() עם הדג' המתאים, למשל:
  - *O\_CREAT* - צור את הקובץ אם אין קיים.
  - *O\_APPEND* - שרשר מידע בסוף קובץ קיים.

## מערכות הפעלה

37

- *mode* - פרמטר אופציוני המגדיר את הרשותות הקובץ, במקרה שפתיחה הקובץ גורמת ליצירת קובץ חדש (למשל, כאשר מכילים *O\_CREAT*).



הרשאות קבצים בLinux:

```
#include <unistd.h>
```

```
int close(int fd); close: קריית מערכת קובץ
```

- פועלה: סורגת את הקובץ המוצעה על ידי *fd*. לאחר הסגירה, לא ניתן לגשת דרך *fd*.
- פרטטים:

- *FD - fd* המיועד לסגירה.

- ערך מוחזר: במקרה של הצלחה - 0. במקרה של כישלון - 1.

CLOSE לא הביאה מוחקת את הקובץ עי' יתכן שכמה תהליכי מצביים עלי ובס *file - object* יש שדה שנקרא *fcnt* שהוא מונה מספר המצביים על אותו *fileobject* הקובץ נמחק אם ורק אם  $0 == fcount$  כי אז אף אחד לא מצביע עליו.

קריית נתונים מקובץ.

```
include <unistd.h >
```

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

## מערכות הפעלה

38

- פעולה: מנסה לקרוא עד *count* בתים מתוך הקובץ הקשור *fd*-*buf* לתוכן החוצץ.
- מחוון הקובץ (*seekpointer*) (מקודם בכמות הבטים שנקראו, כך שבפעולת הגישה הבאה לקובץ (קריאה, כתיבה ועוד) ניגש לנתונים שאחרי הנתונים שנקראו בפעולת הנוכחות).
- פעולה הקריאה עשויה לחסום את התהיליך (כלומר, להוציאו אותו מהמתנה) עד שהיו נתונים זמינים לקריאה, למשל עד שיגיעו נתונים מהdisk.
- פרמטרים:
  - *DF-fd* - המקשר לקובץ ממנו מבקשים לקרוא.
  - *buf* - מצביע לחוצץ בו יאוחסנו הנתונים שייקראו.
  - *count* - מספר הבטים המבוקש.
- ערך מוחזר:
  - במקרה של הצלחה - מספר הבטים שנקראו בפועל מהקובץ *buf*.
  - יתכן שייקראו פחות מ-*count* בתים, למשל אם נותרו פחות מ-*count* בתים בקובץ ממנו קוראים.
  - יתכן גם שלא יקראו בתים כלל, למשל אם מחוון הקובץ הגיע לסוף הקובץ (*EOF*). אם *read()* נקראת עם *count = 0*, יוחזר 0 ללא קריאה.
  - במקרה של כישלון (-1).

כתבת נתונים לקובץ:

*ssize\_t write(int fd, const void\*buf, size\_t count);*

- פעולה: מנסה לכתוב עד *count* בתים מתוך החוצץ *buf* לקובץ הקשור *fd*-*buf*.
- בדומה ל-*read()*, מחוון הקובץ מקודם בכמות הבטים שנכתבו בפועל, והגישה הבאה לקובץ תהיה לנתונים שאחרי אלו שנכתבו.

## מערכות הפעלה

39

- גם פעלת `write()` יכולה לחסום את התהילין (כלומר, להוציאו אותו מהמתנה), למשל עד שתתאפשר גישה לדיסק.

.File – objects

- קריית המערכת `Open()` מוסיפה כניסה חדשה חדשה במקום הפנו הראשון בטבלת `FDT` של התהילין.

- הכניסה החדשה מציבה על אובייקט מטיפוס `.struct_file`

```
struct file {
 atomic_t f_count;
 ...
 loff_t f_pos;
 mode_t f_mode;
 ...
 struct file_operations*
 f_op;
};
```

- `f_count` – סופר את מספר ההצלבות לאותו אובייקט.

- למשל: תהיליך אב ובן יציבו לאותו אובייקט לאחר `fork()`.

- למשל: אותו תהיליך יכול להציג פעמים לאותו אובייקט בעקבות `dup()`. המונה משמש לשחרור האובייקט ב-`close()` מהטהילין האחרון המצביע.

- `f_pos` – ה-`seek – pointer` – מצביע למקום הקריאה או הכתיבה הנוכחי.

- `f_mode` – שומר את הרשות הקובץ, למשל אם הקובץ ניתן לקרוא/כתוב.

- מערכת הפעלה תבודוק את שדה זה לפני ביצוע הפעולות `write, read`.

- `file_operations` – מבנה המכיל מצביעים למימוש של הפעולות.

`open, read, write, lseek`

שוב בתרגול על מודולים ודריברים.

שילוב קלט/פלט בין חוטים.

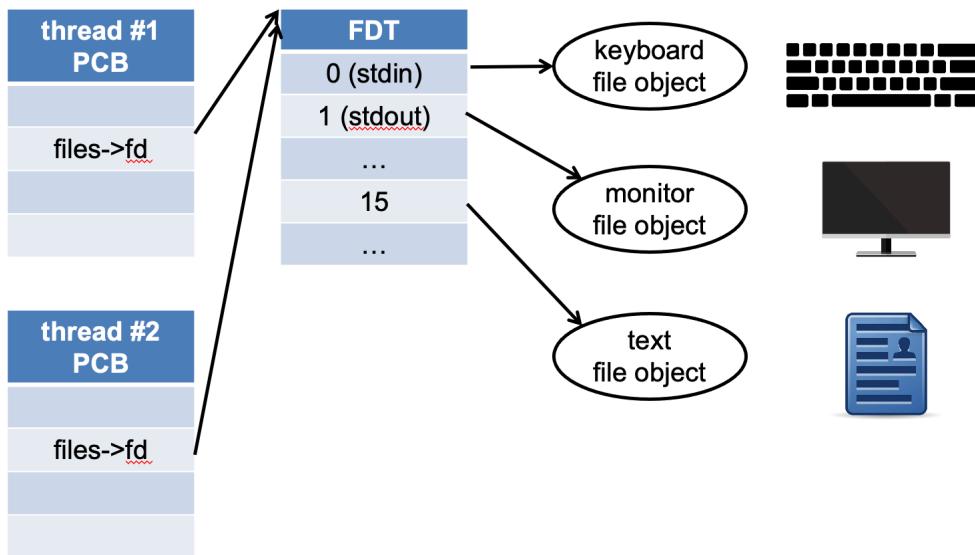
- חוטים של אותו תהיליך משתפים ביניהם את ה-`FDT`.

## מערכות הפעלה

40

- מתארី התהיליכים של כל החוטים מצבעים על אותו *FDT*.
- אם חוט אחד פותח קובץ גם החוט השני יוכל לגשת לקובץ זה.
- אם חוט אחד סגור קובץ אז גם החוט השני לא יוכל לגשת אליו יותר.

חותמים (ותחיליכים) המשתמשים ב-*FD* משותפים צריכים להתאם את פעולות הגישה לקבצים על-מנת שלא לשבש זה את פעולות זה. לינוקס מזינה מגוון אפשרויות לנעילה של קבצים - מעבר לחומר הקורס.



שיתוף קלט/פלט בין תחיליכים.

- קריית המערכת `fork()` מעתיקה את ה-*FDT* לתהlixir הבן.
- כל שינוי ב-*FDT* אצל האב/הבן לאחר `fork()` נראה אצל השני.
- לדוגמה: אם תהlixir הבן פותח קובץ חדש הוא לא נפתח אצל האבא.
- אבל ה-*fileobjects* (קבצים הפתוחים) זהים אצל האב ואצל הבן.
- שדות *fileobject* בכמו מחוון הקובץ, יהיו זהים.

## מערכות הפעלה

41

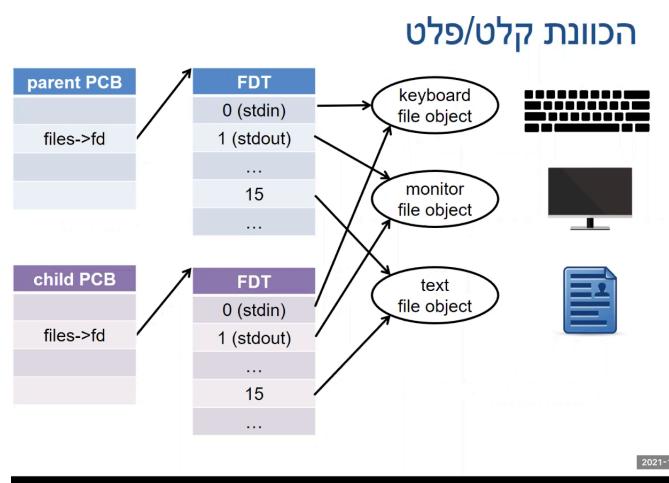
- לדוגמה: אם האב קורא 10 בתים מהקובץ ואחריו הבן קורא 3 בתים, אז הבן יקרא את 3 הבטים שאחרי ה-10 של האב.

שימוש ל-*ב*: כל פתיחה של קובץ מייצרת *fileobject* חדש. לדוגמה: אם אותו תהליך פותח פעמיים את אותו קובץ, אז הגרעין ישמור שני – *file objects* שונים המצביעים לאזוריים שונים באותו הקובץ.

שחרור *file-object*.

מי מבצע שחרור הזיכרון של ה *file-object*? מתי  
ניתן לשחרר?

- ייתכנו מצבים בהם תהליכיים שונים מצביעים לאותו *fileobject*, אך שחרור ה-*fileobject* יכול להתבצע רק לאחר ביצוע (*close*) מכל התהליכים החולקים אותו ה-*fileobject*. נזכר – ל-*fileobject* יש מונה (*f\_count*) הסופר את כמות התהליכים המצביעים עליו בכל רגע נתון. המונה קטן באחת עם כל פעולה *file-object close* על האובייקט. כאשר המונה מתאפס, ה-*file-object* שוחרר.

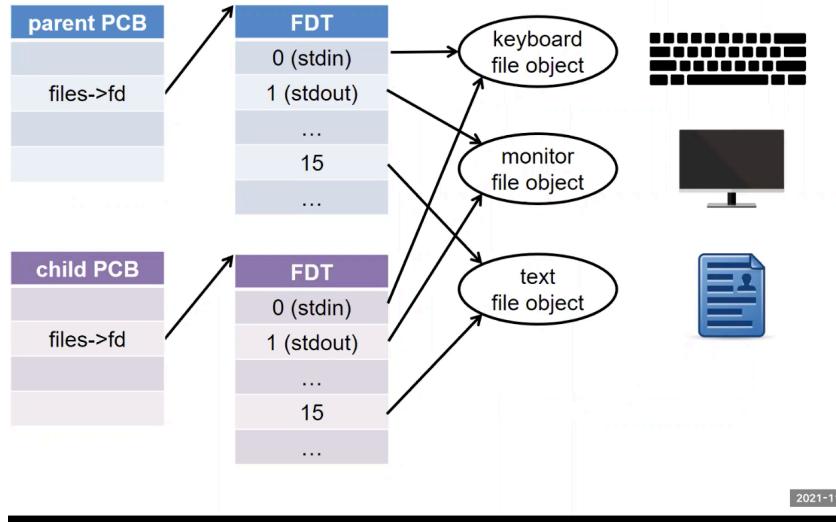


ניתן לראות ש *FDT* של תהליך האבא והבן מצביעים על אותו *File-object* ב מונחה | למעלה. כתם נעשה (*close()*) גערוץ הסטנדרטי של הבן מה יקרה?

## מערכות הפעלה

42

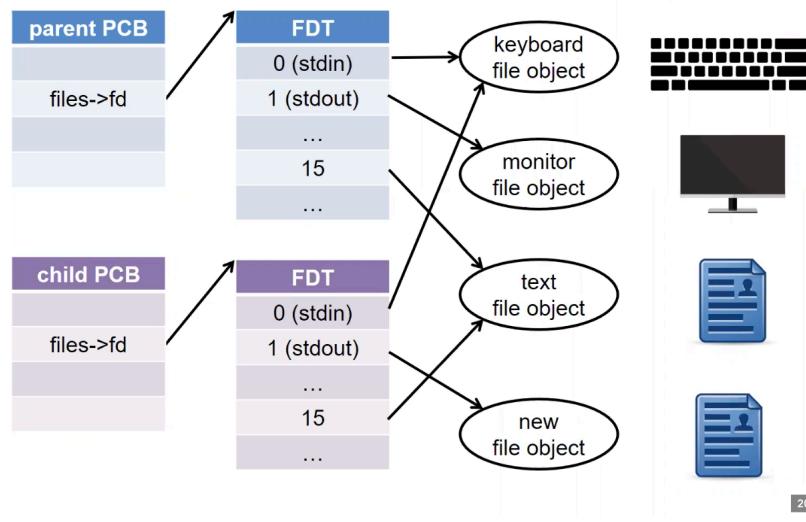
### הכוונת קלט/פלט



2021-11

כעת נעשה *new file - object* ו-*open* רצ' מה שקרה הכניסה  
הראשונה הפניה ב FDT של הבן היא זאת שתביע עליון דהינו,

### הכוונת קלט/פלט



2021-

אפשר לראות מה שעשינו למעלה בקוד הבא:

## מערכות הפעלה

43

shell code:

```
pid_t pid = fork();
if (pid == 0) {
 close(1);
 open("file.txt",
 O_CREAT ... , ...);
 char* args[] =
 {"date", NULL};
 execv(args[0],
 args);
} else {
 wait(NULL);
}
```

פעולת `execv()` ודומותיה אין מושנות את ה-*FDT*-ה של התהליין, למרות שהטהליין מאוחל מחדש. כמובן, קבצים פתוחים אינם נסגרים. התוכנה הזאת שימושית להכוונת קלט/פלט של תכניות (*input/output redirection*). למשל, ניתן לכתוב את התאריך והשעה הנוכחיים לקובץ במקום לפלט הסטנדרטי באמצעות: `date > file.txt` שזה בקוד מתבצע על ידי `execv`.

*CheckPoint* - מה ראיינו עד עכשיו.

- דיברנו על דרך לספר לטהליין על אירוע.
  - בעזרת סיג널ים (*signals*).
  - דיברנו כיצד מתבצע קלט/פלט בלינוקס.
  - בעזרת קבצים (*files*).
- "Every thing is a file"* \*
- ועכשיו - נדבר על "קבצים" מיוחדים לתקשורת בין תהלייכים:
    - תקשורת בין תהלייכים עם קשר משפחתי. *Pipe* -
    - תקשורת בין תהלייכים כלשהם. *FIFO* -

*pipes* בלינוקס.

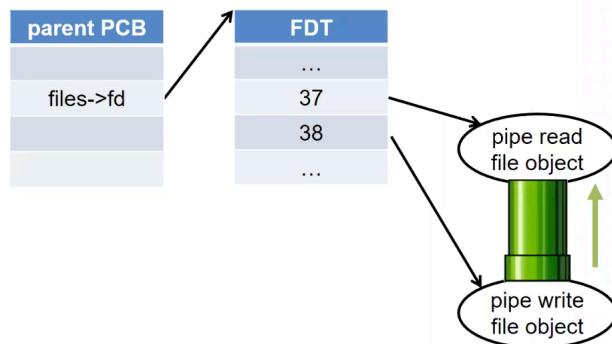
- ערוֹץ תקשורת חד-כיווני בסדר . □ □ □ □
- *pipes* מאפשרים העברת מידע בין תהלייכים.
- ניתן להשתמש ב-*pipes* גם לנטרין בין תהלייכים.
- מערכת הפעלה לינוקס ממשת *pipes* באמצעות "קבצים".

## מערכות הפעלה

- 44
- קיימן  $FD$  לכתיבת, ו-  $FD$  לקריאה.
  - קריאה וכ כתיבה כמו לקבצים רגילים (ע"י קריאות המערכת  $read/write$ )
  - המימוש אינו צריך כל שטח דיסק, ואינו מופיע בהיררכיה של מערכת הקבצים.
  - המידע שנכתב ל-  $pipe$  נשמר בחוצצים (*buffers*) של  $read/write$ .

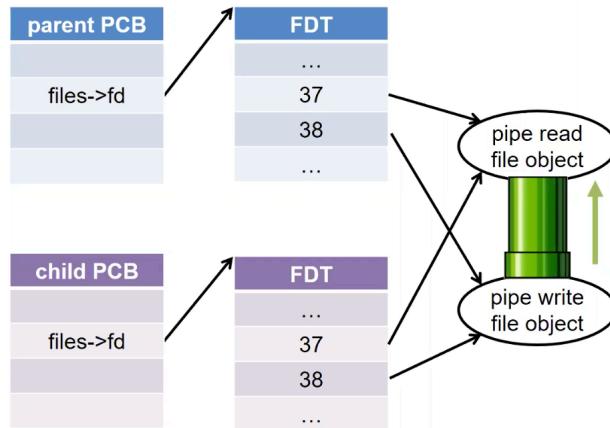
יצירת  $pipe$  חדש.

- פעולה: יוצרת  $pipe$  חדש עם שני  $FD$  (שני קצוות הציגור): אחד לקריאה מה- $pipe$  ואחד לכתיבה אליו.
- פרמטרים: - *filedes* - מערך בן שני תאים.
  - ב- $filedes[0]$  יאוחסן  $FD$  לקריאה מה- $pipe$ .
  - ב- $filedes[1]$  יאוחסן  $FD$  לכתיבה מה- $pipe$ .
- הכניסות המוקצחות ל-  $FD$  הן הראשונות הפנויות ב- $FDT$ .
- ערך מוחזר: 0 בהצלחה ו- (-1) בכישלון.

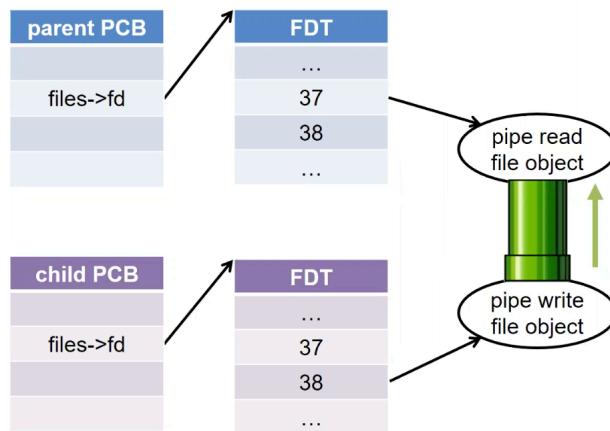


ניתן לראותות של אבא עשוינו  $pipe$  וזו הכניסה 37 מיודעת לקריאה ו- 38 אז לכניסה כתה, אם ועשה  $fork$  הבן גם יציבע לאותו דבר כלומר קיבל הבא:

## מערכות הפעלה



از עכשו נותר לסגור קצוות של כל אחד שלא משתמש בו וזה חשוב  
נקרה אחר כך בדוגמא למטה.



- ה-*pipe* הנוצר הינו פרטி לתהלייך ואינו נגיש לתהלייכים אחרים.  
שיתוף *pipe* יבוצע בדרך אחת בלבד - בעזרת קשרי משפחה:  
- תהלייך אב יוצר *pipe* - שתי כניסה חדשות נוספות ל-*FDT*-*fork* שלו.

- תהלייך האב יוצר תהלייך בן באמצעות *(fork)* - ה-*FDT* משוכפל לתהלייך הבן.
- כתת לשני התהלייכים, האב והבן, יש גישה ל-*pipe*-*pipe* באמצעות ה-*FD* שלו.

## מערכות הפעלה

46

- *pipe* הוא חד-כיווני ולכן האב והבן משחקים תפקדים שונים:
  - האב קורא והבן כותב, או  $\leftarrow$  הפוך.
  - האב והבן צריכים למסור את ה-*FD* השני שאינו בשימוש.
- \* לאחר סיום השימוש ב-*pipe* מצד כל התהליכים (סגירת כל ה-*FD*) מפונים משאבי ה-*pipe* באופן אוטומטי

דוגמה. ראשית ננתן את הקוד הבא:

### תכנות דוגמה – *pipe*

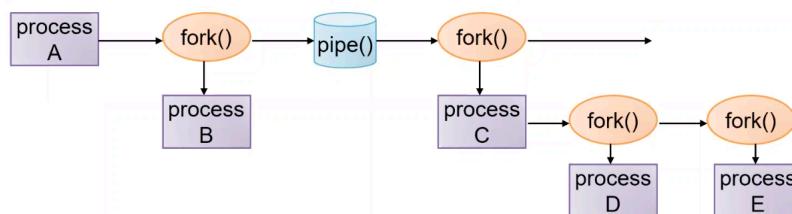
```
int main() {
 int my_pipe[2];
 char buff[6];
 pipe(my_pipe);

 if (fork() == 0) { // son
 close(my_pipe[0]);
 write(my_pipe[1], "Hello", 6);
 } else { // father
 close(my_pipe[1]);
 read(my_pipe[0], buff, 6);
 printf("Got from pipe: %s\n", buff);
 }
 return 0;
}
```

מה קורה אם האבא מתחיל לרצץ לפני הבן?

ראשית תהליך הבן נוצר ואז הוא סגור את הקצה שבו  $\leftarrow$  ישתחש ואז מתחל לכתוב  $\leftarrow$  אב במקרה שהאב התחל לרצץ קודם קודם אז הוא ממחה עד שהבן יכתוב ואז מדפיס למסך.

- למי מהתהליכים הבאים יש גישה ל-*fd* שיצור תהליך A?



תשובה. כל התהליכים יש *close* *pipe* הטענו אחרי ה-*fork* שלו.

## מערכות הפעלה

47

### למה צריך לסגור קצוות מיותרים?

```
int main() {
 int fd[2];
 int grade;

 pipe(fd);
 if (fork() == 0) { // teacher
 close(fd[0]);
 do {
 grade = get_random_between(0, 100);
 write(fd[1], (void*)&grade), sizeof(int));
 } while (grade != 0);

 } else { // student
 close(fd[1]);
 while (read(fd[0], (void*)&grade), sizeof(int)) > 0) {
 if (grade == 100) break;
 }
 printf("Grade = %d\n", grade);
 }
 return 0;
}
```

מה יקרה אם נסיר את השורה הזאת?

מה יקרה אם נסיר את השורה הזאת?

?  
גיאוטרים

הסירה חשובה כי כל תהליך ידע אם השני הולך לכתוב או לקרוא.  
נסתכל בקטע הקוד הבא שבו המורה מגറיל מספרים והתלמיד קורא  
את המספרים עד שהוא קורא את המספר 100 ואז הוא מסיים. אז ה  
close הם כאלה מחליפים המתנות כי נניח שהסטודנט לא סגר ערז  
הקריאה עצמה אז הוא לא ידע שהאב הפסיק כי עדין פתוח הצד שלו של  
הpipe של הכתיבה ויתקע, ואם האב לא סגר הצד שלו של ה pipe  
אז הוא לא יהיה מודע מהבן שהוא סיים ואז ימשיך להגריל מספרים  
במקרה טוב יגריל 0 ויסיים וברע יגריל עד ש ה pipe יתמלא ואז יצא  
להמתנה וימתן שם לנצח.

קריאה המערכת () .dup

- פועלה: משכפלת את ה-  $FD_{oldfd}$  במספר  $FD_{newfd}$  אחר בטבלה.

- עבור dup: ה-  $FD_{newfd}$  החדש יהיה ה-  $FD_{oldfd}$  הפוי בעל העורך  
הנומך ביותר בטבלה.

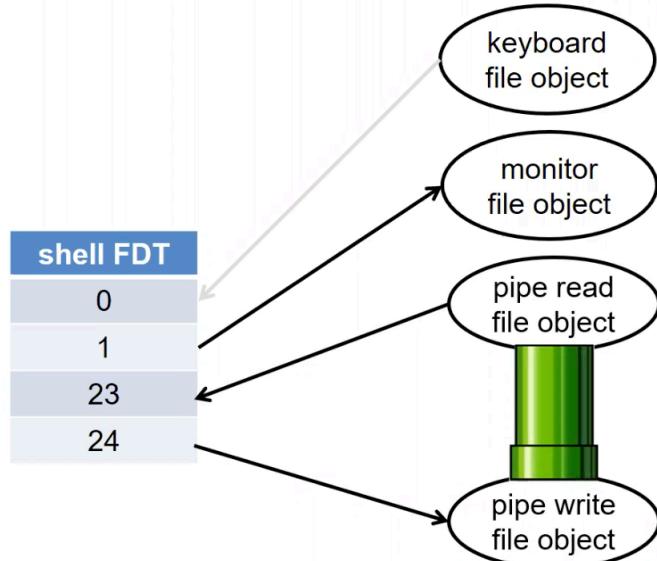
## מערכות הפעלה

48

- עבור  $newfd$  החדש הינו  $FD\_2dup$ , לאחר סגירה, אם היה פתוח.
- לאחר פועלה מוצלחת, וה  $FD\_oldfd$  החדש מצבאים לוותנו  $.file-object$
- פרמטרים:
  - המועד החדש  $FD\_oldfd$  - חייב להיות פתוח לפני ההעתקה
  - ערך מוחזר:
    - בהצלחה, מוחזר החדש.
    - בכישלון מוחזר(-1).

## הכוון זר/ספין באנצטוף קאוד - פונקציית

```
// shell Code
int fd[2];
pipe(fd);
```



>> ls | more

202

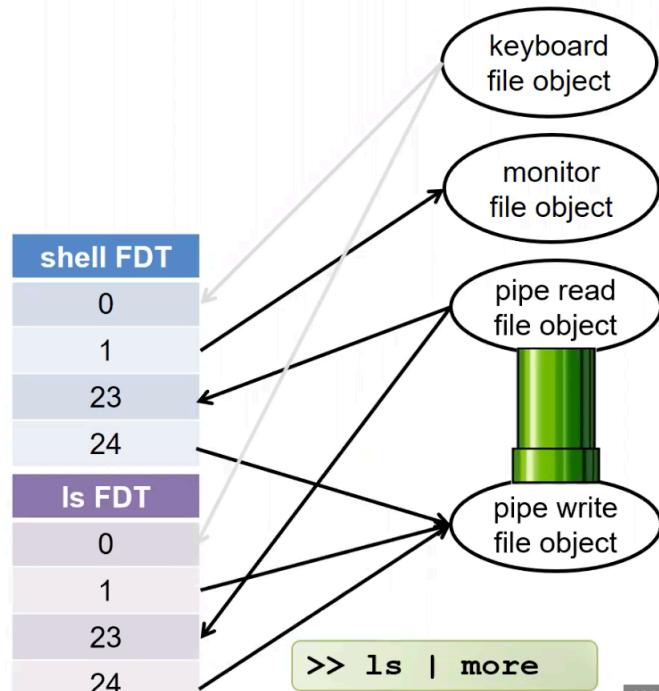
## מערכות הפעלה

### היכלון של ספיבר באנציוור צדוק - פונקציית

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
 // first child
 dup2(fd[1], 1);

 execv("/bin/ls", ...);
}
```

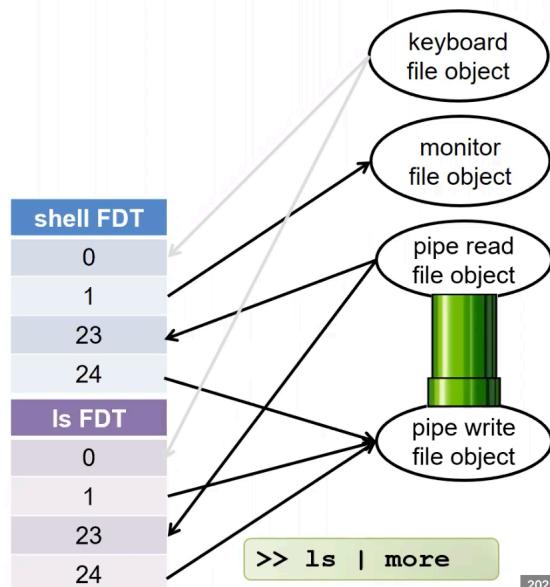


### היכלון של ספיבר באנציוור צדוק - פונקציית

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
 // first child
 dup2(fd[1], 1);

 execv("/bin/ls", ...);
}
```



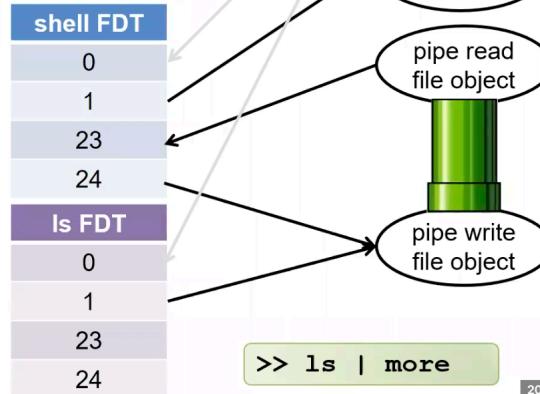
## מערכות הפעלה

50

### לכונון קיוסק באנציוור סאדין - פונקציית

```
// shell Code
int fd[2];
pipe(fd);

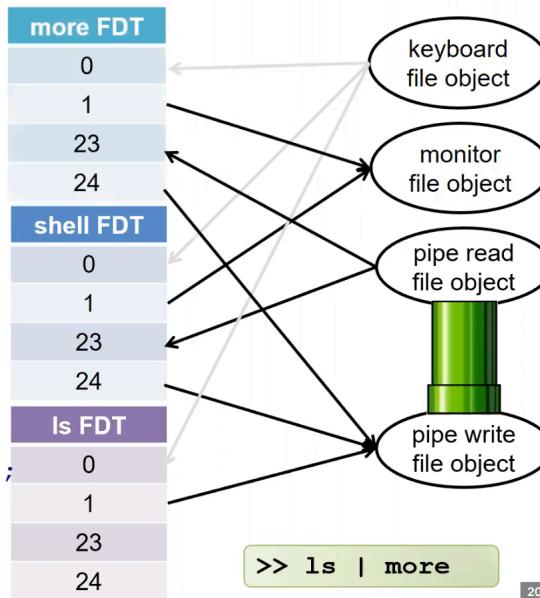
if (fork() == 0) {
 // first child
 dup2(fd[1], 1);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/ls", ...);
}
```



### לכונון קיוסק באנציוור סאדין - פונקציית

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
 // first child
 dup2(fd[1], 1);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/ls", ...);
}
if (fork() == 0) {
 // second child
 execv("/bin/more", ...);
}
```



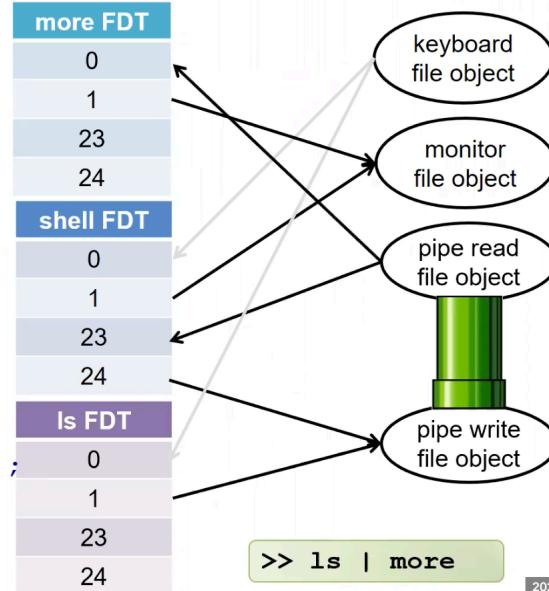
## מערכות הפעלה

51

### לכונון קיוסק באנציוור צדוק - פונקציית

```
// shell Code
int fd[2];
pipe(fd);

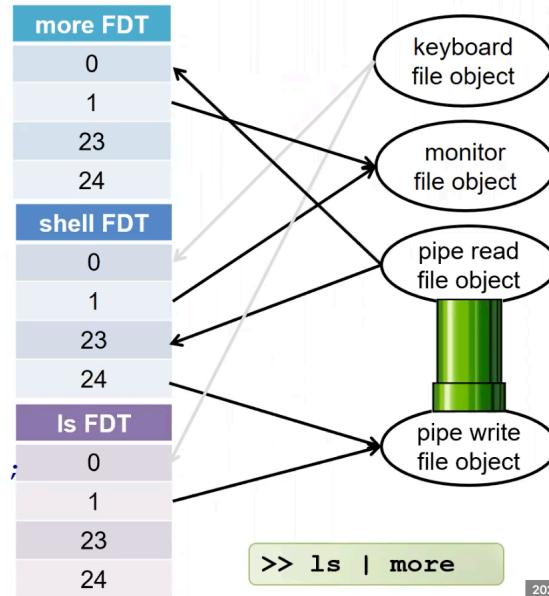
if (fork() == 0) {
 // first child
 dup2(fd[1], 1);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/ls", ...);
}
if (fork() == 0) {
 // second child
 dup2(fd[0], 0);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/more", ...);
}
```



### לכונון קיוסק באנציוור צדוק - פונקציית

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
 // first child
 dup2(fd[1], 1);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/ls", ...);
}
if (fork() == 0) {
 // second child
 dup2(fd[0], 0);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/more", ...);
}
```



202

202

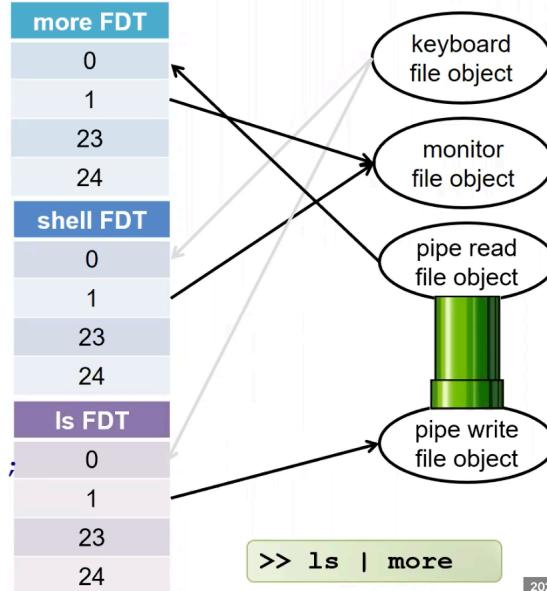
## מערכות הפעלה

52

### ווכוונןizi/C/FIFO באנצ'וונ צקוק - פונו זו

```
// shell Code
int fd[2];
pipe(fd);

if (fork() == 0) {
 // first child
 dup2(fd[1], 1);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/ls", ...);
}
if (fork() == 0) {
 // second child
 dup2(fd[0], 0);
 close(fd[0]);
 close(fd[1]);
 execv("/bin/more", ...);
}
close(fd[0]);
close(fd[1]);
```



202

תרגול 4

תזכורת. ראיינו בתרגול 3 דיברנו קצר על מערכות הפעלה, ועל תהליכיים ובשייעור אחרון דיברנו על תקשורת בין תהליכים למשל באמצעות *pipe* או *everything* – *riderection* אבל מה שחשוב שבלינוקס – *everything* – *is-a-file* קבצים רגילים אבל בלינוקס גם *pipe* הם קבצים וan *FIFO* הוא קובץ וגם *sockets* משמשים בין תהליכיים לאינטרנט והיום נראה שגם התקנים הם קבצים.

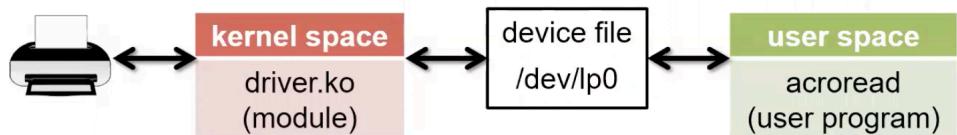
- מודולים בלינוקס הם תוספות קוד לגרעין שניית לטעון בזמן ריצה.
  - לא צריך להדר מחדש את הגרעין, לא צריך לאותחל את המחשב.
- מודולים משמשים להוספת פונקציונליות חדשה לגרעין, לדוגמה:
  - דרייברים להתקני חומרה שמחברים/מנתקים מהמערכת:
  - התקני תווים – מקלדת, עכבר, מדפסת, ...
  - התקני בלוקים – דיסק קשיח, disk – on – key, ...

מה קורה כאשר לוחצים על לחץ הפעלה את המחשב? חלק ממה שקרה על *power – on* זה שמערכת הפעלה נטענת מדיסק לזמן,

והיא לא נטענת עם כל הדרייברים וכל התמיכה לכל ההתקנים בעולם כי

## מערכות הפעלה

זה היה ממש לא יעיל, שכן משתמשים במודלים שונים שפות קוד תוכנות קטנות שניתן לטעון בזמן הריצה. למשל בwindows שימושים *disk* המחשב טיפה חושב ומחפש את התקן הדריבר כדי לתקור עם ההתקן זהה אז בעצם הדריבר שמתקשר עם הדיסק ששמו נקרא מוד. ומה שטוב בהם שאפשר להוסיף אותו למערכת הפעלה בזמן ריצה מבלי להדר מחדש מוחדר המערכת או להריץ מחדש את המחשב. למודלים יש הרבה שימושים אבל השימוש הכל השוב זה למשתמש דרייברים, ודרייברים הם בעצם שכבת תוכנה שאחראית למשתמש עבור מערכת הפעלה את התקשרות עם התקן.



למשל למעלה רואים שיש מדפסת ומצד שני יש את התוכנה שמדפסה למשל במקרה הזה *acroread* שהוא ב-user – space שהדף י יצא מה קורה בדרך? בעצם המדפסת מיוצגת על ידי קובץ שישוב במערכת הקבצים בתיב `/dev/lp0` ואם *acroread* רוצח להדפיס מסמך הוא פותח קובץ של המדפסת וכותב אליו. עכשו אם כתובים לקובץ רגיל אז המידע עובד לדיסק ועכשו שכותבים לקובץ מדפסת מה שקרה זה שהמידע יצא על דף מהמדפסת זהה כל הרעיון. כמובן לכתוב לקובץ רגיל ולקובץ של המדפסת זה לא אותו דבר וכי שמיין זהה לא אותו דבר וכי שאחראית למשתמש את הפונקצונאליות זאת בaczורה שונה על כל התקן זה הדריבר והוא חלק במערכת הפעלה בז"א שמקומים לינוקס היא לר מגעה עם תמייה של כל ההדפסות בעולם זהה הגיוני כי יהיה בעיה לתמוך בכל המדפסות בעולם למשל *hp*, *lenovo* תומכים בכל אלה אז מה שקרה שמחברים את המדפסת ושם חשב בזמן עלייה מזהה שם מדפסת הוא טוען את המודול המתאים לזכרון רק למדפסת הזאת.

## מערכות הפעלה

:Linux boot sequence

- כאשר המחשב כבוי, מערכת ההפעלה, היחסומים וכל הקבצים של המשמש נשמרים על הדיסק.
- כאשר המשתמש מדליק את המחשב (באמצעות לחיצה על כפתור POWER/ON), מערכת ההפעלה בדרכן כלל לא נמצא עדין בזיכרון ולכן צריך לטעון אותה מהדיסק.
- יש פה פרודוקס: כדי לטעון את מערכת ההפעלה מהדיסק לזכרון צריך מערכת הפעלה שiodעת לגשת להתקנים כמו הדיסק.
- הפרודוקס נפתר באמצעות תהליך אתחול הדרומי של המחשב: סדרת רכיבי תוכנה שהמחשב מבצע כאשר המשתמש מדליק אותו.

### מה קורה לאחר הדלקת המחשב?



אנו נעבור על אחד מהתוכניות האלו ונראה מה כל אחת עשויה.  
האם Bios נמצוא על הזיכרון? לא, Bios נמצא על האם – mother board. אגב, איך שלב ה boot – loader יכול לטעון windows. board. ולאו דווקא לינוקס.

מי מטפל בפסיקות שמגיעות לפניו שמערכת ההפעלה עולה? ה BIOS ורם ה LOADER יודעת לטפל בחלק מהפסיקות אבל לא באופן

## מערכות הפעלה

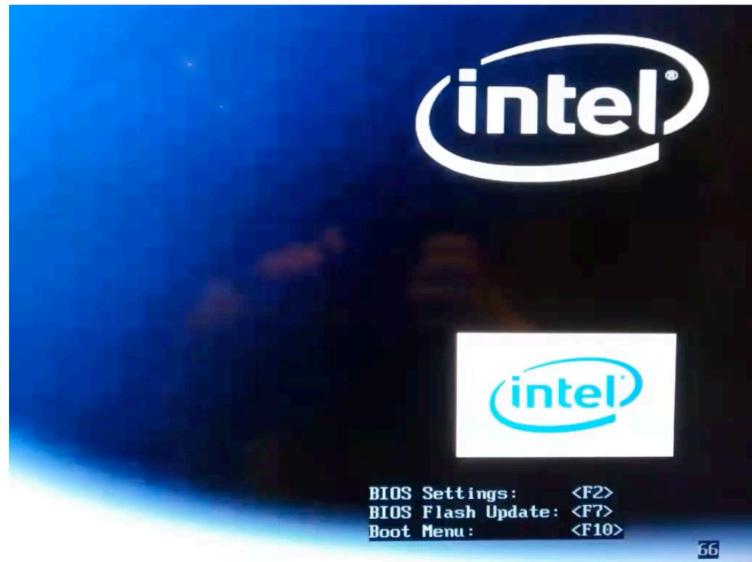
55  
מקיף כמו מערכת ההפעלה. ואם יגעו פסיקות קritisיות או פסיקות שלא הגדרנו איך לטפל בהם אז הוא יתעלם אבל אם הן קritisיות ואי אפשר להתעלם אז המחשב יכול לקרוס.

:*BIOS* (*basic output/Output system*)

- התוכנה הראשונה שמופעלת לאחר הדלקת המחשב.
- קוד ה-*BIOS* צרוב בזיכרון ייעודי בלבד האם ואז נטען לכתובות קבועה בזיכרון באופן אוטומטי ברגע הדלקת המחשב.
- **פקידי ה-*BIOS*:**
  - לזהות את התקני החומרה המוחברים למערכת.
  - לבודק שהתקנים הבסיסיים (מסך, מקלדת,...) פועלים בצורה תקינה.
  - לעبور על רשימה מוגדרת מראש של התקנים, ולחשוף התקן המאפשר אתחול (*bootable device*).
  - אם לא נמצא אף התקן זה, ה-*BIOS* רושם הודעה שגיאה למסך ומפסיק את תהליך האיתחול.
  - אם נמצא התקן זה, ה-*BIOS* טוען את הסקטור הראשון של התקן למקומו קבוע בזיכרון.

מי האחראי שמשמש את ה-*BIOS*? כל חברה יש לה – *mother board* נגד ליאנט יש זהה והוא מייצרת *BIOS* וצורבת אותו בזיכרון *read-only-memory* של ה-*BIOS* הינה צורב על *firmware updates* שיש באגים ולפעמים נרצה לעדכן את ה-*BIOS* וחילקו היום מכיריים או אינטלי, והוא עידכן את ה-*BIOS* וראינו מסך חדש אין נראה מסך ?*BIOS*

## example BIOS screen



האם ברשימה של *bootable devices* מכילה את מערכות הפעלה לינוקס ווידוס? התשובה היא לא, בשלב זה הוא רק מחפש התקן להמשיך שרשרת האתחול.

:*MBR*( *master boot record*)

- דיסק קשיח הוא התקן איחסון המחולק לסקטורים בגודל 512 בתים.

- קרייה/כתיבה מהדיסק נעשית בכפולות של סקטורים.

- \* אם נרצה לכתוב בית אחד לדיסק אנו חיבים לקרווא את כל ה 512 ולבדק את הבית הספציפי שרצים ולכתוב בחזרה 512 בתים לסקטור המתאים בדיסק.
- אם הדיסק הוא התקן המאפשר אתחול (*bootable device*, *MBR*(*master boot record*) או הסקטור הראשון בדיסק נקרא *GRUB*).
- ה-*MBR* מכיל קוד אSEMBLY בסיסי המשמש לטעינת קוד נוסף: מנהל האתחול (*bootloader*).
- יש מספר *boot loaders* נפוצים. מערכות לינוקס משתמשות בדளן כללי ב-*GRUB*.

## מערכות הפעלה

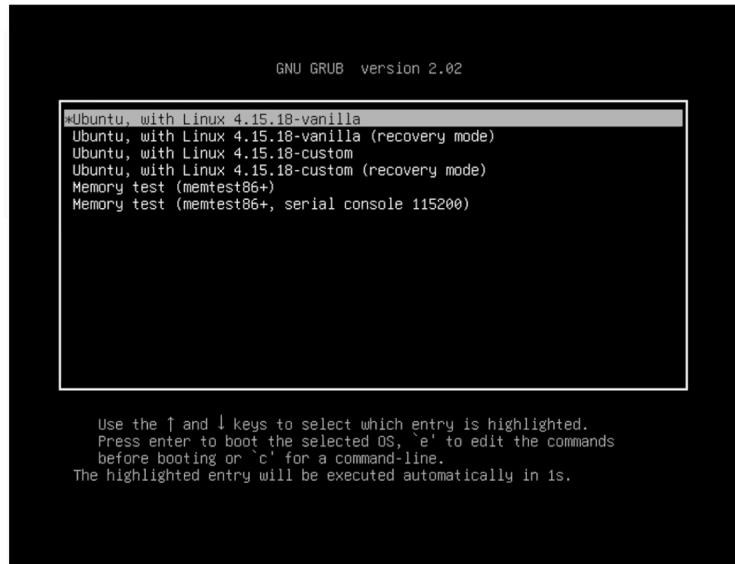
57

הערה.. אם במקרה הסטודנטי ורוצה לטעון מהמערכת הפעלה והקוד הנוס, שהוא בעצם ה-*boot* – *loaded*. למה לא יכולם לשמר את הקוד בסקטור הראשון בדיסק דוגרי? הבעיה היא שהקיד *grub* יותר גדול מ 512 בתים ואי אפשר להכנס אותם לסקטור. לכן אנו בהתחלה טוענים את ה- *MBR* והוא טוען את ה- *grub* מדיסק לזכרון.

### *GRUB (GRand Unified Bootloader)*

- בשקף הבא מופיע דוגמה של תפריט ה-*GRUB*: הוא מאפשר לבחור את גרעין מערכת הפעלה אשר יטען לזכרון.
- תפריט ה-*GRUB* שימושי כאשר מפתחים גרעין לינוקס חדש(כפי שתעשו בשיעורי הבית) :
- נניח כי עדכנתם את קוד הגרעין של לינוקס ושמרתם אותו בתמונה *custom* – 4.15.18 – *boot/vmlinuz* על הדיסק.
- לאחר מכן ניסיתם לטעון את התמונה זה לזכרון ומערכת הפעלה קרסה לכוון יש באגים לפעמים...  
cutת תוכלו להפעיל מחדש את המחשב ולטעון תמונה אחרת ותקינה,
- למשל *vanilla* – 4.15.18 – *boot/vmlinuz* מתוך הדיסק.
- לאחר עליית מערכת הפעלה, תוכלו לנסוח ולחזור את התמונה *custom* – 4.15.18 – *boot/vmlinuz* על הדיסק.

## example GRUB menu



לآن מפריע לנו שיהיה כמו מערכות הפעלה, בסופו של דבר רק מערכת הפעלה אחראית לנו לזכרון ולהריצ. למשל למעלה בחרנו מערכת הפעלה ראשונה באופציות. והוא נותן רק 5-10 לבחן ואם לא יבחר אופן אוטומטי מה שיש על המסך. נשים לב שהזה שגתי להגדיר רק מערכת הפעלה אחת אלה יש כמה יושבות על הדיסק אחת מהן תיתען לזכרון וניא נבחרת בשלב זהה של ה *grub*. למה זה שימושי? נניח שיש לנו שני גרסאות של לינוקס בעולם ה *linus* גרסה *vanilla* הוא מתאר גרסה נקייה שהורדנו *git* ולא נגענו בה והירא גרסה רשמית של לינוקס אז סביר להניח שלא מכילה באגים נטעלים כרגע מה *recovery mode*. נניח שאנו מפתחי לינוקס ונרצה להכניס שינויים וכי לפעם יש באגים אז אני ערכתי המקור של לינוקס ויצרתי תמונה חדשה שנקראת *costum* והיא הגרסה שבה עורכים את השינויים שלי ולפעמים היא מכילה באגים, ברמה שאני עלה הגרסה לזכרון ואתהיל לתקן את המערת ההפעלה והיא תקרוס לגמרי או לא תצליח להעלות בכלל. אז איך ניתן לתקן בכל זאת? פשוט אני אפעיל את המחשב מחדש ונתען את גרסה *vanilla* ונתען מdisk לזכרון ונדכן ווורק קבצים

## מערכות הפעלה

<sup>59</sup>  
וקוד ומקמפל תמונה חדשה של *costum* וIOSHE עוד פעם וככל פעם מנסה להעלות *costum* ואם שוב קרס נעלב שוב *vanilla* וכך יש שני גרסאות לינוקס שחיות על הדיסק וכל פעם נתען אחרת תלויה מהם שנרצה להציג.

וינדוס מאוד בעיתוי ולא נחמד כי התקנות ווינדוס רגילות לחשוב שהן לבד בעולם. למשל אם נתקין ווינדוס הוא ימחק את ה *grup* מהדיסק וישים אֵה *boot – loader* של ווינדוס שנוא טוען רק ווינדוס תמיד. לעומת זאת אם הורדנו וינוס אז לינוקס אז היינו רואים שבתפריט ה *grup s* כתוב *windows*.

האם קבצים שנוצרו במערכת הפעלה אחת יכולים להיות זמינים במערכת הפעלה אחרת? זה לא כל כך פשוט אבל תיאורטית זה אפשר אם אנו מספיק זהירים.

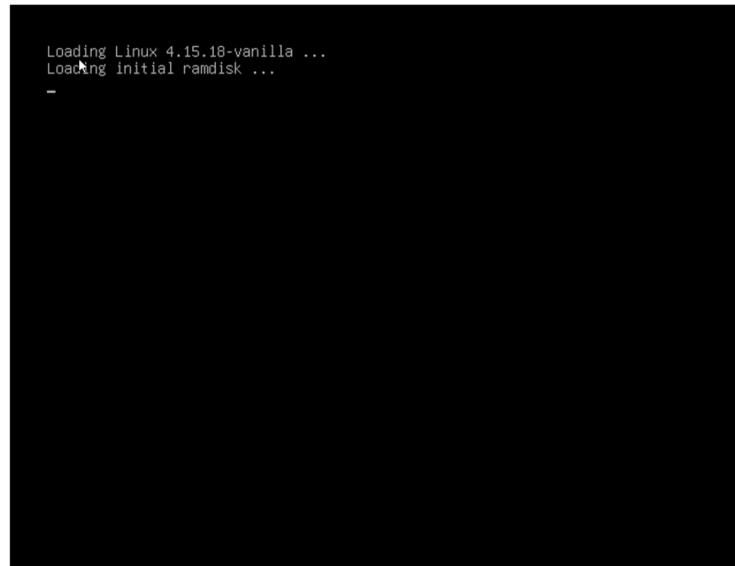
טעינה גרעין לינוקס:

- גם גרעין לינוקס נתען בשלבים. בהתחלה *GRUB* טוען לזכרון *bzImage* תמונה דחוסה של הגרעין, אשר נקראת בדרך כלל או *zvmlinuz*, ועוד מחלץ אותה.
- התמונה הדחוסה של גרסה לינוקס 4.15.8MB שוקלת בערך 4.15MB.
- לצד תמונה הגרעין, *GRUB* טוען לזכרון גם את מערכת הקבצים הראשונית: מערכת קבצים קטנה בשם *init ram – disk* או *initrd*. חשוב לעשות זאת כי לא יכולים לגשת לדיסק. כי לינוקס נתען ועוד לא יודע איפה מערכת הקבצים. ופקיד של מערכת הקבצים היא להזכיר מודולים חיוניים ביויר ובדרכן כלל המודולים הללו ממשים את הדריברים של הדיסקים שעובדים אותם או כרטיס הרשת במידה ומערכת הקבצים נשען ברשותם עובדים על כונן משותף. ואחרי שמערכת הקבצים האמיתית עלתה ניתן למחוק את *initrd* ובמערכת הקבצים האמיתית יש עוד מודולים שהם יאפשרו לנו למצוא את הדריברים של המדפסת או עכבר או התקנים נוספים.
- *initrd = initial RAMdisk*

## מערכות הפעלה

- 60
- *initramfs = initial RAM file – system* •
  - גרעין לינוקס מרייך את התוכנית */init* מתוכן מערכת הקבצים
  - הראשונית */init* היא בדרכן-כלל סкриיפט *. shell*.

## טיענת גרעין לינוקס



שהמחשב עולגה יש המון הודיע על הסמן ואם נרצה לראות אחר כל אפשר להשתמש *dmesg* שמספרת מה קורה בזמן להעיר המחשב.

## מערכות הפעלה

61

### >> dmesg | less

```
[OK] Started Tell Plymouth To Write Out Runtime Data.
[OK] Started Set console font and keymap.
[16.010171] Adding 4015100k Swap on /swap.img. Priority:-2 extents:5 across:4293628k FS
[OK] Activated swap /swap.img.
[OK] Reached target Swap.
[OK] Started udev Coldplug all Devices.
[OK] Started Dispatch Password Requests to Console Directory Watch.
[OK] Reached target Local Encrypted Volumes.
[16.046103] systemd-journald[505]: Received request to flush runtime journal from PID 1
[OK] Mounted Mount unit for core, revision 8268.
[OK] Mounted Mount unit for core, revision 8935.
[OK] Started QEMU KVM preparation - module, ksm, hugepages.
[OK] Started Flush Journal to Persistent Storage.
 Starting Create Volatile Files and Directories...
[16.252771] iscsi: registered transport (iser)
[OK] Started ebtables ruleset management.
[OK] Started Load Kernel Modules.
 Mounting FUSE Control File System...
 Mounting Kernel Configuration File System...
 Starting Apply Kernel Variables...
[OK] Mounted Kernel Configuration File System.
[OK] Started Create Volatile Files and Directories.
 Starting Update UTMP about System Boot/Shutdown...
 Starting Network Time Synchronization...
[OK] Mounted FUSE Control File System.
 Mounting VMware vmblock fuse mount...
[OK] Started Update UTMP about System Boot/Shutdown.
[OK] Started Apply Kernel Variables.
[OK] Mounted VMware vmblock fuse mount.
[OK] Started Network Time Synchronization.
[OK] Reached target System Time Synchronized.
[OK] Started AppArmor initialization.
[OK] Started Authentication service for virtual machines hosted on VMware.
 Starting Service for Virtual machines hosted on VMware...
[OK] Started Service for virtual machines hosted on VMware.
 Starting Initial cloud-init job (pre-networking)...
```

עצמו צריך לדעת לגשת לדייסק בשיביל לטעון את – BIOS masterboot – record או מכיל דרייברים בסיסים אבל בעתיד נרצה להשתמש בדייסקים יותר משוכלים למשל דייסקים שנמצאים ברשות.

מודולים (modules).

- מודולים אפשרים להוסיף לגרעין לינוקס, בזמן ריצה, קטעי קוד חדשים.
- מודול הוא ספריה משותפת (shared library) הנטענת (מקושרת) בזמן ריצה ופועלת במצב גרעין (כלומר  $CPL == 0$  ).
- הפקודה `insmod` טוענת מודול חדש.
- הפקודה `rmmod` פורקת מודול שנטען בעבר.
- הפקודה `lsmod` מציגה את רשימת המודולים הפעילים (כלומר, שנטענו ע"י המשתמש).
- כמה משמשים מודולים?
  - מודולים משמשים בעיקר על מנת להוסיף תמיכה בהתקני חומרה (devices) ע"י דרייברים (drivers):
  - התקני תווים – מקלדת, עכבר, מדפסת, ...

## מערכות הפעלה

62

- התקני בזיכרון - Disk קשיח, ...
  - התקני רשת - נראה בתרגולים הבאים.
- למעשה, מודולים רצים בהרשות גרעין ולכון הם יכולים להוסיף ולעדכן כל פונקציונליות של הגרעין:
  - להוסיף מערכות קבצים חדשות.
  - להוסיף קריאות מערכת חדשות ו/או לשנות קריאות מערכת.
- אם נדמיין שיש מערכת הפעלה עם כל הדריברים המתאימים אז זה בעיתי בכך שמדוברים כי הם מספקים גמישות, למשל אם נוצר רכיב חומרה חדש ונרצה לחבר למערכת והאם אז צריך לקבל מערכת הפעלה חדשה? פשוט עדיף שישלחו לנו דריבר חדש ומודול חדש ואז מתקנים על מערכת ההפעלה הקיימת.
- הדריבר תקין לתקשר עם רכיב החומרה לשלווח פקודות ולקבל ממנו מידע. והוא נוסך על ידי מודול חדש.

יתרונות השימוש במודלים.

- ניתן להוסיף יכולות חדשות לגרעין מבפנים ל�מפל אותו ומבפנים לאתחל (reboot) את המערכת.
- מודולים מפותחים בנפרד מהגרעין כלומר, פחות שורות קוד בגרעין אך זמן קומPILEציה קצר יותר.
- אין צורך ל�מפל פונקציונליות מיותרת, למשל דריברים לחומרה שלא נמצאת ברשותנו.
- הגרעין תופס פחות זכרון.
  - המשתמש טוען לזכור רק מודולים שהוא זוקן להם.
  - ניתן לפרק מודולים שאינם בשימוש ולשחרר זיכרון.
- ניתן להוסיף בעתיד תמיכה בחומרה חדשה שעדיין לא קיימת.

חזרה. נניח שיש לנו מדפסת והתוכנה היא *acoread* שרושה להשתמש במידפסת זו היא משתמש בקריאה מערכת *open* לפתיחת קובץ חדש והקובץ הזה הוא של המדפסת */dev/lp0* אחר כן קרא ל- *write*

## מערכות הפעלה

63

וכתוּב לקובץ של המדפסת כלומר פקודת המערכת *write* צריכה להעיבר פקודה למדפסת שתדפיס את הדף והשתמשנו בה בשתי קריאות מערכת *open/write* על קובץ של המדפסת אבל איך הפקודות הללו יודעתו זהה מדפסת ומה לעשות אז זה לבדוק התפקיד של הדרייבר שהוא אחראי למשם את כל קריאות המערכת של הקבצים עבור התקן הספציפי שמסתכלים עליו, והוא אפשר לזרוב אותו חלק ממערכת הפעלה או לטעון באופן דינמו בתור מודול זה בעצם תוספת קוד שנתינת בזמן ריצה של מערכת הפעלה, ונתן למשם דרייברים בתור מודולים כי אם נרצה למשם את כל הדרייברים שיש בעולם אז מערכת הפעלה הייתה ענquit וזה לא נשמע עיל וחכם. אבל אפשר באמצעות מודול להוסיף לגרעין גם פונקציונליות חדשה. האם זה אומר שכל הדרייברים של התקנים מותקנים בנפח זיכרון של ה *kernel*? התשובה היא לא, אם הדרייברים הם מודולים אז הם יושבים על הדיסק ואפשר לטעון אותם מהדיסק לזכרון במהלך הריצה אבל בדרך כלל הדרייברים הבסיסים ביותר קיימים מראש במערכת הפעלה, אבלרובם نطענים על הדיסק ונطענים רק אם מחברים התקן מסוים. האם מודול זה דרייבר דינמי? סוג של למרות שמודול יכול למש פונקציונליות חדשה למשל להוסיף קריאות מערכת חדשה ולא חייב בהכרח למש דרייבר אלא כל דבר אחר גם. האם מודולו نطענים לזכרון של *kernel* בזמן הריצה? התשובה כן.

דוגמה. נתבונן בדוגמה הבאה:

## מערכות הפעלה

64

### דוגמת קוד: מודול ראשון

```
#include <linux/module.h>
#include <linux/kernel.h> /* for using printk */
MODULE_LICENSE("GPL");

int init_module(void)
{
 printk("Hello World!\n");
 return 0;
}

void cleanup_module(void)
{
 printk("Goodbye cruel world!\n");
}
```

נקראת בטעינה המודול

נקראת בפרקית המודול

למעלה יש לנו שתי פונקציות *init* והיא נקראת בטעינה המודול ו-*cleanup* נקראת בפרקית המודול. חשוב לציין ש הקוד של המודול הוא קוד גרעין ולכן יכול לקרוא לפונקציות שמכירים למשל *printf* כ*f* היא פונקציה של *libc* ו-*libc* היא ספרייה של המשמש שקיימת במרחב המשמש רק. אנו יכולים רצימם בהרשאות גרעין כחץ מקוד גרעין ולכן יכול לקרוא פונקציות הגרעין והמקבילות של *f* *printk*. *printk* היא פונקציית האלה תמיד צריך להגדיר אותן לכל מודול. עכשו נרצה לקרוא את המודול באמצעות *makefile* וניתן ראות שהוא גם קורא ל-*makefile* אחר

## מערכות הפעלה

65

### מבנה המודול

- קובץ makefile לדוגמה:

```
obj-m += hello.o
all:
 make -C /lib/modules/$(shell uname -r)/build
 M=$(PWD) modules
clean:
 make -C /lib/modules/$(shell uname -r)/build
 M=$(PWD) clean
```

- חייבת להיות התאמה מלאה בין גרסת לינוקס עליה נבנה המודול לבין גרסת לינוקס בה הוא רץ.
- חוור תאיות עשוי לגרום לביעות בזמן ריצה.

### טעינה המודול

```
> make
> insmod ./hello.ko
Hello world!
> lsmod
Module Size Used by
hello 868 0 (unused)
...
...
> rmmod hello
Goodbye cruel world!
```



### טעינת המודול

אחרי שקימפלנו את המודול עם *make* נרצה לטעון אותו לזיכרון עם *insmod* כמו שכותב למעלה, אחר כך הוא נתען לזכרון ונודעים שאז ירשם למסך המחרוזת *hello – world* ובאמצעות *lsmod* השם *hello*. ניתן לראות את המודולים שנטענו לזכרון, וברקע פורקח את המודול שאז ירשם למסך המחרוזת *goodbe – cruel – world*.

העברת פרמטרים למודולים.

- מודולים יכולים לקבל פרמטרים מהמשתמש בזמן טיענות.
- פרמטרים מוגדרים בקוד באמצעות המacro *module\_param*.

## מערכות הפעלה

- 66
- המאקרו צריך להופיע מחוץ לפונקציה. בד"כ ממוקם בתחום הקוד.
  - פרמטר ראשון - המשתנה שיכיל את הפרמטר, פרמטר שני - סוג הפרמטר, פרמטר שלישי - הרשות גישה לקובץ המתאים ב-*sysfs* (לא רלוונטי כרגע).
  - יש להגיד לכל פרמטר ערך ברירת מחדל.
- סוגי פרמטרים לדוגמה: *bool, charp, int, short, byte*. ניתן להשתמש במאקרו *MODULE\_PARM\_DESC invbool* כדי להוסיף תיאור לפרמטר. כל ניהול אוטומטיים יכולם לקרוא את התיאור (אפשר גם עם *.modinfo*).

## העברה פרמטרים למודול

- הגדרת פרמטרים בקוד המודול:

```
#include <linux/moduleparam.h>
int iValue=0; // 0 is the default value
char *sValue;
module_param(iValue, int, S_IRUGO);
module_param(sValue, charp, S_IRUGO);
```

- העברת פרמטרים בטעינה המודול:

```
>> insmod ./params.ko iValue=3
 sValue="hello"
```

ניתן להבהיר כמו שוראים למעלה פרמטרים זה אומר שבשורת הפקודה של הטענה ניתן להבהיר עוד פרמטר למל *iValue* או מחרוזת.

גישה לנוטוני גרעין.

- למודול יש גישה למבנה הנחותיים של הגרעין במידה והוא מצורף אליו הקבצים המתאים (ע"י *.include*).

## מערכות הפעלה

67

```
#include <linux/sched.h>

int init_module(void)
{
 printk("The process is \"%s\" (pid %d)\n",
 current->comm, current->pid);
 return 0;
}
```

comm הינו שדה השומר את שם התוכנית המבצעת. מה שם התוכנית שיודפס במקרה זה?

- ראיינו שבני נתונים הći בסיסי בLinux זה PCB ה PCB – PROCESS – CONTROL – BLOCK – PCB מתאר את התהיליך ולנצח של כל תהליך במערכת יש שומר את כל פרטי מידע על התהיליך למשל ת"ז pid שלו ואת מצבו וטבלת קבצים פתוחים שלו התהיליך, למשל מה שמודול יכול לעשות זה שבזמן עלייה שלו לגשתח *current* זהה ה PCB של התהיליך הנוכחי ולהדפיס את ת"ז כמו שראויים בכך למעלה וגם אז comm כלומר הפקודה של התהיליך הנוכחי. כעת בכך למעלה שטוענים המודול מה יודפס למסך ברגע שטוענים, אז יש תשובות אפשריות זה תהליך *init* שהוא הראשון שיוצר shell או shell אבל נזכיר שהו לא תוכנית בעצמו שמרץ את *insmode* אלא הוא יוצר תהליך חדש תהליך בן והוא זה יעשה شيئا שיטען *insmode execu* לכאן המודול הזה שידפיס את תעודת זהה שהוא יקבץ.

התקנים ודורייברים.

התקנים.

- מיוצגים ע"י קבצים מיוחדים בנתיב */dev/* במערכת הקבצים.
- המשמש עובד מול ההתקן באמצעות הממשק הסטנדרטי ל עבודה מול קבצים - קריאות המערכת *.write()*, *read()*, *open()*

## מערכות הפעלה

68

- לדוגמה: כדי להשתמש במדפסת יש לפתח את הקובץ `/dev/lpd` וائز לכתוב אליו את הטקסט להדפסה.

דרייברים. (מנגנון התקן): פועל בהרשאות גרעין וממפה את קריאות המערכת להלו לפעולות ספציפיות להתקן.

- לדוגמה: הדרייבר של המדפסת "mdb" עם המדפסת בפקודות ספציפיות עבורה (אייפה להדפס על נייר, מתי מסתiyaת שורה של הדפסה,...).

- דרייבר הוא שכבת תוכנה החוצצת בין התקן לבין האפליקציה כדי לספק אבטחה לפעולות התקן הספציפי.

התקני תווים ובלוקים. התקן תווים.

- התקן שניגים אליו רצף של בתים.

- לרוב משמשים להעברת מידע. לדוגמה: מסך, מקלדת.

- בדרך כלל ניתן לגשת להתקן תווים רק באופן סדרתי (ולא אקראי). כי למשל לא ניתן להגיד מקלדת מתחת מה הקלידון לפני 200 בתים אלה יותר בגיןו לגשת באפוי זדרתי בית אחרי בית.

התקן בלוקים.

- התקן שנייגן לגשת אליו רק בכפולות של בלוק (למשל 512 בתים).

- לרוב משמשים לאחסון מידע. לדוגמה: דיסק קשיח, דיסק נשלף (*diskonkey*).

התקן בלוקים מאפשר גישה אקראית למידע שבו.

- לינוקס מוסיפה שכבה נוספת (*page cache*) ומאפשרת לקרוא מהתקני בלוקים גם בתים בודדים.

התקני תווים. התקן תווים מאופיין ע"י שני מספרים: תכלס איך יודעים

אם זה קובץ של התרון או קובץ רגיל לצורכי העניין.

- מספר ראשי (*major number*) - מזיהה את הדרייבר הקשור להתקן

## מערכות הפעלה

69

- מספר שני (*minor number*) - מזהה את התקן הספציפי הקשור לאותו דרייבר. יכולים להיות מספר התקנים, למשל מספר עכברים, המנוהלים ע"י אותו דרייבר).

```
>> ls -l /dev
crw-rw-rw- 1 root root 1, 3 Aug 31 10:33 null
crw----- 1 root root 10, 1 May 12 10:33 psaux
crw----- 1 root tty 4, 1 May 12 10:33 ttys1
crw-rw-rw- 1 root root 1, 5 Aug 31 10:33 zero
```

התקני תווים מסומנים ע"י תווים ראשונה

אם כתוב *c* אז זה אומר זה *character device* ואם הוא תקין אז היה רשום *d* לצורך העניין לכל התקן יש שני מספרים ששמורם אליו המספר הראשי והמספר הראשי הוא מקשר את התקן הספציפי זהה לדרייבר מסוים עוד מעט נראה שלכל דרייבר יש מספר והמספר הראשי שמקשר הדרייבר ששייך לתקן הספציפי זהה למשל יכול להיות שני עכברים אותו דבר משתמשים באותו דרייבר וכן יכול להיות שהדריבר הוא משותף כלמור לשני עכברים יש את אותו מספר ראשי ו כדי להבדיל ביניהם צריך לשמור את מספר המשני שהוא בעצם מבידל בין שני העברים הללו.

התקני תווים פיקטיבים. בLinux יש קונספט מעניין שנקרא התקני תווים פיקטיבים זהה בעצם התקנים שהם לא באמת פיזיים ולא ירטואליים למעשה, הוסיף אותם בגלל שיש להם פונקציונות שבדרך כלל שימושית. ולכלם יש מספר ראשי אחד.

| write()                                     | read()                                | device file                                 |
|---------------------------------------------|---------------------------------------|---------------------------------------------|
| מצילה ולא עושה דבר<br>(המידע שנשלח נדחק)    | מחזירה מיד EOF (end of file)          | /dev/null                                   |
| מחזיקה מיד ENOSPC (no space left on device) | מחזירה רצף אפסים לאחר המבוקש          | /dev/zero                                   |
| כותבת לרצף הבתים האקראי<br>ומותר "רעש"      | מחזירה רצף בתים אקריא שנדרך בזמן ריצה | /dev/random<br>/dev/urandom<br>/dev/arandom |

למשל *dev/null* אפשר לחשב עליו כחומר שחר שאם מנסים לכתוב אליו מיד נעלם וגם אף פען לא נצליח לקרוא ממנו ותמיד מקבל *EOF*. *dev/zero* לא משנה כמה מנסה לקרוא למשל 200 תווים מקבל 200

## מערכות הפעלה

70

אפשרים ואם גנסה לכתוב אליו המידע ייעלם. *dev/full* זה התקן *ENOSPCno space left on device* נקרא ממנו אפשרים ואם גנסה לכתוב אליו קיבל וזה שיהה לנו כמו דיסק והרבה פעמים בזמנים נוח שהיה ההתקן הזה שיהה לנו כמו דיסק שהוא מלא ואז ניתן לסמץ דיסק. יש את */dev/random* שהוא התקן מחזיר רצף בתים אקרים ואם גנסה לקרוא ממנו קיבל רצף בתים אקרים ואם נכתוב אליו נהרס. נניח שיש לנו

```
./spammy-program > /dev/null
```

זה מאוד שימושי בשבייל לפנות פלט, למשל נניח שיש תוכנית *spammy-program* ומפניים פלט שלה ל */dev/nul* זה שימושי לסנן פלט, הקבה פעמים מריצים תוכנית ומדפסה הרבה למסן *junk* זה לא מעניין אותנו בכלל. באותו אופן ניתן לעשות,

```
.cat/dev/zero > file.txt
```

למעשה פה מה שקרה זה שגדול הקובץ *file.txt* יילך ויגדל לניצל אם לא נעשה *c trl* – מה שקרה זה שהפקודה לא תפסיק ותכתוב עוד ועוד אפשרים ל *file.txt* עד(SIGTERM) המיקום בדיסק. כי תקרא *cat* ותקרא *EOF* ופעם אף פעם לא תקבל

יצירת התקן חדש.

```
mknod <name> <type> <MAJOR> <MINOR>
```

- פועלה: יוצרת קובץ התקן חדש. הפועלה דורשת הרשות root.

- פרמטרים:

- שם הקובץ החדש שייצג את התקן. *NAME*.

- סוג התקן (C – התקן תווים, D – התקן בLOBים).

- המספר הראשי של הדרייבר המפעיל את התקן. *MAJOR*.

## מערכות הפעלה

71

- המספר המשני של התקן (מספר בין 0 ל-

χ255

- . *mknod()* מומשת באמצעות קריית המערכת ()  
ניתן להסיר התקן באופן דומה למחיקת קובץ רגיל *rm*.  
דריירים.

תזכורת. שוב יש לבדוק בין התקן ודורייר התקן זה ההתקן הפיזי למשל מקלדת עכבר. והדרייר הוא שכבת התוכנה עם הגרעין שימוש בעור הקבצים האלה *write*, *read* והלאה. קריית המערכת () *open* מקצת כניסה חדשה במקום הפניו הראשון ב-*FDT* של התהlixir. הכניסה מצביעת על אובייקט מטיפוס *struct file*. מה שלא ראיינו זה שבכל *file\_struct* יש מבנה מיוחד שנקרא *file\_operation* שהוא שומר מלא מכך מצביעים לפונקציות זהה הדרייר בעצם.

```
struct file {
 atomic_t f_count;
 ...
 loff_t f_pos;
 mode_t f_mode;
 ...
 void *private_data;
 struct file_operations* *f_op;
};
```

זה למעשה  
הדרייר

- מערכת הפעלה מגדרה אוסף פעולות שנייה לבצע על קבצים - באמצעות קרייות מערכת () *read()*, *write()* ....
- \* בפרט זהו גם אוסף הפעולות שנייה לבצע על התקן תווים.

- כל קובץ פתוח מצביע למבנה נתונים מסווג *file\_operations*,  
שהוא מערך של מצביעים לפונקציות המממשות את אותן הפעולות.  
- *f\_op* הוא השדה המצביע למבנה זה ב-*struct file* -  
מצביע *NULL* מייצג פונקציה לא ממומשת, או מימוש ברירת מחדל.

## מערכות הפעלה

72

- גישה מונחית עצמים:

- הקובץ הוא האובייקט.

- המתודות של האובייקט מוגדרות ע"י אוסף הפעולות ב-

.*fops*

```
ssize_t sys_read(unsigned int fd,
 char * buf, size_t count) {
 ...
 struct file * file = fget(fd);
 if (!file)
 return -EBADF;
 if (!(file->f_mode & FMODE_READ))
 return -EINVAL;
 if (file->f_op && file->f_op->read != NULL)
 return file->f_op->read(file, buf,
 count, &file->f_pos);
 ...
}
```

ציה *.sys\_read()*

הסביר לך. יש לנו למשל את קריאת המערכת *read* למשל ברינוקש ממושת על ידי *sys\_read*. אז מה הקוד למעלה עושים בעצם? נשים לב *sys\_read* צריכה לקרוא מקובץ אבל שוב יש לנו המומן קבצים אפשריים למשל קובט מהדיסק או קובט שמייצג מקלדת וכו', אז מכל *read* צריך לקרוא באופן שונה. אז מה שעושים קודם כל זה ש *read* מקבלת כפרמטר ראשון את *fd* – *file-descriptor* או קודם מקבלת את *file* – *struct* ובודדים שהוא לא *null* אחריה מוצאת את ה *file* מוחזרים שגיאה, ולאחר מכן בודקים את ההרשאות של הקובץ האם בכלל ניתן לקרוא ממנו. אחרי זה ניגשים ל *f\_op* ובודק אם יש מתודה שנקראת *read* ואם יש מתודה שנקראת *read* אז הולכים ומריצים אותה על הארגומטים שהמשתמש העביר לנו.

שדות חשובים ב *file operations*

## מערכות הפעלה

73

- *open* - מצביע לפונקציה לפתיחת התקן. אם מואתחל ל-*NULL*, פועלות(*open()* תמיד תצליח).

*int(\*open)(struct inode\*, struct file\*);*

- *release* - מצביע לפונקציה לשחרור התקן. קריית המערכת(file object) *close()* מושותף למשל, *fork()*, לאחר קריאה ל-*release()*, תבצע קריאה ל-*close()*. רק לאחר שכל העותקים של התקן נסגרו. כמו במקרה של *NULL*, ניתן לאותחל את הפונקציה ל-*open()*.

*int(\*release)(struct inode\*, struct file\*);*

- *flush* - מצביע לפונקציה לנקיי החוצצים וכתיבת המידע בהם ישירות ל התקן. מופעלת כל פעם שתהילין סגור העתק של התקן מסויים. במידה ומואתחל ל-*NULL*, מערכת הפעלה לא תבצע את הפעולה.

*int(\*flush)(struct file\*);*

- *read* - מצביע לפונקציה לקריאה מהתקן. במידה ומואתחל ל-*NULL*, קריית המערכת *read* תחזיר *EINVAL*.  
*ssize\_t(\*read)(struct file\*, char\*, size\_t, loff\_t\*);*

- *write* - מצביע לפונקציה לכתיבה לתקן. במידה ומואתחל ל-*NULL*, קריית המערכת *write* תחזיר *EINVAL*.  
*ssize\_t(\*write)(struct file\*, const char\*, size\_t, loff\_t\*);*

## מערכות הפעלה

74

- *llseek* - מצביע לפונקציה לשינוי המיקום הנוכחי בקובץ. ישפייע על פעולות *read/write*. מוחזר את המיקום החדש בקובץ.

```
loff_t(*llseek)(struct file*, loff_t, int);
```

- *-ioctl* - משמש להעברת פקודות ייחודיות להתקן. במידה ומאותחל *int(\*ioctl)(struct inode\*, struct file\*, unsigned int cmd\_id, unsigned int)*
- נשים לב שיש לנו קריאת מערכת *open* שהיא כללית ויש *open* של דרייברים והן לור אותה פונקציה, קריאת המערכת *open* תקרה ל *open* של הדרייבר כי היא כללית עלפתיה קבצים אבל יש קבצים נפתחים בצורה אחרת וכך כל דרייבר ממש את *open* שלו בצורה שונה. ואם הוא לא ממש למשל אז ישים *null* ואז לא יקראו להם.

## מיימושם דרייברים באמצעות מודולים.

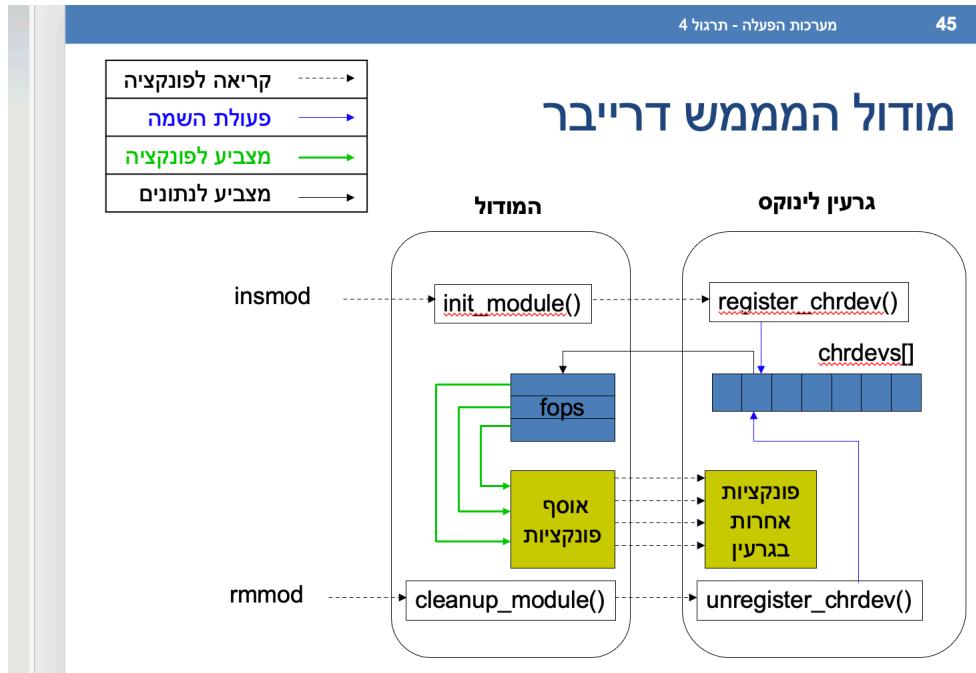
- ניתן לבנות דרייבר בתוור מודול (כלומר, בנפרד משאר הגרעין) ואז "לחבר" אותו בזמן ריצה בשעת הצורך.
- המודול ירשום את הדרייבר במהלך הטעינה שלו. כלומר הפונקציה *() register\_chrdev()* תקרא לפונקציה *init\_module()*.
- שימושו לב: רישום דרייבר רק מקשר בין דרייבר לבין מספר ראשי. אין קשר ישיר בין דרייבר להתקן. עוד לא קישרנו את הדרייבר להתקן צריך ליצור קובץ של ההתקן כמו שראינו מקודם.
- המודול ימחק את הרישום בזמן הפריקה שלו.
- כלומר הפונקציה *cleanup\_module()* תקרא לפונקציה *unregister\_chrdev()*.

## מערכות הפעלה

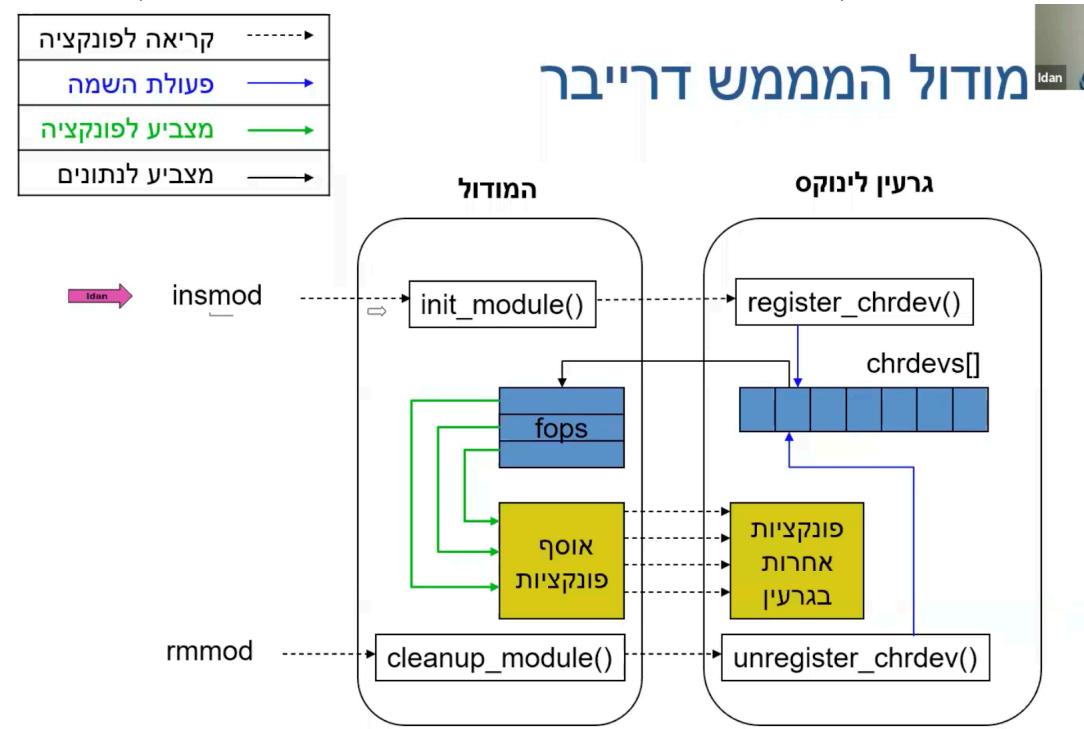
75

מערכות הפעלה - תרגול 4

45



אבחנה. יש הבדל בין דרייבר להתקן *chrdev* זה מערך של דרייברים ולא התקנים בשם שלו מאוד מבלבל כי אנשים שיצרו לינוקס לא עשו

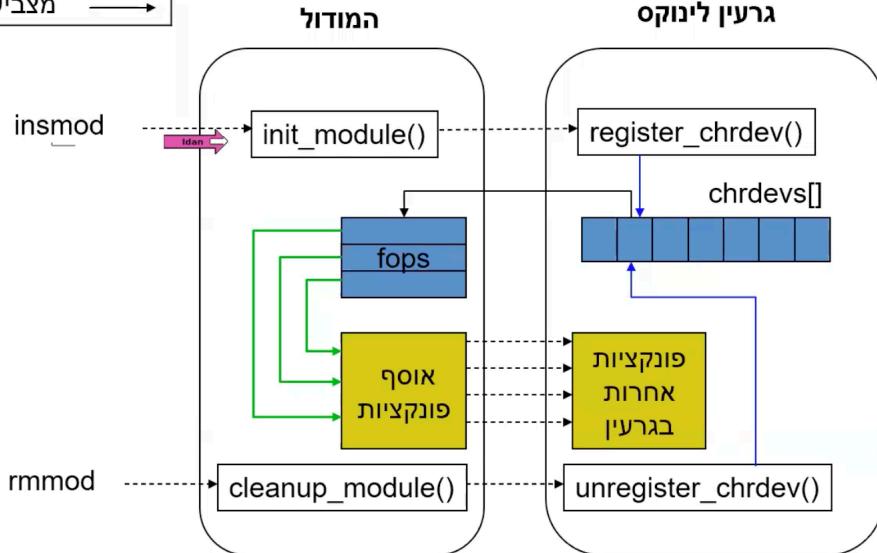


לא אותו דבר.

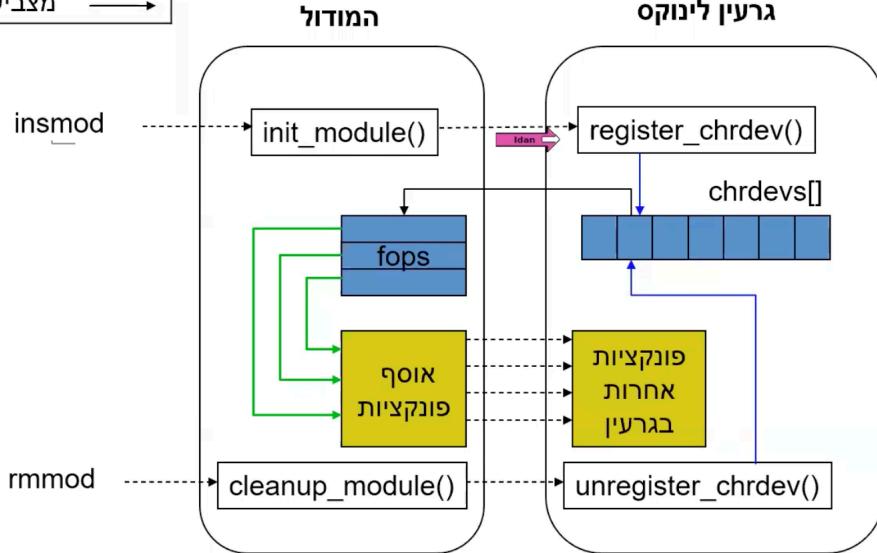
## מערכות הפעלה

76

### מודול המממש דרייבר



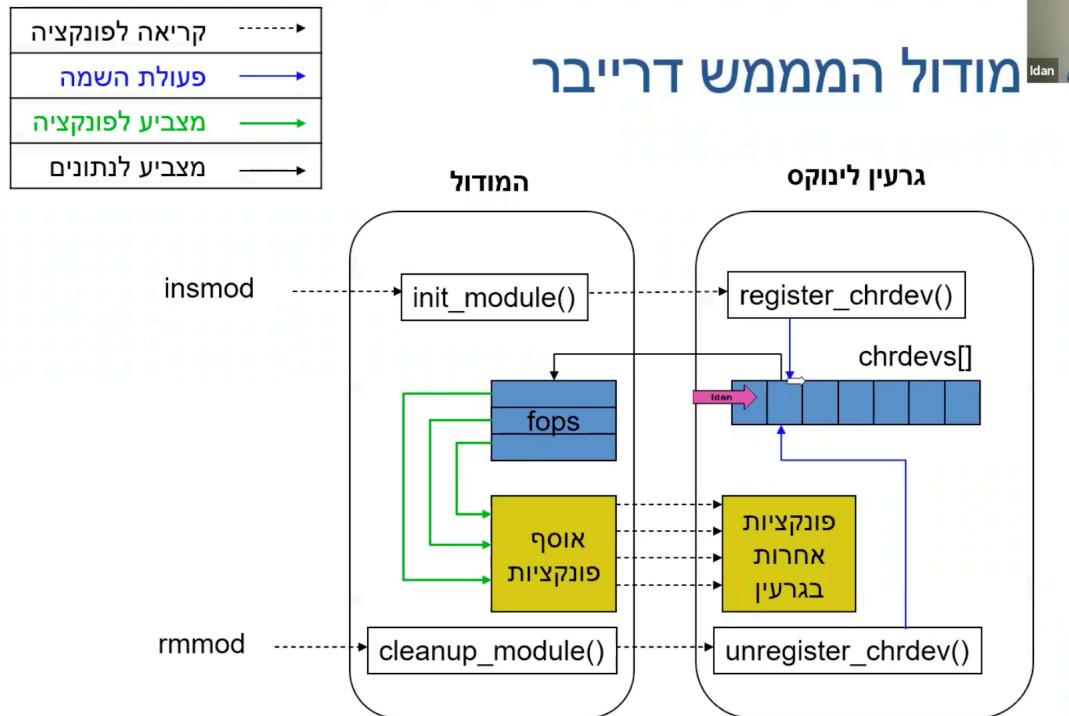
### מודול המממש דרייבר



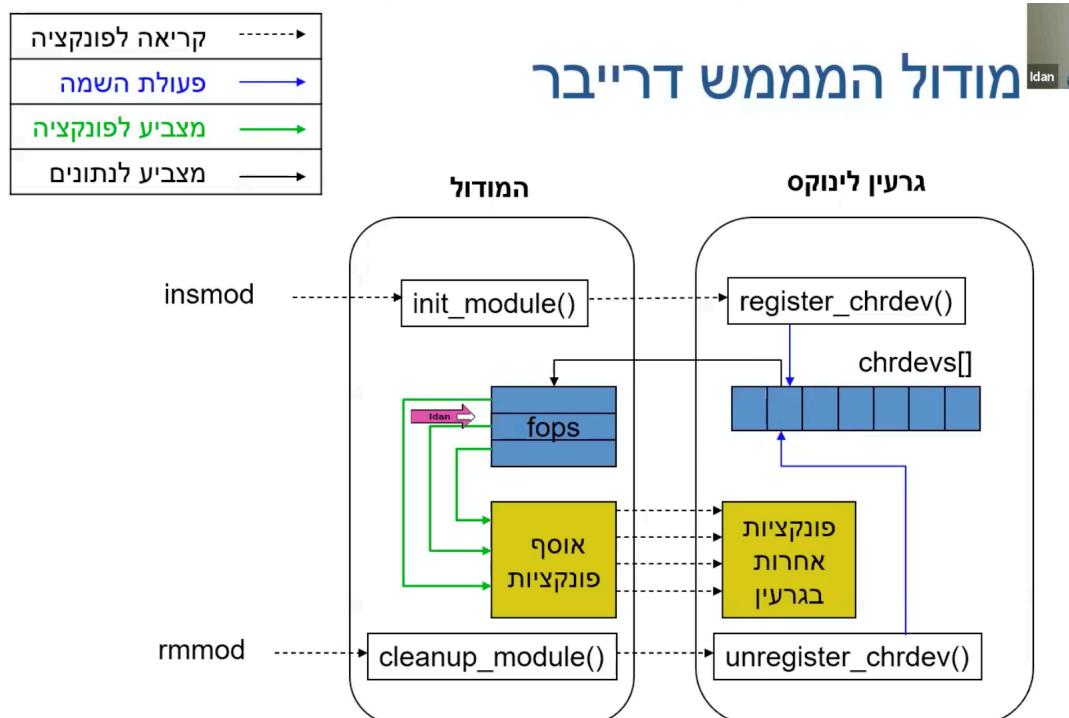
## מערכות הפעלה

77

### מודול המממש דרייבר



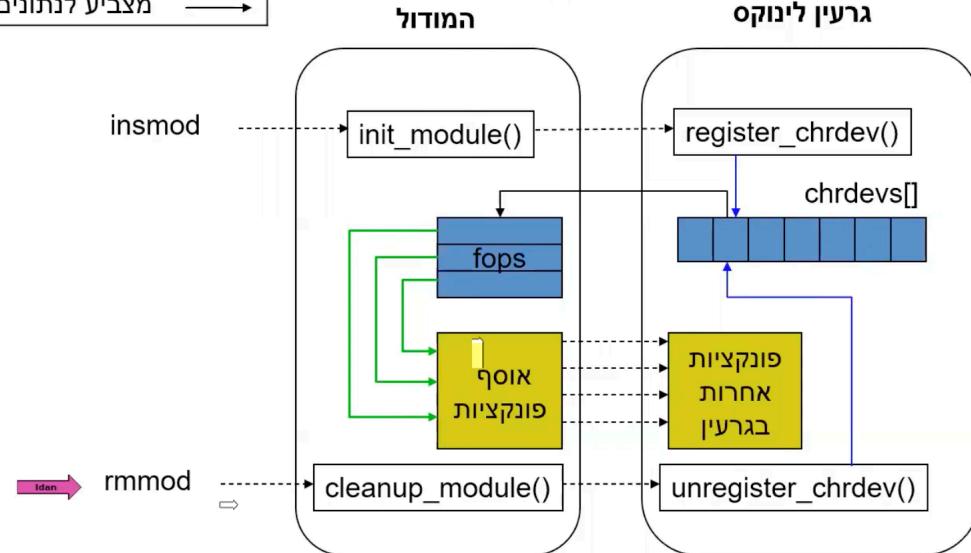
### מודול המממש דרייבר



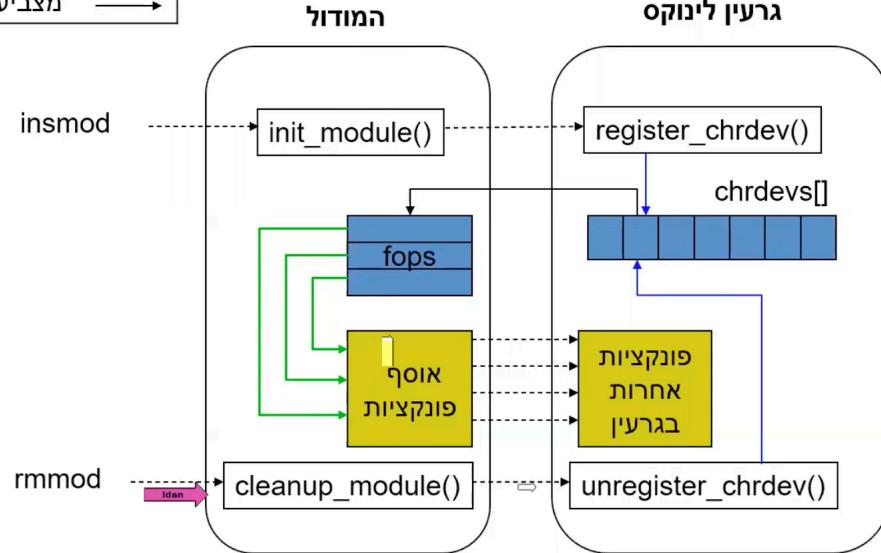
## מערכות הפעלה

78

### מודול המממש דרייבר

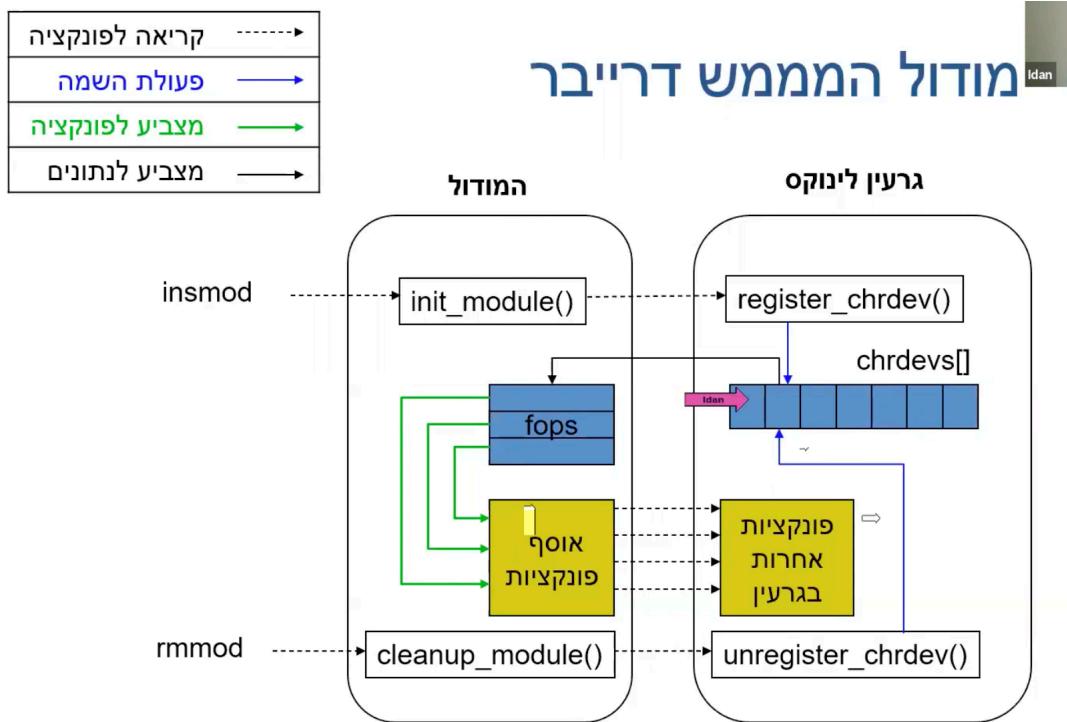


### מודול המממש דרייבר



## מערכות הפעלה

79



הסביר לתרומות למעלה. יש לנו מודול וגרעינו לינוקס לגרעין לינוקס יש מבנה נתונים ששומר את כל הדרייברים במערכת זהה המערך `chrdevs` במבנה אינדקס במערך שהוא יושב فيه. לכל דרייבר יש מספר ראשי זהה האינדקס במערך שהוא יושב فيه. בתמונה ראשונה ניתן לראות שעושים `insmod` כלומר טווענים מודול אחרי שהוא נתן הפונקציה הראשונה שמריצ' היא `init_module()` שבצム רוצה לרשום דרייבר אז מה שעושה הולכת לפונקציה שנקרהת `register_chdrev()` שהיא פונקציה של הגרעין שורשתם דרייברים על ידי חיפוש תא פניו במערך ורשותת אליו את הדרייבר. הדרייבר הוא בסך הכל מערך של מתודות שבצム הבינה `fops` שראינו מקודם שככל ... , `read`, `open`, `release` ואם נרצה למחוק את המודול מה שנעשה אז נקרא `rmmod` שתקרא `cleanup_module` מה שנוועה אז נקרא `null` `pointer` `register_chdrev` שיכולה לעрон מבני נתונים של הגרעין. אנחנו משתמשים גם ממשים את `init_module`

## מערכות הפעלה

80

בثور כתבי המודול, ואין גם בעיה ביעילות זמן כלומר לא צריך טבלת ערבות כדי להוסיף את דרייבר למערך הדריברים החיפוש יהיה ב( $O(1)$ ) כי אף פעם לא סורקים שאנו מוחזק קובץ אנחנו יודעים לבדוק מה המספר הראשי ומינורי. (נראה את זה).

רישום דרייבר חדש.

*intregister\_chrdev(unsigned int major, const char\*name, struct*

- פעולה: רישמת דרייבר חדש להתקני תווים ומקצה לו מספר ראשי.

- מוסיפה רישום שלו לקובץ *./proc/devices*.

- מוסיפה רישום שלו במערך הדריברים *.chrdevs*.

- פרמטרים:

- *major* - המספר הראשי אותו רוצים להקצותה לדרייבר.

- *name* - שם הדרייבר כפי שיופיע ב-*./proc/devices*.

- *fops* - מערך של מצביעי פונקציות, הממשים את פעולה ההתקן.

- ערך מוחזר: במקרה של הצלחה יוחזר ערך 0 או חיובי, אחרת -1.

החתימה של *register\_chrdev* מקבלת מספר מגורי ואת הערה. החתימה של *register\_chrdev* מקבלת מספר מגורי ואת השם של הדרייבר ומצביע ל-מערך *file\_operations* ובעצם כל פעם כותבת לטא המתאים במערך שאינך שלו זה *major* כموון זה מסוכן כי אין לנו בתוור בונים מודול יודעים מה לחתה לה בתוור מספר מגורי שכבר תפוס יכול להיות לך יש משהו שנקרא הקצתה דינמית של מספרים ראשיים.

הקצתה מספר ראשי.

- לינוקס תומך ב-512 מספרים ראשיים.

- חלק מהמספרים הראשיים מוקצים באופן סטטי להתקנים נפוצים.

- הקצתה סטטית של מספרים ראשיים יכולה להיות בעייתית אם שני התקנים שונים יבקשו אותו מספר ראשי.

## מערכות הפעלה

81

- ניתן וודיף לבצע הקצאה דינמית של מספר ראשי ע"י העברת ערך 0 עבור הparameter *major*. מערכת הפעלה תחפש מספר ראשי פנוי החל מ-512 ומטה.
- המספר הראשי שנבחר יהיה ערך החזרה של `register_chrdev()`.
- במקרה של הקצאה סטטית ערך החזרה יהיה 0.
- ניתן לראות את המספר שהוקצה גם ב-`/proc/devices`.

```
1 Character devices:
2 1 mem
3 2 pty
4 3 ttyp
5 4 ttys
6 5 cua
7 7 vcs
8 10 misc
9 13 input
10 14 sound
11 29 fb
12 36 netlink
13 142 pts
14 143 pts
15 162 raw
16 180 usb
17
18 Block devices:
19 1 ramdisk
20 2 fd
21 8 sd
22 9 md
23 65 sd
24 66 sd
```

## מנגנון הרישום

- קובץ `/proc/devices` נראה כך:
  - major number , driver name •
- שימושם לב שקיים גם מערך נוסף, המtauud דרייברים עבור התקני בלוקים, בדומה ל-`chrdevs`, ולכן יתכונו כפליות של מספרי `.major`.

## שאלת סיכון

(10 נק') תארו את רצף הפעולות המתבצעות במעמד קריית המערכת `open` בקוד זה:

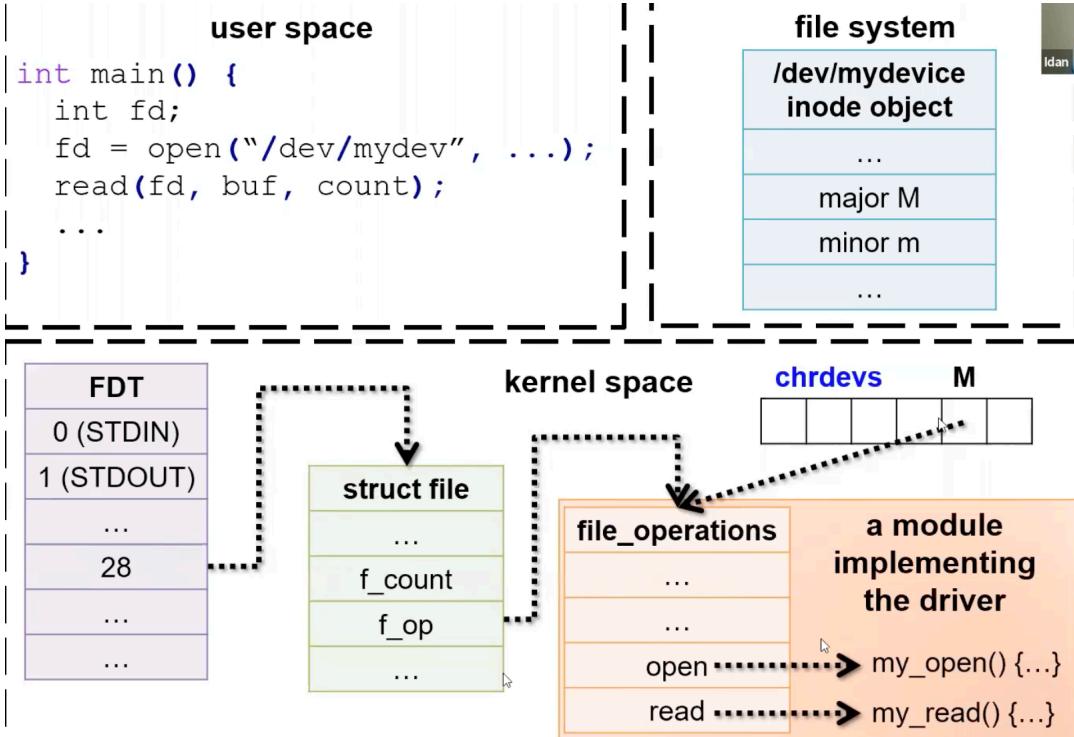
```
int main() {
 int fd;
 fd = open("/dev/mydev", ...);
 read(fd, buf, count);

}
```

מה הולך לקרות שמריצים. דבר ראשון שפותחים קובץ ניגשים למערכת הקבצים (עוד לא למדנו) ולכל קובץ יש מבנה נתוחים ייחדו שנקרא `inode` ששומר כל הפרטים על הקובץ בדיסק בפרט המספר המינורי והמגורי של הקובץ וכן שפותחים הקובץ `idev` אנחנו יודעים כבר המספר המינורי והמגורי ברגע שיש לנו המספר והמגורי אז מה שקריאת המערכת `open` יכולה לעשות זה לגשת למערך הדריירברים לתא מספר המגורי ואז להוציא ממש את ה `file_operations` (מכיל מתודות) וatz יצרת `file_object` שהיא אינקס לטבלת ה `FDT` ומתוך הכניסה ב `FDT` מצביעה ל `f_op` ומשם מתוך השדה `op` מצביע `my_open` שעובד בתוכן המודול. ואז מריםה `file_operations` שנמצאת במודול, אחר כן `read` תפעיל את `.my_read`

## מערכות הפעלה

83



הערה. אנו מניחים שהמודול כבר קיים וקוראים ממנו.

מה קורה עם המספר המינורי?

## מערכות הפעלה

84

### דוגמת קוד – מודול המממש דרייבר (1)

```
#include "my_module.h"
#define MOD_NAME "MY_MODULE"
#define OWNER "Leonid Raskin"

MODULE_LICENSE("GPL");
MODULE_AUTHOR(OWNER);

/* Params */
int iNumDevices = 0; MODULE_PARM(iNumDevices, "i");

/* Globals */
int my_major = 0; /* will hold the major number of my device driver */

struct file_operations my_fops = {
 .open= my_open,
 .release= my_release,
 .read= my_read,
 .write= my_write,
 .llseek= NULL,
 .ioctl= my_ioctl,

};

struct file_operations my_fops2 = {
 .open= my_open,
 .release= my_release,
 .read= my_read2,
 .write= my_write2,
 .llseek= NULL,
 .ioctl= my_ioctl,
 .owner= THIS_MODULE,
};
```

נשים לב שמדובר פה שני דרייברים

### דוגמת קוד – מודול המממש דרייבר (2)

```
int init_module(void) {
 my_major = register_chrdev(
 my_major, MOD_NAME, &my_fops);
 if(my_major < 0) {
 printk(KERN_WARNING
 "Bad dynamic major\n");
 return my_major;
 }
 //do_init();
 return 0;
}

void cleanup_module(void) {
 unregister_chrdev(
 my_major, MOD_NAME);
 //do_clean_up();
```

## מערכות הפעלה

85

בעצם אנו הולכים להשתמש במצור המינורי כדי להתאים דרייברים שונים ולהחליף דרייבר בזמן הטעינה. נשים לב ש יש לנו פה שתי פונקציות *init\_module*, *cleanup* הפונקציה *init\_module* רושמת את הדרייבר *my\_fops* והוא עשויה רישום דינמי של הדרייבר והפונקציה *register\_chrdev* תחזיר לתוך *my\_major* את המספר המגורי שהוקצתה אחרי כך בודקים אם הפונקציה לא נכשלה ואז מוחזירים שגיאה אחרת מחררים *my\_major*.

## דוגמת קוד – מודול המממש דרייבר (3)



```
int my_open(struct inode *inode,
 struct file *filp) {
 filp->private_data =
 allocate_private_data();
 if(filp->f_mode & FMODE_READ)
 // handle read opening
 if(filp->f_mode & FMODE_WRITE)
 // handle write opening
 if(filp->f_flags & O_NONBLOCK)
 // example of extra flag

 if(MINOR(inode->i_rdev)==2)
 filp->f_op = &my_fops2;
 return 0;
}

int my_release(struct inode
*inode,
 struct file *filp) {
 if(filp->f_mode &
FMODE_READ)
 //handle read closing
 if(filp->f_mode &
FMODE_WRITE)
 //handle write closing
 return 0;
}
```

נשים לב ש למעלה יש מתודות *my\_open*, *my\_release* שיושבות ב-*2* *my\_fops1*, *my\_fops2* בהתחלה *my\_open* עשויה כל מיני פקודות היא מקצת למשל זיכרון פרטיו של הדרייבר ואם פתחו לקריאה עשויה שהוא כתיבה שהוא אחר או עם דגלים אחרים שהוא אחץ ואז בודקת את המספר המינורי על ידי מקרו שנקרא *MINOR* שבעצם פועל על *inode* של ההתקן וسؤالת איזה מספר מינורי יש להתקין זהה אם הוא למשל 2 אז מחליפה את ה *f\_op* שיצביע על *my\_fops2* שיאושב במודול. וזאת דרך מזורה בlynokס שבה ממשים דרייברים שונים

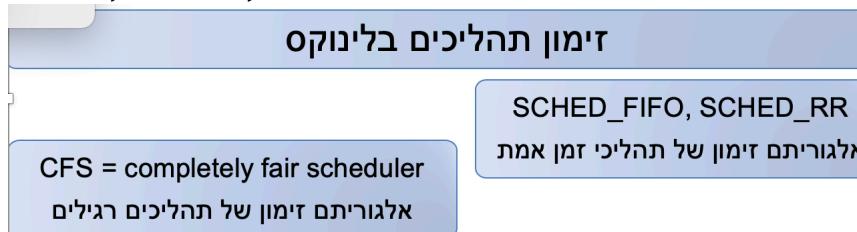
## מערכות הפעלה

86

לאותו מספרים מינוריים. תיאורטically לא אמרות לkeras ולבן דרייבר עדיף שהיה מספר מגורי משלו. נזכיר שיש רק 512 מספרים מגוריים ולפעמים יכול להיות יותר התקנים מ 512 למשל ראיינו של כל ההתקנים הפיקטיבים כולם הולכים לאותו מספר מגורי אבל יש להם מספר מינורי שונה והם לא אותו התקן אז יש להם דרישים שונים אז בעצם בשבייל לחתם להם דרישים שונים מה שהסבירנו על ידי לשמל לחתם ל file – operations את ה *my\_open* שתדרום את ה *file*.

תרגונן 5

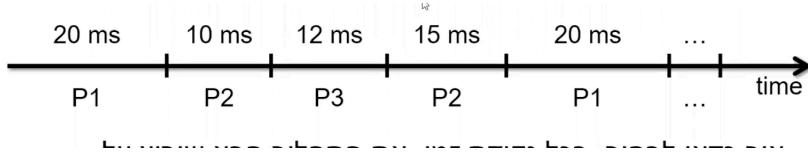
- **זמן התהליכים (scheduler)** הוא הרכיב במערכת הפעלה שאחראי על בחירת התהליך הבא שירוץ על המעבד.
- אנחנו נלמד, בהתאם לדוגמה, את אלגוריתם הזמן של לינוקס.



- אבל לפני שנלמד דוגמה "אמתית" ומורכבת, נרצה להבין את הגישות הבסיסיות בזמן תהליכי כדי לפתח אינטואיציה מבוא לזמן תהליכי. אנו יודעים שעל מעבד נתון יכול להריץ תהליכי אחד לכל היותר, אבל, גם אם יש 4 מעבדים או 8 מעבדים, ברוב המקרים מספר התהליכי יהיה יותר מ 4 או 8 אז מערכת הפעלה חייבת לחתם זמן מעבד ולחולק אותו בין התהליכים, עיקנון הירוטואלציה ראיינו בתרגול הראשון. כתע, איך מערכת הפעלה מחליטה לאיזה תהליך לחתם לרווח בפרק הזמן, متى עצרת אותו וmuevirah לתהליך הבא? ומהי פעולה החלפת הקשר לתהליך הבא וכמה זמן אם כמ הוא ירווח?
- במערכות מחשבים אמיתיים, השמייקה קצרה מדי: בכל רגע מחכים לרווח יותר תהליכי ממספר המעבדים שיכולים להריץ אותם.
- אין ברירה: מערכת הפעלה צריכה לחלק את זמן המעבדים בין התהליכים.

## מערכות הפעלה

87



- איך כדאי לבחור, בכל נקודת זמן, את התהליין הבא שירוץ על המעבד? ומהן למטה לתהליין זה?
- אין תשובה "נכונה". צריך למצוא פשרה בין מספר דרישות הנדרסיות:
  - הוננות, אינטראקטיביות ויעילות.
  - יעילות, אינטראקטיביות, והוננות.
- יעילות - תהליכיים מסתימים מהר ככל הנitin. בדרך כלל יעילות מוגדרת באמצעות מدد זמן המתנה הממוצע.
- אינטראקטיביות - תהליכיים מגיבים מהר לפעולות .  
  - לא היינו רוצים ששיחת טלפון תהיה מוקוטעת כי תהליין חיפוש קבצים רץ בركע.
  - הוננות - אין הדרה חד-משמעות, אלא יש כמה גישות.
    - למשל: תהליין שהגיע ראשון, ירוז ראשון על המעבד.
    - למשל: כל התהליכיים מקבלים את אותו זמן מעבד.
- שלושת הדרישות סותרות אחת את השניה, ולכן יש למצוא איזון ביןיהן.

אלו ממדדים חשובים ואמורים להיות אופטIMALים, יעילותן אנו רוצים שהתהליכיים שלנו יסתימו ככל שיותר מהר, רוצים שהמעבד יעבוד כמה יותר קשה להריץ תהליכיים למשל לא רוצים שהמעבד יהיה עסוק יקח זמן לחהליט מי תהליין הבא לריצה אלא יריץ תהליין הבא לריצה יזבז זמן אז זה לא יעיל. יעיל אבל עד שהוא ימצא תהליין הבא לריצה יזבז זמן אז זה לא יעיל. אינטראקטיביות זה ממד חשוב לנו כמשתמשים אנושיים וזה תהליכיים שלנו יגיבו מהר לפקודות  $O/I$ . ככלנו מכירים את הקטע שהמחשב קצת חורג ומרגיש איטי יותר מהרגיל ואז כותבים שורה ב  $WORD$  ונתקע ואז בבת אחת כותב את התווים על המסך, זאת תחושה לא טובה, אנו רוצים שהכל יהיה אינטראקטיבי שלוחצים עלתו מיד יגש למסך כךומר עברו מהר עם התקני  $O/I$  למשנה אם כרטיס רשת, עכבר, מקלדת,

## מערכות הפעלה

ss

או למשל אם אנחנו בשיחת טלפון בזומט רוצים שהזום יהיה איטי כי תהליך חישוב קבצים רץ ברקע למשל לא אכפת לנו שההתהילף הזה יסגור השהייה 5 שניות אבל 5 שניות בשיחת טלפון זה הרבה זמן. ומדד שלישי זה הוגנות, למשל כמו בחיים אמיתיים מי שmagus ראשון מקבל שירות ראשון אבל לא רוצים שהוא יתקע את כל השאר לנצח, עוד דבר של הוגנות זה שכולם יקבלו אותן זמן מעבד. לראשונה אנו מפתח אלגוריתמים שהוא ממש שונה ממה שקרה בחיים האמיתיים ונתקדם לפחות עם ההנחות כך שיתאים.

### 5 הנחות על העומס.

- נרצה לפתח אלגוריתם זימון בסיסי.
- לשם כך נניח 5 הנחות פשוטות ולא ריאלייטיות על העומס (*workload*)  
במערכת:

- (1) כל התהילכים רצים למשך אותו זמן.
- (2) כל התהילכים מגיעים באותו זמן( $t = 0$ ).
- (3) אם תהליך התחיל לירות, אז הוא יירוץ עד לסיוםו ללא הפסקות.
- (4) התהילכים משתמשים רק במעבד ולא מבצעים *O/I*.
- (5) זמן הריצה של כל התהילכים ידוע מראש.

בהמשך נסיר לפחות את ההנחות האלו כדי לפתח אלגוריתם "אמתית".

מדד ביצועים (מטריקות).

- כדי להשוות בין אלגוריתמי זימון שונים באופן כמותי, נגדיר מסוף

מדד ביצועים. המدد הראשון יהיה:

• "זמן התגובה הממוצע".

• "לכל תהליך נגדיר את "

• "זמן התגובה":

$$responseTime = terminationTime - arrivalTime$$

- מכיוון שיש הרבה תהילכים, נמצא את זמן התגובה על פוי כולם.
- כרגע, תחת ההנחות שהגדכנו, כל התהילכים מגיעים בזמן 0.

## מערכות הפעלה

89

- זמן התגובה == זמן הסיום.

אלגוריתם *FCFS*

*FCFS* = *first come, first served*.

*FIFO* = *first in, first out*:

• אופן פעולה: תור הוגן.

• לדוגמה: נניח כי 3 תהליכי *A, B, C* הגיעו בזמן 0

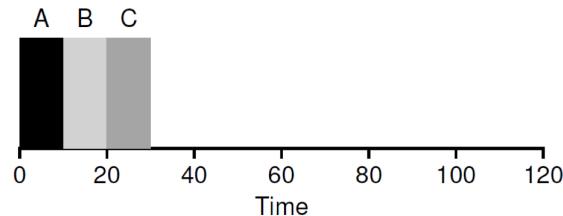
למערכת. נניח שגם *A* הגיע טיפה לפני *B*, *B* שהגיע טיפה לפני

. *C* בנוספּה נניח כי זמן הריצה של כל אחד מהתהליכים הוא 10

שניות.

- איך נראה הזמן? מה זמן התגובה הממוצע?

$$\text{averageResponseTime} = (10 + 20 + 30)/3 = 20$$



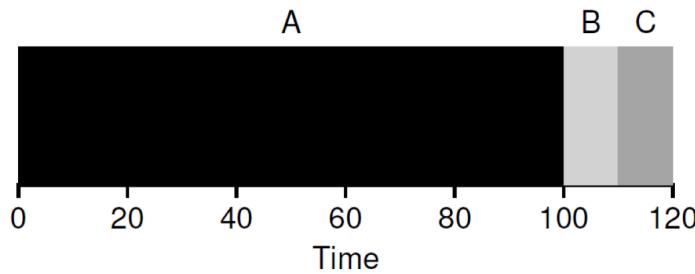
• כתע נסיר את הנחה ("כל התהליכים רצים למשך אותו זמן").

לכל תהליך זמן ריצה משלו.

• תוכלו לחשב על דוגמה שבה *FCFS* אינו יעיל?

• למשל אם נkeh A ארוך מאד שזה נקרא *convoy effect*

$$\text{averageResponseTime} = (100+110+120)/3 = 110$$



•

## מערכות הפעלה

90

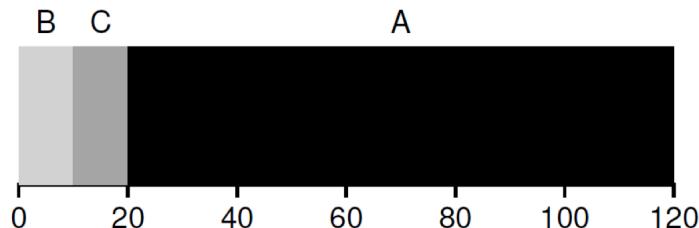
- אלגוריתם  $FCFS$  עלול לסייע מ"אפקט השיריה": מצב שבו תהליך אחד ארוך מעכבר הרבה תהליכיים קצרים. כתע עדיף לנו נגד לשים הקצרים בהתאם וכאן יש אלגוריתם זה.

אלגוריתם  $SJF$

$SJF = \text{shortest job first}$ .

- אופן פועלה: השם אומר הכל.

$$\text{averageResponseTime} = (10 + 20 + 120) / 3 = 50$$



אופטימליות של  $SJF$ .

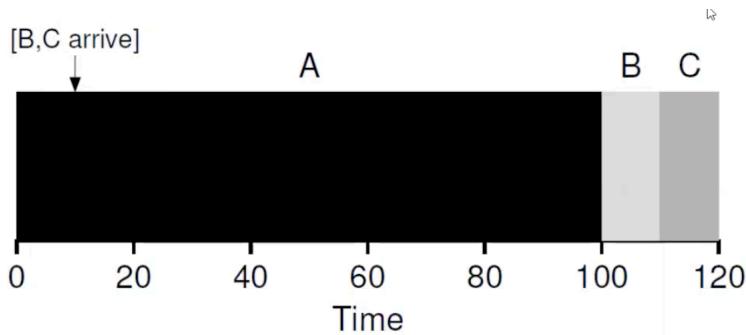
- תחת ההנחה שהגדרנו בהתחלה (פחות הנחה 1 עלייה כבר ויתרנו) ניתן להוכיח כי  $SJF$  הוא האלגוריתם האופטימלי במדד זמן תגובה ממוצע.
  - ההוכחה - בהרצאה.
- האינטואיציה מאחורי  $SJF$  היא לחת עדיפות לקלוחות == (תהליכיים קטנים כדי שייווצרו == זמן תגובה נמוך).
  - אונלוגיה ל��ופת "עד 10 פריטים" בסופר.
- כתע נסיר את הנחה 2 (כל התהליכים מגעים באותו זמן ( $t = 0$ )). תהליכיים יכולים להגיע בכל זמן  $t > 0$ .
- תוכלו לחשב על דוגמה שבה  $SJF$  אינו יעיל? שוב אפקט השיריה...

- תהליכיים  $\square$  מגע בזמן  $t = 0$  ורץ למשך 100 שניות.
- תהליכיים  $\square$ ,  $\square$  מגעים בזמן  $t = 10$  ומקשים לרוץ 10 שניות כל אחד. שוב תהליכיים קצרים מתעכבים מאחוריו תהליכי ארוך.
- 

$$\text{averageResponseTime} = (100+100+110)/3 = 103.33$$

## מערכות הפעלה

91



הנחנו ש תהליך ממשיק למשך עד שמסתיים בלי הפקעות, למשל תחילין  $t = 10$  מגע ב  $t = 10$  ו אז לא צריכים עד שמסתיים תהליך הארוך, אך  $SJF$  עדין בעיתי כי לא מתחשב בעובדה שתהליכיים שונים יכולים להגיע בזמןים שונים, הצעה לשפר את  $SJF$  זה להחליף לתחלין cocci קצר ביחס כמה שנשאר לארוך כרגע ואנו מותרים על הנחה מספר 3 שזה לעזרת תהליכיים באמצעותם.

הפתרון: הפקעה.

- כדי לפרטור את הבעיה, נסיר את הנחה 3 "(אם תהליך התחליל למשך, אז הוא ירוץ עד לסיוםו ללא הפסקות)". מערכת הפעלה יכולה להפסיק את המעבד מתחלך רצ, ככלומר: לעזרת תהליכי הנוכחי ולקראת לתחליך אחר במקומו.
- המעבר בין תהליכיים נקרא "החלפת הקשר".
- תשכורת: המנגנון שמאפשר הפקעה הוא פסיקות שעון.
- האלגוריתם שנוצר עכשוינו נקרא  $SRTF$

. $SRTF$  אלגוריתם

$SRTF = \text{shortest remaining time first}$ .

- נקרא גם:  $STCF = \text{shortest time to completion first}$ .
- אופן פועלה: כמו  $SJF$ , אבל עם הפקעות. בכל פעם שתהליכי חדש מגיע למערכת,  $SRTF$  מחשב למי מבין תהליכיים (כולל תהליכי החדש) נותר cocci פחות זמן לירוץ, ובוחר את תהליכי זהה ליריצה.

## מערכות הפעלה

92

- תחת ההנחות החדשות, ניתן להוכיח כי  $SRTF$  אופטימלי במדד זמן התגובה הממוצע.
- בתוספת הנחה כי זמן החלפת הקשר הוא אפסי.

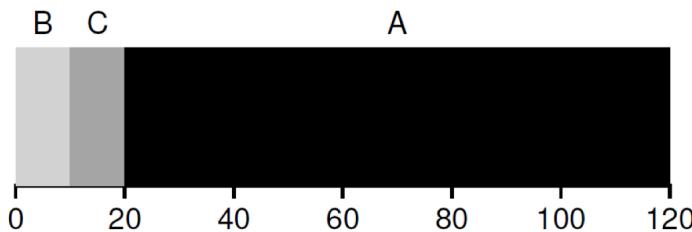
דוגמא  $SRTF$ .

- תחילה  $A$  מגיע בזמן  $0 = t$  ורץ למשך 100 שניות.
- תהליכי  $B, C$  מגיעים בזמן  $t = 10$  ומקשים ל clueן 10 שניות כל אחד.

•

$$averageResponseTime = (10 + 20 + 120) / 3 = 50$$

$$averageResponseTime = (10 + 20 + 120) / 3 = 50$$



•

- האלגוריתם הזה הוא מאד טוב מבחינה זמן המתנה אבל גרוע במדדים אחרים למשמל אם יגעו תהליכי קטנים שזמןם כולל ההוא בידוד מה שנשאר לגדלו שה  $A$  אז אנו מראיעים את  $A$  מזמן מעבד, כי תהליכי קטנים ממשיכים לבוא והוא עדין מכהה, אנו נשלב אלגוריתמים שכל אחד מהם פוטר מدد קלשו.

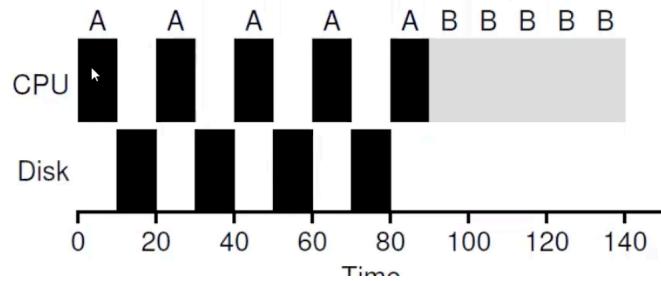
שילוב התקני  $O/I$ .

- כעת נסיר גם את הנחה 4 ("התהליכי משתמשים רק במעבד ולא מביצעים  $O/I$ ".) התהליכי נגישים לתקני  $O/I$ , לדוגמה דיסק או כרטיס רשת. גישה לתקני  $\square/\square$  אורך זמן רב (במונחי מעבד) - מספר מילישניות או יותר.
- תהליך שמתян  $\square/O/I$  לא משתמש במעבד - בעיות יעילותות.
- כעת נשים לב שכאשאר תהליכי נגישים לתקני  $O/I$  הם תוקעים את המעבד  $\square$  לא עבודה עד שmagiu המידע הנוכחי, והגישה הזאת

## מערכות הפעלה

93

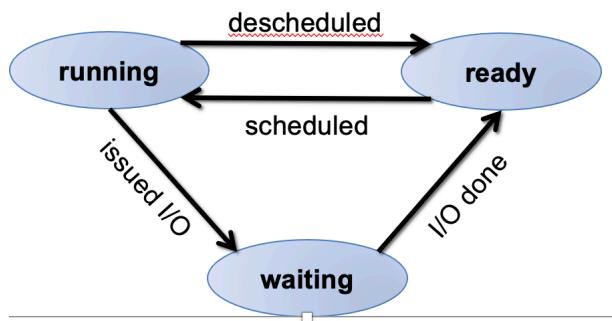
בדרכן כלל אורךת הרבה זמן בימד למעבד, שבמה יכול לבצע מליוני פקודות.



- יכולים לראות בדוגמה שעבור תהליך *A* הוא רץ לא באופן רציף כי כל 20 שניות אחרי יש גישה לדיסק, אך מה שנדמי לעשות זה לשלב את *B* בהמתנות של *A* דרך החלפות הקשר.

ניתן טוב יותר של המשאבים.

- כאשר התהליך הנוכחי מבקש *I/O* - אלגוריתם היזמן יעביר אותו למצב המתנה וזמן תהליך אחר במקומו.
- כאשר ההתקן מסיים את פועלתו ושולח פסיקה לمعالג - אלגוריתם היזמן יחזיר את התהליך למצב מוכן לריצה(לא בהכרח יריץ אותו).



דוגמא נוספת של *SRTF*.

- תהליך *A* רץ 50ms בסך הכל.
- בכל 10ms הוא מבצע *I/O* שאורך גם כנ 10ms. תהליך *B* גם רץ 50ms בסך הכל, אבל לא מבצע *I/O* בכל. הגישה המקובלת היא להתייחס לכל תחת-משימה של  כך תהליך עצמאי באורך 50ms. איך יראה זימון תחת אלגוריתם *SRTF*?

## מערכות הפעלה

94

- למעלה נסתכל על  $A$  כ 5 תהליכיים קטנים כי זה הזמן שרצץ רצוף בלי לגשת להתקני  $O/I$  ואז אלגוריתם  $SRTF$  הוא רואה שהמקטע  $A$  יש  $10ms$  שניות לרווח אלו  $B$  יש לו  $50ms$  אך מי שרצץ זה מקטע של  $A$  אחר כך, המעבד פנוו אז אין ליט בירור החוץ מלהרין את  $B$  אחרי שמסתיימת הגישה לדיסק שוב יש בדיקה למקטע של  $A$  יש  $10ms$  לרווח אלו של  $B$  יש  $40ms$  אך שוב יבחר  $A$  והלאה. השים לב שאנו מנהיכם שאלגוריתם יידע את זמן הריצה של כל תהליך או מקטע או גישה מראש זהה בעיתוי, ותכסף נסיר אותה.

### :Batch scheduling

- אלגוריתמי *batch scheduling* מניחים את הנחות 3,4,5 שראינו:
  - אם תהליך התחל לרווח, אז הוא ירוץ עד לשינויו ללא הפסוקות.
  - התהליכיים משתמשים רק במעבד ולא מבצעים  $O/I$ .
  - זמן הריצה של כל התהליכיים ידוע מראש.
- שימו לב: אנחנו הנחנו עבודות סדרתיות, אבל בהרצאה מטפחים גם בעבודות מקביליות == (רצות על מספר ליבות במקביל). רוב התוצאות התיאורטיות כבר לא תקפות לעבודות מקביליות.
  - למשל,  $SJF$  כבר לא אופטימלי במדד זמן המתנה ממוצע.
- אבל האינטואיציה שקיבלו בעבודות סדרתיות בדרך כלל נשמרת.

נשארנו רק עם הנחה ...

- "זמן הריצה של כל התהליכיים ידוע מראש".

*batch scheduling* במקורה של כפי שלמדתם בהרצאה:

- הרבה משתמשים עובדים על מחשב-על (*supercomputer*)
  - בעל הרבה ליבות.
  - המשתמשים שלוחים "עבודות" (*jobs*) לתוך הריצה.
  - אלגוריתם הזמן משבע את העבודות על הליבות הפנויות.
  - המשתמשים נדרשים לספק הערכה לזמן הריצה של העבודות שם שלוחים.

## מערכות הפעלה

95

- עבודה שchorget זמן הריצה שהוגדר לה - נוצרת ע"י אלגוריתם

### הזמן

- נניח שאנו בתור פיזיקאים שעובדים במאיצ' החלקיים בשווין ומקשים ממנו לחת הערכה לזמן הסימולציה ואם ניתן הערכה קטנה מדי פשוט ירגנו לנו העבודה וכן למה לחת הערכה מאוד גבוהה שתסתהים בצורה תקינה? זה יהיה תיעודן נמוך, התשובה היא שזה תיעודן נמוך, למדנו שרוב אלגוריתמי זמן של *batch* תהליכיים קצרים, אומנם לא ראיינו אלגוריתמי זמן של *backfill* אז בעצם ניתן עדיפות לתחליכים אבל בהרצאה ראיינו *backfill* אז מכך ניתן עדיפות לתחליכים קצרים, אז בתור פיזיקאי נרצה למצוא פשרה מצד אחד נרצה למצוא זמן הערכה נמוכה לזמן הריצה כדי לקבל עדיפות גבוהה (תחליכים קצרים מקבל עדיפות גבוהה יותר) מצד שני לחת הערכה גבוהה יותר שלא יקתוו באמצעותו. (כל התרגול הזה מניחים שתתהליכיים רצים על ליבת אחת).

מדד חדש: זמן המתנה.

- עד כה למדנו מספר אלגוריתמי זמן והשוינו ביניהם לפי מדד "זמן המתגובה":

$$responseTime = terminationTime - arrivalTime$$

- כתעת נציג מדד חדש - "זמן המתנה":

$$waitTime = startTime - arrivalTime$$

- כמו קודם, המדד יהיה זמן המתנה הממוצע על פני כל התחליכים.
- איזה מדד קובע את הביצועים?
- התשובה תלוי בעומס ...

איזה מדד יותר חשוב מבין אלה? *waitTime* או *responseTime*?  
שוב זה תלוי בתחום *BOUND I/O* שהם מעוניינים בזמן המתנה נמוך, סרט וידאו למשל זה לא סרט שרע רוב

## מערכות הפעלה

96

הזמן אלא העין האנושית, אם אנחנו מציגים שלה 60 תמונות בשניה זה ייראה לה כמו סרט או וידיאו, אך מה שנגן סרטים כזה האמור לעשות זה שכל  $15ms$  שניות ישם *frame* חדש לעיניים של המשתמש. למחשב זה פעולה די זולה, הוא צריך לתהווור פעם כל  $15ms$  ישם מעוניין זה *frame* חדש ב  $1ms$  ואז יחזיר לשישון וכן אלה, נגיד לתהלייף כזה מעוניין בזמן המתנה נמוך ולא זמן תגובה נמוך.

נהוג לסווג תהליכיים לשני סוגים. תהליך חישובי *CPU Bound*

- מעוניין בזמן תגובה נמוך. *throughput sensitive*. דוגמה:  
סקריפט *python* שמנתח נתונים ע"י חישובים אלגבריים.
- בדרך כלל לא מיותר על המעבד מרצונו אלא מופקע.

תהליך אינטראקטיבי *I/O Bound*

- מעוניין בזמן המתנה נמוך.  
*latency sensitive* -  
דוגמה: גן סרטים שמחלייף 60 פריטים בשניה. (הוא מאוד מהיר ואז הולך לשישון ככלומר מיותר על זמן המעבד).
- בדרך כלל מיותר על המעבד מרצונו אחרי פרק זמן קצר ברגע המתנה לפעולות *I/O*.

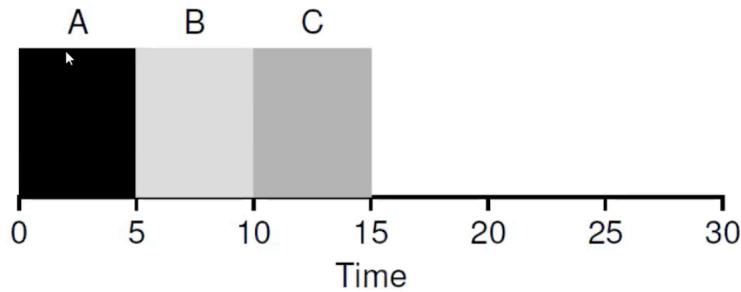
*SJF/SRTF* לא מצטיינים בזמן המתנה.

- גנית שלושה תהליכיים  $A, B, C$  מגיעים באותו הזמן.
- כל תהליך רץ למשך 5 שניות.
- מה יהיה זמן המתנה תחת אלגוריתם *SRTF* או *SJF* ?

$$averageWaitTime = (0 + 5 + 10)/3 = 5$$

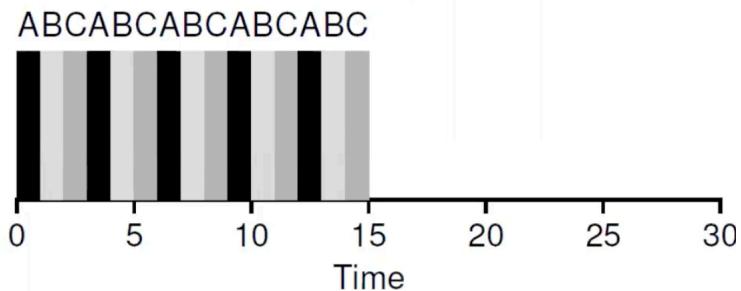
## מערכות הפעלה

97



### אלגוריתם *Round Robin*

- במוקום להריצת תהליכיים עד לסיוםם, האלגוריתם מריץ כל תהליך במשך פיסת זמן מסוימת (*timeslice*) או (*quantum*) או (*averageWaitTime*) ... עבור להריצת תהליך אחר למשך אותה פיסת זמן, וכן הלאה ...
- אם נניח כי הקוונטום הוא  $1s$ , אז:  $\approx 2$  .  $\square$  . לצורך חישוב זמן התגובה, נתיחס לכך פיסת זמן קצר תהליכי עצמאי באורך  $1s$  אשר מגע ברגע שפיסת הזמן הקודמת הסתיימה.
- למשל בדוגמה ש  $quantom = 1$
- נשים לב שהוא לא טוב לתהליכיים חישוביים, נשים לב שזמן סיום המוצע זה די גורע לעומת  $SJF$ .



- שוחרים  $quantom$  הוא קובע את זמן ההמתנה ואם הוא נמוך מזמן ההמתנה נמוך, אבל אם הוא עמוק מדי, אז זמן של החלפת ההקשר נהיה משמעותי. (כלומר יש תקורה).
- שיקולים בבחירה הקוונטום.
- הקוונטום חייב להיות כפולה של הזמן בין פסיקות שעון.

## מערכות הפעלה

98

- למשל, אם פסיקות שעון מגיעה כל  $10ms$ , אז הקוונטום יכול להיות  $10ms, 20ms, 30ms$  ...
- למה עדיף קוונטום נמוך?
  - זמן המתנה נמוך יותר  $\leftarrow$  אינטראקטיביות.
- למה עדיף קוונטום גבוה?
  - פחות החלפות הקשר  $\leftarrow$  ביצועים טובים יותר.
- למשל נניח כי הקוונטום הוא  $10ms$  וזמן החלפת הקשר הוא  $1ms$ . מה התקורה של אלגוריתם  $RR$  במקרה זה?

זמן תהליכיים בלינוקס. שני סוגי תהליכיים בלינוקס:

תהליכי זמן-אמת (*real – time*).

- נדרש לעמוד באילוצים קשיים על זמן התגובה, ללא תלות בעומס על המערכת.
  - דוגמה: תוכנת הטיס האוטומטי במטוסים.
- רק משתמש-על (*root*) יכול להגדיר תהליך זמן-אמת ע"י שינוי העדיפויות שלו.

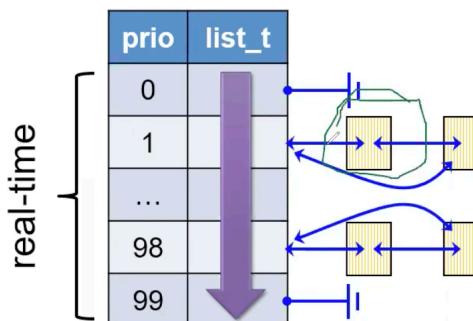
תהליכי זמן אמת: נניח שבמיטוס מותקנת מערכת הפעלה לינוקס, כי היא מריצה עליה את התהליך של הטיס האוטומטי שמחילה על הנחיתה והלאה, נניח אם מטוס שרוכבים עליו מרץ לינוקס, לא היינו רוצחים בזמן תהליכי הנחיתה מתרחש פתאום יצוץ עדכון של לינוקס, ומציג את התהליך של הנחיתה צידה ורץ במקומו, יש תהליכיים קריטיים בטילים מערכות צבא, שسور שמיشهו ייז אונצ צידה, תהליכיים אלה הם תהליכי זמן אמת כי הם צריכים שביצועים שלהם יהיו בזמן אמיתי, והם ה/cgi עדפים במערכת. תהליכיים רגילים הם כל התהליכיים שם לא זמן אמת, כל מה שמכירים למשל *PPT* או דף דין אז זום, וגם תהליכי שמנתח מידע בזמן טישה לאחרי טישה. אבות מייסדים הלינוקס אי שם בשנות ה 90 היו נאים חשבו שהייה להם מערכת הפעלה שתறוץ על כל דבר, מפליאפון עד מטוסים, לווינים מחשבים אישיים, בפועל היום אחרי 20 שנה הבינו שהוא לא עובד, יש היום מערכות צבאיות לווינים,

## מערכות הפעלה

שיטוטים וכל מה שצריך לעבוד בזמן אמיתי בדרך כלל יש לו מערכות הפעלה ייעודיות לתחביבי זמן אמיתי וה*feature* לכך של לינוקס לא ממש שימושי, אבל נלמד עליה בכל מקרה למרות שימושים רגילים לא משתמשים בתהביבי זמן אמיתי.

אלגוריתם הזמן של תהביבי זמן-אמת.

- כל התהביבים המוכנים לריצה(מצב *TASK\_RUNNING*) נשמרים בטור הריצה: מערך תורים באורך 100, טור לכל עדיפות מספרית. כל תהביב נמצא בטור אחד בלבד.
- התהביבים מזומנים לפי עדיפות - תהביבים בעדיפויות נמוכה יותר רק אחרי שהסתינו כל התהביבים בעדיפויות הגובה.
  - מספר גובה == עדיפות נמוכה.
- האלגוריתם תמיד בוחן ריצה את התהביב שנמצא בראש הטור עם העדיפויות הגובה ביותר. (מסומן בירוק).



מדיניות זמן של תהביבים. האם יש הפקעות בין תהביבי זמן אמיתי? זה תלוי במידיניות?

- בכלל תהביבים זמן-אמת יש מדיניות זמן (*scheduling policy*) *SCHED\_FIFO* (או *SCHED\_RR*) או *(sched\_setscheduler)*.
- נקבעת ע"י המשמש באמצעות קריאות מערכות.
- מדיניות זמן - תשפיע על כמה זמן ריצה כל תהביב יקבל ואופן העבודה הטור.

## מערכות הפעלה

100

### *SCHED\_FIFO*

- זימון  $\leq$  סדר הגעה. תהליך  $FIFO$  מועתר על המעבד רק אם:
  - הוא יצא להמתנה (למשל בגל  $O/I$ ) - בעtid יחזור  $\leq$  סוף התוור.
  - הוא קורא לקריאת המערכת  $ched\_yield$  - עובר מיד  $\leq$  סוף התוור (נשאר בתוור הריצה).
- תהליך  $FIFO$  מופקע רק ע"י תהליך זמן-אמת אחר עדיף יותר.

### *SCHED\_RR*

- חלוקת זמן בין כל התהליכים המוכנים לריצה בעלי העדיפות הטובה ביותר.
- כל תהליך מקבל קוונטום (פיסת זמן) בתورو,  $\leq$  סדר מעגלי.
- שימוש: תהליכיים במדיניות זמן  $SCHED_RR$  עלולים "להיתקע" בתור אחרי תהליכיים במדיניות זימון  $SCHED_FIFO$ .
- תהליכי  $FIFO$  יכולים להרעיב תהליכי  $RR$  כי הם לא מועתרים על המעבד אף פעם

| תהליך    | A  | B  | C  | D  |
|----------|----|----|----|----|
| זמן הגעה | 1  | 2  | 2  | 3  |
| עדיפות   | 30 | 31 | 30 | 30 |

דוגמא.

- אם כל התהליכים במדיניות  $SCHED_FIFO$ :
- אז תהליך ראשון שירוץ זה A עד שהוא מסתיים או יותר על B המעבד אמר כל C אחר כן D ורק אם A, C, D ירוץ כי הוא עם עדיפות הכי נמוכה.

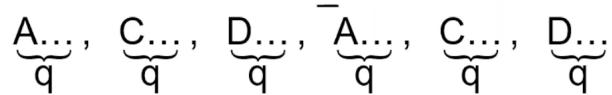
$\underbrace{A\dots}_{\text{until finished}}, \underbrace{C\dots}_{\text{until finished}}, \underbrace{D\dots}_{\text{until finished}}, B\dots$

## מערכות הפעלה

101

- אם כל התהליכים במדיניות  $SCHED\_RR$ :

- תהליך  $B$  לא ירוץ, ירוץ רק תהליכים עם עדיפות הכי גבוהה



- ללא קשר למדיניות הזמן, תהליך  $B$  ירוץ רק כאשר שלושת התהליכים  $A, C, D$  יסתינו ואו לא יהיו מוכנים ליריצה.

מה קורה אם  $FIFO$   $RR A, C, D$  לシリוגן אז במקרה זה אם  $A$  היה אז  $RR$  אז הוא רץ את ה *quantum* ומיותר ואז אם היה  $FIFO$  הוא רץ עד הסיום, אז זה קצר מוחר אם מירוץ  $RR$  ירוץ  $B$  וייתר על המעבד ואם  $FIFO$  הוא מירוץ ולא יותר. מה היה אם בעדיפות גבוהה יותר למשל 29? הוא היה רץ על הזמן עם שמשתים ומיותר על המעבד. האם זה לא הונע מעורב מי שמתmesh ב  $RR$ ? כן, אבל כל תהליכים של זמן מערכות זמן אמת מתאימים למערכות מאוד ספציפיות, למשל טיל של משימה להקל ומערכות כאלה משתמשים חייבים להיות אחרים לkneg כל תהליך בעדיפות הכי מתאימה לו שמערכת תעבור חלק. זה אלגוריתם שלא סומך על מערכת הפעלה שתעשה משהו חכם בשביבו אלא על משתמש שיגיד לו בדיק מה לעשות.

הוספת תהליכים לתור הריצה.

- תהליכי חדשים ותהליכים שחררו מהמתנה יכנסו לסופו התור המתאים

לעדיפותם.

• שאלה: מה היתרונות בלהכניס לסופו התור לעומת תחילת התור?

- אם המערכת לא עמוסה (כלומר מעט תהליכים) - זה כמובן לא משנה.

• תשובה:

- הוספת תהליכים בסוף התור שומרת על הוגנות כי תהליכי

שהגיעו קודם יריצו קודם. (כמו בחיים אמיתיים מי שmagus

ראשון מקל לפני)

## מערכות הפעלה

102

- הוספת תהליכיים בתחילת התור הייתה עלולה ליצור הרעבה אם תהליכיים חדשים ממשיכים להגיע ותהליכיים בסוף התור לא זוכים לרווח.
- תמיד מリストים קודם תהליכיים עם עדיפות ביותר, וחושב לדעת שימוש בוחר עדיפות, ואם נרצה שתפקיד תמיד יפעל אז ניתן לו עדיפות 0 שהוא הכי גבוה שיש..

חישוב ה-*time slice*. פסיק קצר שלא דיברנו עליו זה אין קבועים את ה *quantum*, כל פסיקת שעון אנחנו מורידים את ב *quantom* זהה אם למשל יש לנו 1, המקרו בקוד שגדים את ה *quantom* הזה אם למשל יש לנו 50 פסיקות שעון אז ה *HZ* הוא 50 אז למשל חצי שנייה זה  $\frac{1}{2} HZ$  לפי ההיגיון הזה ה ברירת מחדל של *quantom* זה  $\frac{1}{10} HZ$  קלומר מספר פסיקות שעון שיש לנו בעששית השנייה. נשים לב שה *quantom* הוא כפולה של פסיקת שעון. נשים לב שה *quantom* זה פרק הזמן להחלה תהליכיים, החלה יכולת להתרחש רק שמערכת הפעלה מקבלת שליטה בידה וזה קורה רק במרחב זמן קבועים זהה כל פסיקת שעון. (זה לא יכול לקרות במרחב זמן שהוא לא פסיקת שעון). דבר נוסף, ה *quantom* יתאפס כי ככל תהליך שומרים משתנה שהוא ה *quantom* וכל פסיקה שעון מורידים את ה *quantom* את ה *quantom* של המשתנה של התהליך הנוכחי שרך, ברגע שמדובר 0 או מבנים שתפקיד סיים את ה *TIME SLICE* שלו ונוטנים לו עוד פעם שתפקיד סיים את ה *TIME SLICE* מחדש. (ה *TIME SLICE* בתחלת זהה עברו על התהליכיים).

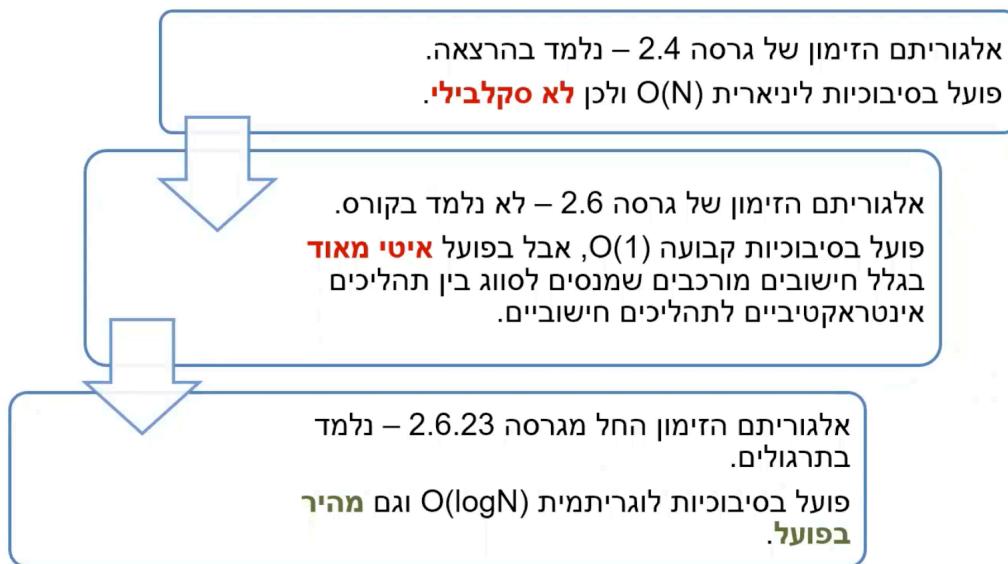
- כאמור, ככל תהליך במדיניות *SCHED\_RR* מוקצב פרק זמן לשימוש במעבד - *time slice* או *quantum*.
- ה-*time slice* מוגדר ביחידות של פסיקות שעון. בכל פסיקת שעון ערכו קטן ב-1. כאשר הוא מגיע ל-0, התהליך סיים את הזמן שהוקצב לו.
- *TASK\_TIMESLICE* מוקצתה לתפקיד *time slice*.

## מערכות הפעלה

103

- החישוב עובר מיחידות של מילוי-שניות ליחידות של מספר פסיקות שעון.
- $HZ$  = מספר פסיקות השעון בשניה.

*defineRR\_TIMESLICE(100 \* HZ/1000)*



תהליכיים בלינוקס.

בהרצאה רأינו אלגוריתם **لينكس** הhei ישן והוא מאד פשוט, והוא נראה  $(O(N))$  כאשר מספר התהליכים זה  $N$  כי עובד בסיבוכיות זמן לינארית והבעיה אליו שהוא לא סקלריאלי עם השנים מספר התהליכים גדול ומה שקרה שאלגוריתמים בסיבוכיות לינארית נהיו איטיים מדי. בהרצאה גם רأינו איך הוא אלגוריתם שגדיר *nice* לכל תהליך ועם *goodness* פקטורי, וכל פעם סורקים את כל המערכת של התהליכים מחשב לכל אחד את הפרמטר *goodness* ובסוף בוחר התהליך הכי טוב, אחר כך פיתחו אלגוריתם זמן שעבוד ב- $(O(1))$  נesson על הניר זה רשום  $(O(1))$  אבל בפועל הוא מאד איטי, כי למשל נניח ש 10000000 הוא גם קבוע, כי הוא ניסה לעשות כל מיני חישובים מסוימים להבין

## מערכות הפעלה

104

איזה תהליכיים חישוביים או אינטראקטיביים, ואז פיתחו אלגוריתם זימון  
שנלמד ורץ ב  $O(\log n)$ .

*Completely Fair Scheduler (CFS)*

- אלגוריתם זימון של גרעין לינוקס החל מגרסת 2.6.23 הוא  
פותח כדי להשיג:

-יעילות - האלגוריתם מבזבז מעט זמן על קבלת החלטות.

- מדריגות (utility) - הביצועים מדדרים בצורה מתונה  
יחסית כאשר מספר התהליכים גדול.

- במקומות לנוסות להקטין את הזמן המתגובה או זמן ההמתנה, *CFS*  
פשוט מנסה להיות הוגן ולתת לכל תהליך נתח שווה של זמן  
המעבד.

- עד כדי עדיפות: תהליכיים בעדיפות גבוהה יותר יקבלו נתח  
גדול יותר.

- לצורך פועלתו, אלגוריתם *CFS* מגדר מושג חדש: זמן ריצה  
וירטואלי.

- הרחבבה של אלגוריתם *RR*: קווונאים משתנים באופן דינמי בהתאם  
למספר התהליכים במערכת.

דוגמת הרצה:

- נניח כי במערכת יש ארבעה תהליכים  $A, B, C, D$ .

• האלגוריתם קובע זמן *epoch* שבמהלכו ירצו

$$sched\_latency = 48ms$$

- אם יש  $N$  תהליכים במערכת ועולם באותה עדיפות, הקווונטום של  
כל תהליך יהיה:

$$Q_i = sched\_latency / N$$

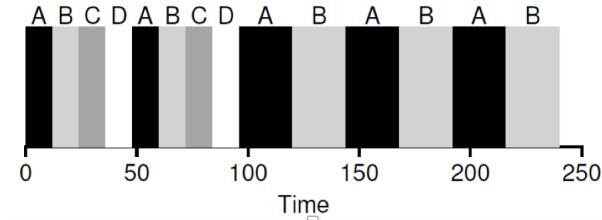
- אז הקווונטום של כל תהליך יהיה  $12ms$ .
- כתעת נניח כי לאחר שני *epochs* התהליכים  $C, D$  מסתיימים.

## מערכות הפעלה

105

- אז התהליים הנתרים  $A, B$ , ממשיכים לרוֹץ עם קוונטום של  $.24ms$

- הזמן של  $CFS$  נראה כמו  $RR$  עם קוונטום דינמי:



•

זמן ריצה וירטואלי ( $vruntime$ )

- כאשר תהליך רץ, הוא צובר לעצמו זמן ריצה וירטואלי
- באופן אידיאלי, זמן הריצה הוירטואלי שווה בין כל התהליים בכל נקודת זמן.
- אבל באופן מעשי, רק תהליך אחד יכול לרוֹץ על המעבד בכל רגע נתון, ולכן יהיו הפרשים בין התהליים שונים.
- כאשר  $CFS$  מזמן תהליך לריצה הוא יבחר את התהליים בעלי זמן הריצה הוירטואלי הנמוך ביותר (כדי להיות הוגן).
- האלגוריתם שומר בכל רגע את המקסימום והמינימום של זמן הריצה הוירטואלי על-פני כל התהליים המוכנים לריצה – מיד נבין למה.
- יהיו פערים בין זמנים ריצה וירטואלי, מה ש  $CFS$  יעשה יבחר את התהלייף עם הזמן הוירטואלי הכי נמוך, ומה שהוגן זה למתן לו לרוֹץ עכשו. נראה שעדריפות משנות את היחס בין זמן ריצה וירטואלי לאמיתי.

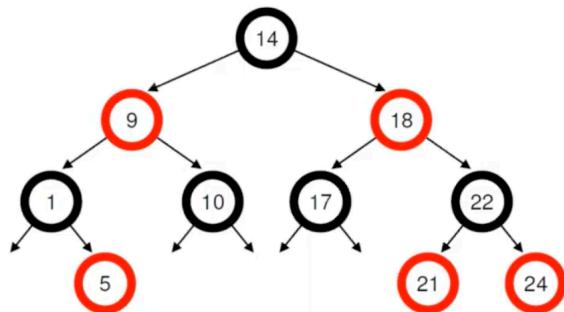
מבנה הנתונים של  $CFS$ :

- עץ אדום-שחור – עץ חיפוש בינהי מאוזן עם סיבוכיות חיפוש, הכנסה, ומחיקה לוגריתמיות.
- $O(\log N)$  כאשר  $N$  הוא מספר התהליים.
- מפתח החיפוש בעץ הוא זמן ריצה הוירטואלי ( $vruntime$ ).
- $vruntime = max\_vruntime$

## מערכות הפעלה

106

- אלגוריתם  $CFS$  בוחר תחילה הבא לדרישה הכי שמאלי ביותר. כלומר מקבל החלטה ב- $O(\log n)$ .
- נשים לב שלתחיל חדש אוטחל עם המקסימום בין כל זמני התהליכים שנמצא כרגע, כי אמנו שרוצים שהוא יהיה בסוף התור וב- $CFS$  השוקל לחתת לו את ה  $max\_vruntime$ . (אפיו שזה יכול להריעיב אותם).



מה קורה במערכת עמוסה? אם מספר התהליכים  $N$  במערכת גבוהה, אז ה *quantum* קצר מדי וatz יש תקורה גבוהה של החילפת הקשור, המערכת עלולה לסייע מהחולפות הקשורות ופגיעה בביצועים. לכן מוגדר גם זמן מינימום על הקוונטום:

$$Q_i \geq min\_granularity = 6ms$$

לדוגמא, אם יש 10 תהליכיים במערכת, אז הקוונטום של כל אחד אמרות להיות:

$$Q_i = 48ms / 10 = 4.8ms$$

בפועל, כל תהליך קיבל  $6ms$  וכך משך הסיבוב שבו כל התהליכים ירצו יהיה:

$$epoch = 60ms \geq sched\_latency$$

זה הכוונה דאמנו כי ה  $epoch$  דינמו כי  $RR$  מתחרג.

## מערכות הפעלה

107

יציאה וחרזה מהמתנה.

- רק תהליכיים מוכנים ליריצה נשמרים בעץ.
- תהליכיים שמבצעים  $O/I$  יוצאים מהעץ וועברים לתור המתנה.
  - **תרחיש בעיתתי:**
  - שני תהליכיים  $A, B$ , רצים זה לצד זה.
  - תהליך  $B$  יוצא להמתנה ארוכה של 10 שניות.
  - כאשר  $B$  מתעורר, הוא נכנס לעצם  $vruntime$  קטן ב-10 שניות מזו של  $A$ . לפיכך, תהליך  $B$  יהיה זה שירוץ ב-10 השניות הבאות. הרעבה של תהליך  $A$ .
  - כדי למנוע את התרחיש הבועתי הנ"ל, מוסף תהליכיים שחזרו מהמתנה ארוכה עם  $vruntime = \min_vruntimes$ .  
כדי להיות הוגנים בדרך כלל לאלו שיוצאים הם תהליכיים אינטראקטיביים שקצרים ויצאו כי רצו בתור המתנה עד שיקבלו מידע מרשת או התקני  $O/I$  וכן שחוור רתים לתוכה לו לרווח אבל לא רוצים שירוץ ותקכו את האחרים.

עדיפות.

- $CFS$  מאפשר למשתמש להגדיר עדיפויות לתחנכים וכן לחזק את זמן המעבד בצורה שונה בין התחנכים.
- העדיפות של התהליך מיוצגת ע"י הערך  $20 \leq nice \leq +19$ .

- ברירת המחדל היא  $0.nice = 0$  -

- תהליך "נحمد" יותר יהיה בעדיפות נמוכה יותר.

```
static const int prio_to_weight[40] = {
 /* -20 */ 88761, 71755, 56483, 46273, 36291,
 /* -15 */ 29154, 23254, 18705, 14949, 11916,
 /* -10 */ 9548, 7620, 6100, 4904, 3906,
 /* -5 */ 3121, 2501, 1991, 1586, 1277,
 /* 0 */ 1024, 820, 655, 526, 423,
 /* 5 */ 335, 272, 215, 172, 137,
 /* 10 */ 110, 87, 70, 56, 45,
 /* 15 */ 36, 29, 23, 18, 15,
};
```

## מערכות הפעלה

108

- בlynokס להיות נחמד זה לא טוב כי תהליכיים כאלה מקבלים פחות זמן ל线索. קצת נוסחות.

- נניח שיש במערכת  $n$  תהליכים עם עדיפות:  $P_1, P_2, \dots, P_n$  ומשקלים:  $W_1, W_2, \dots, W_n$ . נניח כי  $W_0$  הוא המשקל המתאים לעדיפות  $0 = nice$ .
- אז זמן הריצה הוירטואלי של התהליך  $i$ -י מתקדם לפי:
- 

$$VRi+ = (W_0/W_i) \cdot \Delta T$$

כאשר  $\Delta T$  הוא זמן הריצה לפי שעון אמיתי. זמן הריצה הוירטואלי זהה לזמן הריצה האמתי עבור ברירת המחדל.  $0 = nice$  ניתן להוכיח כי הקוונטום של התהליך  $i$ -י הוא:

$$Qi = (Wi/\sum Wi) \cdot sched\_latency$$

דוגמת חישוב.

- נניח כי יש שני תהליכים:
- $P_1$  עם ערך 5 משקל  $W_1 = 3121$   $0 = nice$ .
- $P_2$  עם ערך 0 משקל  $W_2 = 1024$   $0 = nice$ .
- נקבע כי תהליך  $P_2$  מתקדם (בערך) פי 3 יותר מהר מהתהליך  $P_1$ :

$$VR1+ = 1/3 \cdot \Delta T$$

$$VR2+ = \Delta T$$

ואכן:

$$Q1 = \frac{3}{4} \cdot sched\_latency$$

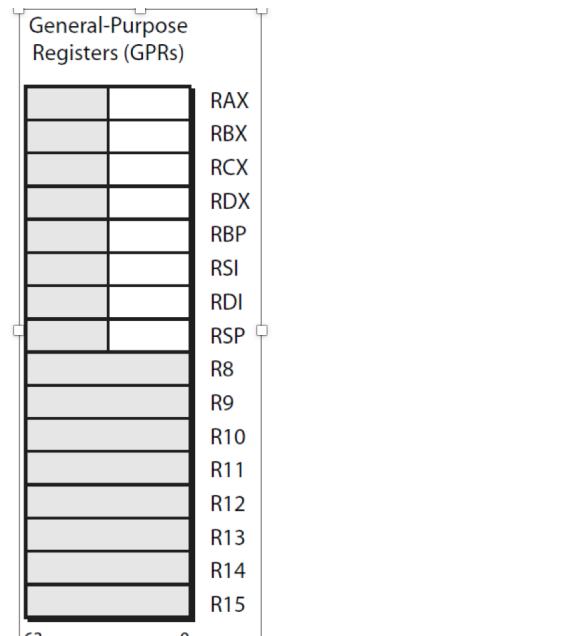
## מערכות הפעלה

109

$$Q2 = \frac{1}{4} \cdot sched\_latency$$

תרגול 6

תקציר. החלפת הקשר? מימוש החלפת הקשר בLinux. יצירת תהליך חדש בLinux. סיום תהליך בLinux.

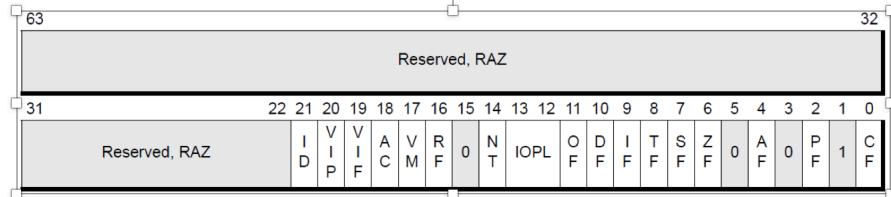


רגסטרים מיוחדים.  
רגסטרים מיוחדים.

- *RIP* - מצביע על הפקודה הבאה לביצוע. כמו כל הרגיסטרים, גם הוא ברוחב 64 ביט.
- *RFLAGS* - שומר את המצב הנוכחי של המעבד, לדוגמה:
  - בית מס' 7 (*sign flag*) מציין האם תוצאה חישוב האחרון הייתה שלילית.
  - בית מס' 6 (*interrupt flag*) מציין אם המעבד מקבל פסיקות.
- *CS* - שומר את רמת ההרשות הנוכחית (*CPL*).

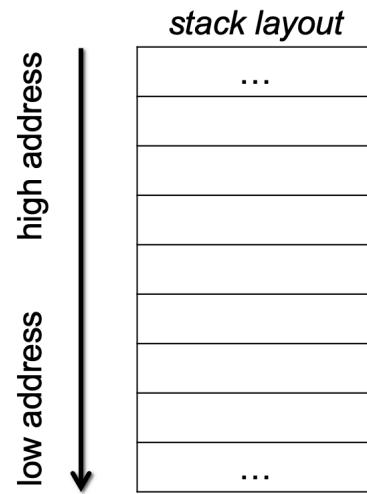
## מערכות הפעלה

110



### מחסנית הקריאה

- כל תכנית מתחזקת מחסנית קריאות במהלך ריצתה.
  - למשל, כדי לשמר משתנים מקומיים של הפונקציה.
- בעת קרייה לפונקציה התוכנית שומרת על המחסנית
  - את כתובות החזרה מהפונקציה,
  - ואת הפרמטרים המועברים לפונקציה.
- אנחנו נזכיר את המחסנית גדלה למטה (כתובות נמוכות למטה).



קונבנציית לינוקס לקרייה לפונקציות. חוקים של הקוראת (*caller rules*)

- לפני הקרייה לפונקציה: שמירת הרגיסטרים באחריות הקוראת
- העברת 6 פרמטרים ראשונים ברגיסטרים (משמאלי לימין): *(rax, rcx, rdx, rdi, rsi, r8-r11)*
- העברת 6 פרמטרים ראשונים ברגיסטרים (משמאלי לימין): *(rdi, rsi, rdx, rcx, r8, r9)*.
- העברת שאר הפרמטרים (מעבר לששת הראשונים) על המחסנית.
- פקודת מכונה *.call*.
- בחזרה מהפונקציה:

## מערכות הפעלה

- שליפת הפרמטרים שנדחו על המחסנית.
- שליפת הרегистרים שנשמרו.

**חוקים של הנקראת (callee rules)**

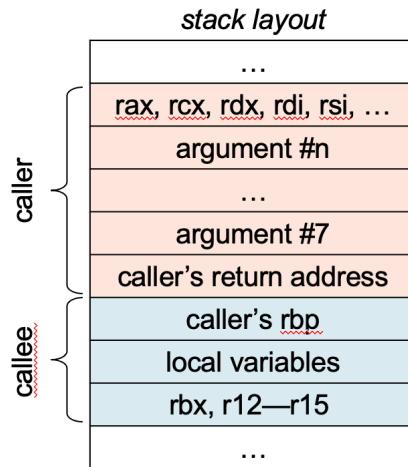
- בכניסה לפונקציה:
  - פתיחת מסגרת חדשה (שמירת  $rbp$  הישן והצבעה לראש המחסנית).
  - הקצאת משתנים מקומיים.
  - שמירת הרегистרים באחריות הנקראת ( $rbx, r12-r15$ ).
  - המחסנית.
- ביציאה מהפונקציה:
  - העברת ערך החזרה  $rax$ .
  - שליפת הרегистרים שנשמרו.
  - שחרור המשתנים המקומיים. חזרה למסגרת הישנה (שליפת  $rbp$  שנשמר קודם).
  - פקודת מכונה  $ret$ .

קריאה וחזרה מפונקציה.

- פקודת המכונה  $call$  :
  - דוחفت את כתובת החזרה (ערךו הנוכחי של  $rip$ ) למחסנית,
  - וקופצת לתחילת הפונקציה על-ידי הצבת כתובת הפונקציה  $.rip$ .
- פקודת המכונה  $ret$  :
  - שולפת את כתובת החזרה מראש המחסנית,
  - וקופצת לכתובת זו על-ידי הצבתה  $.rip$ .

## מערכות הפעלה

112



מחסנית הגרעין בזמן טיפול בקריאה ממערכת או פסיקה.

- בתחילת הטיפול כל הרגיסטרים נשמרים על מחסנית הגרעין באמצעות המacro *PUSH\_REGS*.
  - כי הגרעין צריך את מצב המעבד לפני קריאת המערכת או הפסקה.
- בסוף הטיפול כל הרגיסטרים נשלפים באמצעות המacro *POP\_REGS*.
  - בין הרגיסטרים נשלה גם *rax* וכן שגרת הטיפול למשה מוחזירה את הערך במקרה של קריאת מערכת.

## מערכות הפעלה

113

| kernel stack |
|--------------|
| ss           |
| rsp          |
| rflags       |
| cs           |
| rip          |
| orig_rax     |
| rdi          |
| rsi          |
| rdx          |
| rcx          |
| rax          |
| r8           |
| r9           |
| ...          |
| r15          |

מהי החלטת הקשר? קודם כל אנו נתמיכל את הנושא מרמה אבסטרקציה ונצלוּן כדי לראות איך הכל מנהל מבנית רגיסטרים וכו'. ההחלטה הקשר קוראים לזה כך מהAMILה הקשר ביצוע (מכיל כל מידע הנדרש לביצוע התהליין), שאנו מחייבים בין תהלייכים אנו מספקים את הביצוע של התהליין שרצ ושמור את כל הקשר שלו בצד ולטעון ההקשר של התהליין הבא לביצוע וכך הפעולה נקבעת החלטת הקשר.

- **כל תהליין יש "קשר ביצוע" (execution context)** המכיל את כל המידע הדרוש לביצוע התהליין. מחסניות, רגיסטרים, תוכולת זיכרון, קבועים פתוחים, ...
- **"ההחלטה הקשר"** =
  - עצירת הביצוע של התהליין הנוכחי ושמירת ההקשר שלו.
  - טיעינת ההקשר של התהליין הבא לביצוע.
- **קשר התהליין הנוכחי מתחלף** - מכאן שם הפעולה "ההחלטה הקשר".

יתרונות וחסרונות של ההחלטה הקשר. קודם כל בקטנת זמן תגובה של תהלייכים אינטראקטיביים זהה יתרון שני למטה, אז דוגמא קנוונית היא למשל WORD כך שתהליין A רץ על המעבד ותהליין שני שלו מחליף

## מערכות הפעלה

114

לתוים ממקלדת, ברגע שמקבל פסיקת מקלדת, אנו נרצה ש  $A$  יטפל בה מיד ופסיקת מקלדת נעה תחילה  $B$  וברגע שהוא מתעורר מערכת הפעלה מיד תעshaה החלפה בין תחילה  $A$  וב  $B$  בשביל לראות מיד את התו של על המסך ואם לא הייתה החלטת הקשר אז החוויה של המשתמש הייתה מאוד לא טובה, למשל כולם מכירים את התחושה שיש הרבה תחילcis ואז ההחלטה הקשר קוראות לאט ואז אנו מקלדים עוד ועוד ואז WORD נראה קופא ומקליד הכל בביטחון אחת. סיבה שנייה זה להגדיל נזילותות של המעבד, ברגע שתחילה ממתין חנтоונים מדיסק או כרטיס רשות כל דבר אחר, אז עדיף מערכת הפעלה לה Kapoorיא אותו ולתת לתחילה אחר לרווח שבמעבד שלנו לא יהיה מובטול וחסר עבודה. כמובן, יש חיסרון בהזאה כך שההשפעות הקשר עולגה זמן ואם נעשה יותר מדי אז נזיבז די' זמן ובמערכות רגילות אנו מנסים נזיבז רק 2-3% על ההחלטה הקשר.

- ניצול טוב יותר של משאבי המערכת. לדוגמה: כאשר תחילה  $A$  ממתין לנוטונים מהדיסק, מערכת ההחלטה תגרום לו לפנות את המעבד ותקרא לתחילה אחר □ במקומו.

- הקטנת זמן התגובה של תחילcis אינטראקטיביים.

- לדוגמה: תחילה  $A$  רץ על המעבד, תחילה  $B$  ממתין לתווים ממקלדת. ברגע שתגיע פסיקת מקלדת, היא תטוף בהקשר של תחילה  $A$ , ותעיר את תחילה  $B$ .

- מערכת הפעלה תפקיע את המעבד מתחילה  $A$  ותזמן לריצה את תחילה  $B$ .

- פגעה בנזילותות המעבד (CPU utilization).

- מערכת הפעלה מבזבזת זמן על שמירת ההקשר הנוכחי וטעינה ההקשר החדש.

שני סוגים של החלטת הקשר. ההחלטה הקשר יזומה

- תחילה מותר מרצונו על המעבד, למשל באמצעות:

- קריאת מערכת חוסמת (כמו `wait()`, `read()` . . .), אשר מוציאה את התחילה להמתנה.

## מערכות הפעלה

115

- קריית מערכת (`exit()`) אשר מסיימת את התהיליך.
- קריית מערכת (`sched_yield()`) - קריית מערכת ייעודית לוויתור על המעבד.

החלפת הקשר כפואה) == הפקעה).

- הגሩין מפקיע (כלומר, לוקח בכוח את המעבד מהטהיליך, למשל בעקבות
  - פסיקת שעון (מטופלת בשגרה) (`scheduler_tick()`) אשר מוגלה כי הזמן שהוקצב לתהיליך הנוכחי אל.
  - אירוע אסינכרוני אשר מעיר תהיליך בעל עדיפות טובה יותר מהטהיליך הרץ-current. לדוגמה: פסיקת דיסק או שחרור מנעול שתהיליך המתין לו.

החלפת הקשר יזמה זה בעצם מה שקורה שתהיליך מוותר מרצוינו על המעבד שהוא מבין שהוא הולך לקרוא לקריית מערכת שתוותר על המעבד למשל `wait` קריית מערכת חוסמת שתהיליך אבא רוצה להמתין לבנים שלו שהיא הולכת לוותר על המעבד כי תעביר תהיליך אבא לתוכה המתנה ולהמתין עד שיקרה איזשהו אירוע שייעיר אותו או `read` שקוראת נתונים מדיסק שתהיליך קורא לה הוא יודע שהוא הולך לוותר שקוראה על המעבד ומעבירה אותו מצב משתמש למצו גרעין ומשם מרצוינו על המעבד ומעבירה אותו במצב שוחרר על המעבד או המימוש של קריית המערכת `read` תוצאה את התהיליך להמתנה או `exit` שמסיימת את התהיליך הנוכחי לכו בודאי שמוותר על המעבד או `epoch` שזאת פקודה ייעודית שמוותר על המעבד ל `sched_yield` מסויים.

החלפות הקשר כפויות אמרנו שם יש לנו תהיליך בן אחד שMRI'ץ `while` ולטחן המעבד ולא רוצה לשחרר את המעבד כי הוא רע אז לא נרצה לחת לו להשתלט אז נקח ממנו בכוח (הפקעה) ממנו את המעבד בעקבות שני סוגי של אירועים פסיקות שעון (AIROU KNONI RCIB CHIZONI) של המעבד כמו שעון מעורר שמעיר את המעבד כבפרק זמן קבועים 10ms, 20ms ואז מטופלת ענ' ידי מערכת הפעלה והיא מחליטה מה

## מערכות הפעלה

116

לעשות) או אירוע אסנכרוני שמעיר תהליך בעל עדיפות יותר גבוהה ממנה שרשן כרגע ואז עושם החלפת הקשר.

הערה. בטרמינולוגיה של לינוקס צריך לשים לב ש מעבר ממצב משתמש למקורין זה לא החלפת הקשר, אנו אומרים שהתהליך רץ במצב גרעין.

. הפקעה (*preemption*)

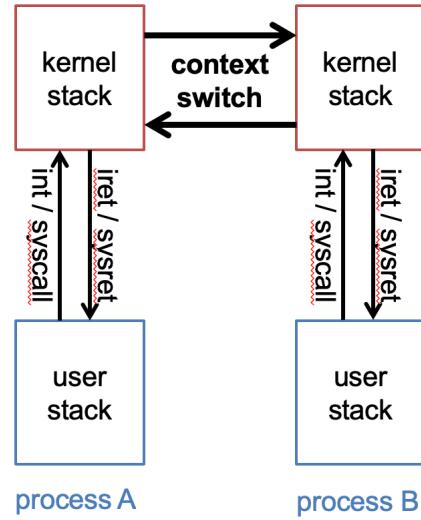
- בעיה: תהליך משתמש עלול לרווץ לנצח (למשל, לוולאה אינסופית) ולמנוע את המעבד משאר התהליכים. פגעה בהוננות (responsiveness).
- פתרון: לינוקס מפרקעה (*preempt*) את המעבד מהתהליך אחד לטובת תהליך אחר, בעזרת התקן חומרה מיוחד - השעון.
- מערכת הפעלה מבקשת מהשעון לשלווה פסיקה במרוח כי זמן קבועים כדי להעביר את השליטה למערכת הפעלה. כל הפסיקות, בפרט פסיקת שעון, מטופלות במצב גרעין, ואז הגרעין מחליף הקשר אם יש צורך.

. החלפת הקשר מתרכשות במצב גרעין.

- החלפת הקשר דורשת את התערבות מערכת הפעלה, וכך היא מתרכשת במצב גרעין.
  - כפי שקרה התרשים, המעבד במצב משתמש למצב גרעין מתרכש רק בעקבות קריאת מערכת או פסיקה (חומרה/תוכנה).
- לב העניין הוא ההמלפה בין מחסניות הגרעין של שני התהליכים המעורבים.

## מערכות הפעלה

117



בצד שמאל רואים תחילה  $A$  שיש לו מחסנית משתמש וגרעין החלפת הקשר זה המעבד בין תחילה  $A, B$  זהה בין מחסניות המשתמש והוא קורה בפועל על ביצועה הצעת  $A$  עובד ממצב משתמש לגרעין וממצב גרעין של  $A$  למצב גרעין של  $B$  וממצב גרעין של  $B$  למצב משתמש של  $B$ . ככלומר החלפת הקשר היא מתרכשת רק בעקבות מעבר של רמות הרשאბ כלומר אי אפשר לעבוד ממחסנית משתמש של  $A$  למחסנית משתמש של  $B$ . נשים לב שגם תחילה לא עובד למצב גרעין לא יכול להיות החלפת הקשר אבל זה בלתי אפשר כי יש יש שעון שנוטן פסיקות במרוחבי זמן קבועים ודואג להעביר תחילה ממצב משתמש לגרעין כדי שמערכת הפעלה תקבל שליטה לדיים.

אין גרעין מפעיל החלפת הקשר?

- ע"י קריאה לפונקציה (`schedule()`) אשר: בוחרת מי יהיה התהילה הבאה ליריצה. מבצעת את החלפת הקשר ע"י קריאה לפונקציה `.context_switch()`.
- `schedule()` היא שער הכניסה היחיד להחלפת הקשר בلينוקס.

ראינו לפניו ש `schedule` בוחר את התהילה הבא ליריצה (אלגוריתם זימון של לינוקס) הפונקציה קוראת ל `context_switch()` והתקף ייד

## מערכות הפעלה

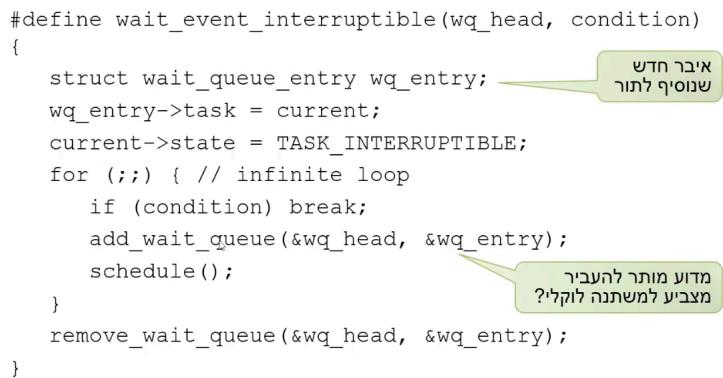
118

היחד שלזה זה להחליף בין תחליין הבא ליריצה עם התהליין שרגעםprev בהנחה שיודעת מי התהליין הבא ליריצה.

דוגמת קוד מתוכן הגרעין.

- המacro ()*wait\_event\_interruptible*(*wq\_head*, *condition*) מכניס את התהליין הנוכחי למתנה בתור *wq\_head* עד אשר התנאי *condition* מתקיים.

```
#define wait_event_interruptible(wq_head, condition)
{
 struct wait_queue_entry wq_entry;
 wq_entry->task = current;
 current->state = TASK_INTERRUPTIBLE;
 for (;;) { // infinite loop
 if (condition) break;
 add_wait_queue(&wq_head, &wq_entry);
 schedule();
 }
 remove_wait_queue(&wq_head, &wq_entry);
}
```



הסביר קוד. נשים לב שיש לנו מקרו שמכניס את התהליין שמקביל לתור המתנה עד שתנאי כלשהו מתקיים יש לנו פה את *schedule* שהוא משתמש בה שתהליין עבור לתור המתנה הוא צריך לוותר על המעבד לכין קורא לה. האם *schedule* נקראת בכל פסיקות שעון? לא! כי ראיינו שככל פסיקת שעון מורידה ממספר הפסיקות שתהליין ירוז כלומר אם רץ 5 פסיקות שעון אז שפסיקה מגיעה אז הוא יורד עכשו 45 פסיקות שעון זהה ה *counter* ואז יגע עוד ועדו עד שיגר 50 ואז קוראים ל *schedule* וועושים החלפת הקשר.

שתי דרכי להפעיל החלפת הקשר. דרך ראשונה זה *schedule* למשל בקריאה מערכת *wait* המימוש שלזה קורא לה מקרו שראינו למעלה וזה קורא להחלפת הקשר יזומה. דרך אחרת היא על ידי הפקעה לראיינו בהרצאה שתהליין שלווה לשני פסיקה ואז מדליק את ה *need\_resched* ששמור ב *PCB* של התהליין ושהוא רוצה לעבור למצב משתמש הוא בודק את הדרוג ואם הוא דלוק אז *schedule* נקראת ומתחבצעת החלפת הקשר.

# מערכות הפעלה

119

## שתי דרכי להפעיל החלפת הקשר

הפעלה דחונית  
Lazy Invocation

|              |                                   |
|--------------|-----------------------------------|
| kernel stack | rsp,rflags,rip,...                |
|              | rax,rbx,rcx,...<br>(by PUSH_REGS) |
|              | scheduler_tick()                  |
|              | ...                               |
|              |                                   |
|              |                                   |
| PCB          | need_resched = 0                  |
|              |                                   |

הפעלה ישירה  
Direct Invocation

|              |                                   |
|--------------|-----------------------------------|
| kernel stack | rsp,rflags,rip,...                |
|              | rax,rbx,rcx,...<br>(by PUSH_REGS) |
|              | sys_wait()                        |
|              | ...                               |
|              | schedule()                        |
|              | ...                               |
| PCB          | ↳                                 |
|              |                                   |

## שתי דרכי להפעיל החלפת הקשר

הפעלה דחונית  
Lazy Invocation

|              |                                   |
|--------------|-----------------------------------|
| kernel stack | rsp,rflags,rip,...                |
|              | rax,rbx,rcx,...<br>(by PUSH_REGS) |
|              | ↳                                 |
|              |                                   |
|              |                                   |
|              |                                   |
| PCB          | need_resched = 1                  |
|              |                                   |

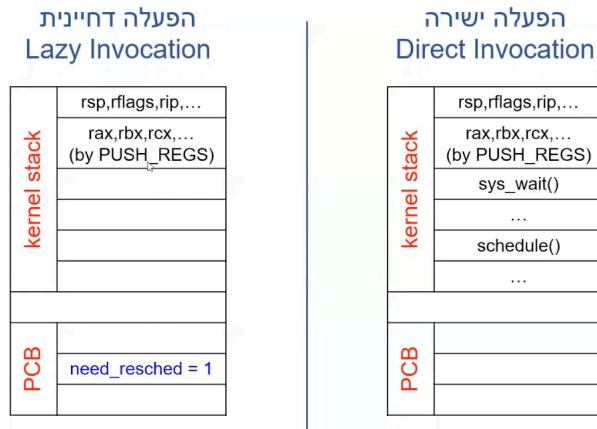
הפעלה ישירה  
Direct Invocation

|              |                                   |
|--------------|-----------------------------------|
| kernel stack | rsp,rflags,rip,...                |
|              | rax,rbx,rcx,...<br>(by PUSH_REGS) |
|              | sys_wait()                        |
|              | ...                               |
|              | schedule()                        |
|              | ...                               |
| PCB          | ↳                                 |
|              |                                   |

## מערכות הפעלה

120

### שתי דרכי להפעיל החלפת הקשר



למשל ניתן לראות שסדרען של תהליך הקורא ל-*wait* המחסנתו שלו כולל את כל הפקנציות ונקרו שדיברנו עליהם זהה קורא למשל בהפעלה ישירה לעומת הפעלה דחינית גרעין בה מבצע שגרת טיפול בפסיקת שעון *schedule\_tick* שמדליקת את הדגל *need\_resched* וברגע שתסתיים ונרצה לחזור למצב משתמש יש בדיקה ואם הדגל אחד אז *schedule* נקראת.

הפסקה במצב גרעין. למה צריכים הפסקה זו? אפשר לנקחת את המעבד גם מהתהילך שרשן במצב גרעין, זה אומר שרגם אם התהילך במצב הביצוע של קריאת מערכת *fork* אפשר לעצור את התהילך ולעבור לתהילך אחר לביצוע. זה שימושי בגלל הוגנות וגם אינטראקטיביות כי גרעין יש דברים שלוקחים זמן למשל *fork* מעתיקת התהילך הנוכחי וזה יכול להיות ברובה מידע לנו ניצן לעצור ולתת לתהילך לרווח.

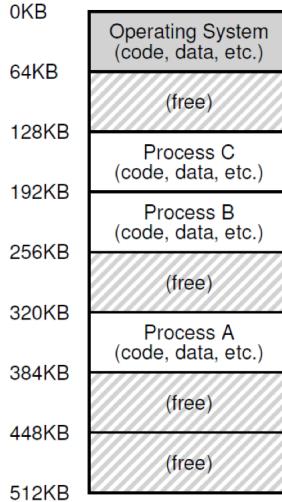
היכן נשמר הזיכרון של התהיליך?

- אזרוי הזיכרון של כל התהיליכים (וגם של מערכת ההפעלה) חיים זה לצד זה.
- החלפת הקשר אינה דורשת "שמירה" או "טעינה" של אזרוי זיכרון, אלא רק החלפת מרחבוי הזיכרון.

## מערכות הפעלה

121

- במעבדי  $64x$ , מערכת הפעלה מחליפה את מרחב הזיכרון ע"י טוינט ערך חדש לרגיסטר  $.CR3$ .



חשוב לציין שבhalbת הקשר אין לא מוחקים שום זיכרון אנו פושים מקפיאים השימוש فيه וניגשים לזיכון של תחלה אחר. מה כן צריך למשור בזמן החילופות בקשר אם כבר לא הזיכרון? למשל כתובות חוזרת. היכן נשמר הקשר התהליין?

(1) המחסנית, הערימה, הקוד נמצאים בזיכרון. כאמור, אין צורך

"לשמור" ולטוען" אוטם מחדש בכל החלפת הקשר.

(2) נתונים אחרים, למשל הקבצים הפתוחים (*file descriptors*), נשמרים במתאר התהליין (*PCB*) – אשר גם נמצא בזיכרון.

(3) את הרегистרים והדגלים (מצב המעבד) יש לשמר ולטוען מחדש לאחר מכן. השמירה והטיענה מתבצעת במחסנית הגרעין ובסדה *PCB-thread*.

(4) בנוסף, יש לשמר פריט מידע נוסף שעד עכשו התעלמן ממנו – את בסיס מחסנית הגרעין של התהליין הנוכחי.

ברגע שיש החלפת הקשר התהליין מחליף ממחסנית משתמש למחסנית גרעין, כיוון שהרגזטרים משתמשים בהם במצב משתמש כדי לא ל Abed את הערך בהם שעוברים למצב גרעין אחרי שפסיקה מתקבלת אנו שומרים אותו במחסנית הגרעין בשדה מיוחד בתוך ה *PCB* עכשו

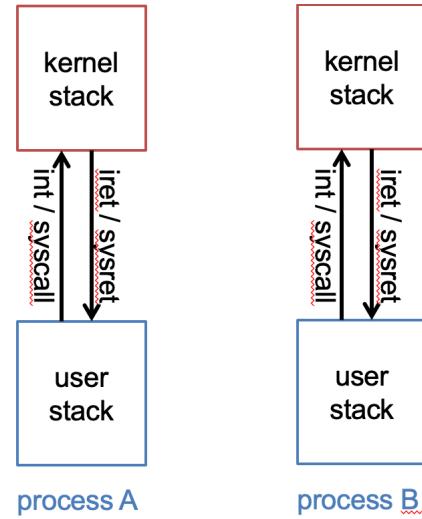
## מערכות הפעלה

122

ברגע שיש פסיקה אנו שומרים אותו ומרבים אותו שוחזרים למאובט  
משתמש אם נדרש.

תזכורת.

- כאשר תהליך רץ במאובט משתמש ואז מתאפשרת פסיקה, המעבד מחליף מחסניות ושומר את ההקשר של המשתמש על מחסנית הגרעין.
  - לכל תהליך יש מחסנית גרעין משלהו.
- שאלת: איך המעבד יודע איפה נמצאת מחסנית הגרעין של התהילין הנוכחי?
- תשובה: באמצעות מבנה מיוחדanntו נקרא  $.TSS$ .

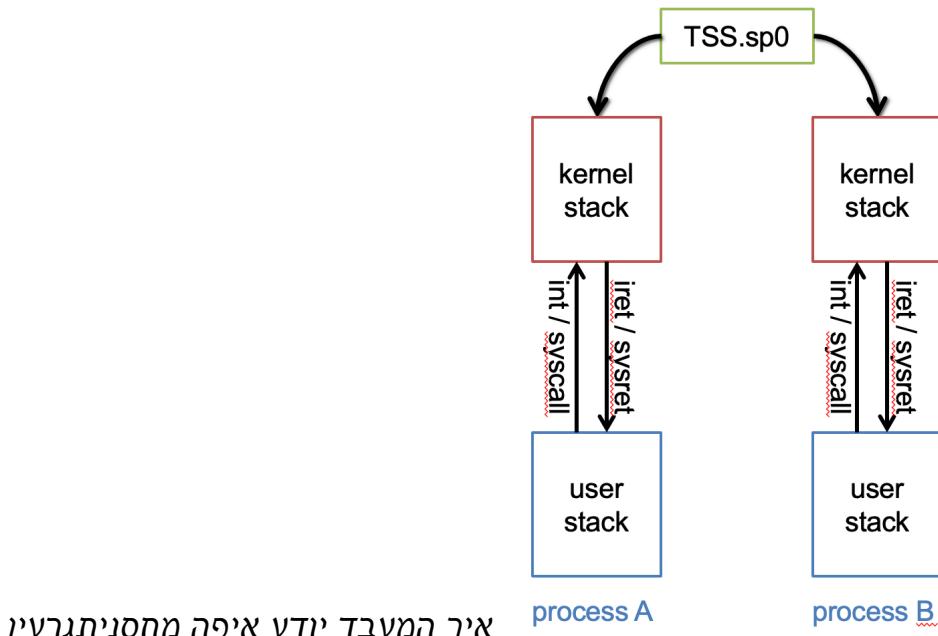


$.TSS$

- מעבר בין רמות הרשאה ( $user mode \rightarrow kernel mode$ ) מעבדי 32 – IA-32 קוראים את השדה  $TSS.sp0$  כדי למצוות את בסיס מחסנית הגרעין.
- הצלפת הקשר צריכה אם כן לעדכן את השדה  $TSS.sp0$  כדי שיבצעי למחסנית הגרעין של התהילין הבא לביוץ.

## מערכות הפעלה

123



אין המעבד יודע איפה מחסנית גרעין

process A

process B

של התהיליך? כזכור לכל מחסנית גרעין יש כתובת משלה. נשים לב שאי אפשר דורך *PCB* כי שмагיעה פסיקה הגרעין עוד לא יודע איפה ה *PCB* הרி *PCB* הוא בתוך הגרעין וудין לא עברנו למצב גרעין. אז לכל מחסנית גרעין יש כתובת ששומרים ברגסטר שנקרא *TSS* והוא מצביע לבסיס של מחסנית גרעין לתהיליך שרגע ובעצם בהחלפת הקשר טוענים לרגistrar את כתובת של נחסנית של התהיליך החדש. נשים לב לכל מעבד יש סט רגיסטרים שלו וזהו כל אחד יש לו *TSS* משלהו.

שדה *thread* במתאר התהיליך. אנחנו הולכים לעבור בין מחסניות יש לנו רגistrar מיוחד שמצויב תמיד למקום הנוכחי במחסנית *dsr* והוא שונה מ *0ds* שראינו שמצויב לבסיס של המחסנית כרגע אנו מדברים על הרגistrar שמצויב בראש של המחסנית מקום הנוכחי במחסנית איפה שדוחפים והבסיס הוא לא איפה שדוחפים וכמובן אנו נשמר אותו בעת החלפת הקשר נשמר אותו בתוך סטרukt חדש *thread\_struct* שמופיע ב *PCB*.

## מערכות הפעלה

124

- השדה הוא מבנה מטיפוס *struct\_thread* אשר נמצא בתוך ה-*PCB*.
- השדה משמש לשימרת חלק מהקשר התהליין. פריט המידע החשוב מבחרינו הוא המצביע לראש מחסנית הגרעין שנשמר ונטען בזמן החלפת הקשר:

:*TSS* (*Task State Segment*)

- (*TSS* (*Task State Segment*) הוא מבנה נתונים הנמצא בזיכרון הגרעין, ומכיל מידע על הקשר הנוכחי המתבצע במעבד.

```
struct tss_struct {
 ...
 unsigned long sp0;
 ...
};
```

מצביע לבסיס מחסנית הגרעין  
של התהליין הנוכחי במעבד

- מצבע ע"י רегистר מיוחד במעבד הנקרא *TR*.
- במערכת מרובת ליבות מוקצה לכל ליבה מבנה *TSS* משלה

**context\_switch()** – פונקציה C  
כללית, לא תלויות ארכיטקטורה.

**switch\_to\_asm()** – פונקציה ספציפית  
לארכיטקטורה, כתובה באסמבלי Ci  
קוראים/כותבים לרגיסטרים.

**switch\_to()** – פונקציה C, ספציפית  
לארכיטקטורה Ci ניגשים למבנה TSS.

שלבי החלפת הקשר.

רואים פה שיש כמה שלבים להחלפת הקשר, קדום של הfonקציה הראשית היא Si כללית והיא לא תלית ארכיטקטורה אבל כMOVED שחלפת

## מערכות הפעלה

125

הקשר צריכה ארכיטורה וצריה לגעת ולדבר בשפה של רגיסטרים וכך  
צריכים לעבור לפונקציה אסםבלי שבהיא אחראי על זה, ואז עוברים  
*to switch* – שהיא סימפטיית לארQUITוRA כי מדובר בשפה  
של *TSS* והוא סימפטיית תלוית ארכיטורה כי מדובר בשפה של  
מבנה *TSS* שקיים רק במעבדי אינטל אבל אם נעבור ל –  
*RISC* אז לא נראה דברים כאלה.

```
context_switch(..., struct task_struct *prev,
 struct task_struct *next, ...){
 ...
 ...
 switch_mm(oldmm, mm, ...);
 ...
 __switch_to_asm(prev, next);
 ...
}
```

מצביע למתרחשות הינה הולך ומשתנה על המעבד

מצביע למתרחשות הינה מקבל את המעבד

החלפת מרחבי הזיכרון

המקרה ששומר את ההקשר  
הנוכחי וטוען את ההקשר החדש

.context\_switch()

הסביר קוד. נשים לב שני ארגומנטים החשובים פה *next*, *prev* מטיפוס סטרukt זהה טיפוס *PCB* מתאר תהליכי ששורר כל פרטיו מידע של תהליכי, אז אנחנו מקבלים מצביעים אליהם אז יש לנו שני PCB אז עושים החלפה ביניהם ויש גם *switch\_mm* שעושה החלפה של מרחבי זיכרון בין תהליכיים ובסיום יש את המקרה *switch\_to\_asm* שכתובה באסםבלי.

## מערכות הפעלה

126

(1) \_\_switch\_to\_asm הפונקציה

```
_switch_to_asm:
 pushq %rbp
 pushq %rbx
 pushq %r12
 pushq %r13
 pushq %r14
 pushq %r15
```

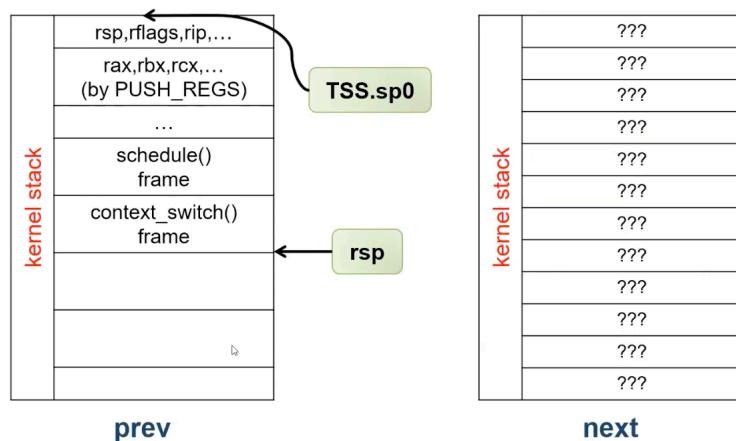
`movq %rsp, prev->thread.rsp`  
`movq next->thread.rsp, %rsp`

הfonקציית `switch_to_asm` עמודת לבצע החלטת הקשר לתוך חדש ולכך על הרגיטרים במעבד יכולם להשתנות. אבל הקומpileר לא יודיע את זה!  
 לנוכח הקוד של `switch_to_asm` ציר לשמר בעצמו את הרגיטרים שהם באחריות ה Fonקציית הנקראטית ושלוחר אותם לפני החזרה מהפונקציה.

לפי קונבנציית הקריאה לפונקציות, הפרמטרים next, prev נמצאים ברגיסטרים rsi, rdi בהתאמה

הסביר קוד. נשים לְבָשָׂר שקדם כל אנו דוחפים את 6 הרגסטרים על מחסנית, אלו רגסטרים שהם באחריות הפונקציה הנקראת כי החלטת הקשר אפשר לחשוב עליה כקריאה לפונקציה כמו שתהלייך מסוים יכול לקרוא לפונקציה וכל הרגסטרים יכולים להידرس לו שהוא مختلف לתהליין אחר גם כל הרגסטרים יכולים להידرس לו כי התהליין החדש יכול לגעת בכל הרגסטרים. אחרי זה עושים החלטת מחסניות, מיד נראה את זה בשרטוט.

## תמונה מצב לפני החלפת הקשר

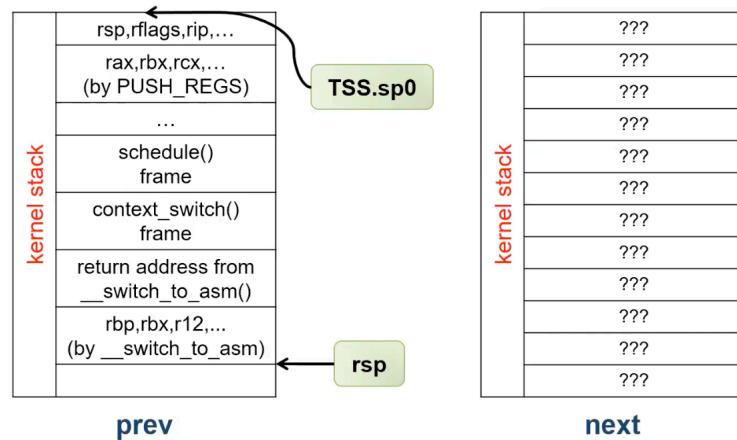


## מערכות הפעלה

127

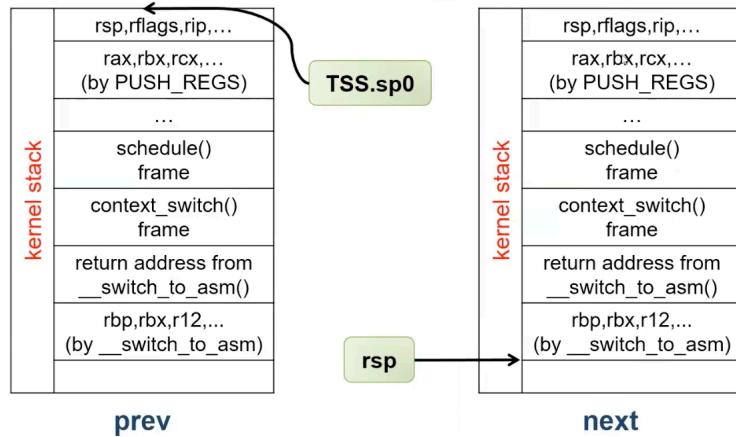
יש לנו החלטת הקשר וננו הולכים לראותו למשל בצד שמאל יש את התהיליך *prev* וימן את *next*. נשים לב שההחלטה על הקשר לא קוראת ברגע לא צפוי, אלא ברגע שאיכשהו התהיליך *prev* הגיע למצב גרעין שקרה על שרה *schedule* או *context\_switch* באותו רגע ה-*kdss* מצביע לתוך מחסנית הגרעין של *prev* כלומר באותו רגע קראת *kdss* מצביע *swirch\_to\_asm* והוא יודעת לשמור את הרגיטרים ואת כתובות החזירה ואז יהיה לנו מצב זה,

### תמונה מצב לפני החלטת המחסניות



עכשו אנו מחליפים מחסניות? אם ניתן לדעת לאן *dsr* יצביע עכשו בתוך *next*? תשובה ראשונה היא ש *rsp* יצביע לבסיס של *next* ומשם הוא מתחילה, או נקודה שבא סימנו את הפעלה, האם ניתן לדעת אותה? נשים לב שהיא ב *thread* שבו נשמר ה *rsp*. עכשו נשים לב ש *prev* עושה החלטת הקשר אבל גם עשה החלטת הקשר בעבר אז גם הוא שיחק פעם את התפקיד של *prev* لكن מחסנית שלו ראה כמו המיסנית של *prev* כלומר *next* נראית אליו גם היא פעם קראה על שרה `schedule`, `context_switch` אבל מה אם *next* אף פעם לא עבר החלטת הקשר?

## תמונה מצב במקורה הסטנדרטי



`:switch_to_asm(2)` הפקזה

- השורה הבאה במאקרו מחליפה את מחסניות הגרעין, מזו של לוזן של לוזן - זו למעשה נקודת החילוף ההקשר:

*movq next- > thread.rsp,*

- שאלת: לאן מצביע `next`? תשובה: תלויה...  
נפריד לשני מקרים:  
fork() מקרה אתחול: התהיליך `next` הוא תהליך חדש שנוצר ע"י ()  
ועדין לא רץ אף פעם. נחזיר לדzon במצב זה בסוף התרגום.
  - מקרה סטנדרטי: התהיליך `next` כבר רץ בעבר וuber החלטת הקשר. במקרה זה המחסנית של `next` נראהת כמו המחסנית של `prev, prev`, כי `prev` הוא תהליך שעובר עכשו החלטת הקשר!

מה הדבר הראשון צריך לעשות שועברים למחסנית של *next*? זה לא חייב להיות זהה בין המחסניות. למה אין לך מראות אותו דבר? כי יש כמה דרכים להגיע לך מהר יותר.

## מערכות הפעלה

129

### הfonקציה `__switch_to_asm`

```
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbx
popq %rbp
```

שחזור הרגיסטרים ממחסנית הגרעין של next.  
אל הרגיסטרים next שמר כאשר הוא קרא  
להחלפת הקשר בעבר.

```
jmp __switch_to
```

קפיצה (jmp) במקום קריאה (call) לפונקציה. למה?  
כתובת החזירה מהfonקציה ()  
`__switch_to` כבר  
שמורה על המחסנית של next.

מה קורה ב `__switch_to`

### הfonקציה `(__switch_to)`

- פונקציה זו משלימה את רצף הפעולות במסגרת החלפת ההקשר.

```
__switch_to(struct task_struct *prev_p,
 struct task_struct *next_p) {
 ...
 update_sp0(next_p);
 ...
 return;
}
```

עדכן מצביע מחסנית הגרעין  
של התהיליך הנוכחי ב-TSS

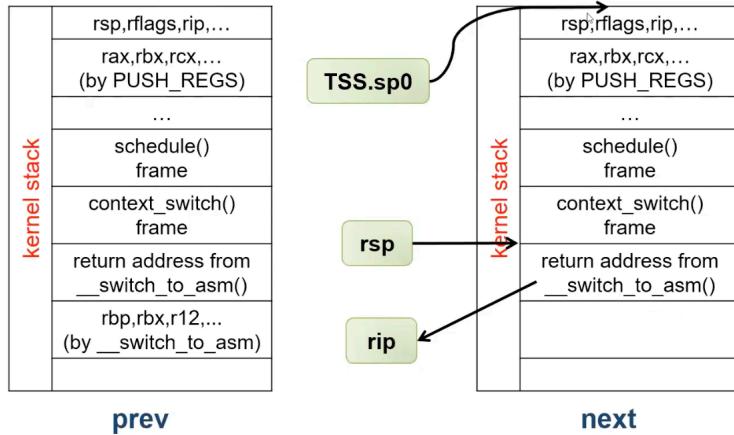
שליפת כתובת החזירה מראש המחסנית

נשים לב שכיבים קפיצה במקום `call`. בעצם מה שהולך לקררות  
אחרי `switch_to`

## מערכות הפעלה

130

### תמונה מצב אחריו (`_switch_to()`)



ש `switch_to_asm` מסתיימת היא שולפת כתובה החזירה מ `switch_to` לתוכו `rip` רגיסטר שמצוין לפקודה הבאה לביוץ ואז בעצם סיימנו החלפת הקשר וחזרים למי שקוראים ל `asm` ולנקודה ממנה קראנו ל `context_switch` ואז המסגרות יתקפלו ונחזיר ממחסנית גרעיין של `next` למחסנית משתמש שלו ונסיים. אז למה עושים `jmp` ? אם היינו עושים `call` איז הינו שומרים עוד פעם את כתובות החזירה מהפונקציה שהיא בעצם היצא פקודה הבאה לביוץ זה לא מה שרצוinos ל לעשות אלא רוצים לחזור לצי שקרה ל `switch_to_asm` בקוד של `next` ולא קוד של `prev`. ولكن בשביל ללקחת כתובות על המחסנית של `next` צריך `switch_to` וא; הפונקציה `switch_to` אחראית לשלוף את זה ל `rip`.

- שאלה: מדוע הפונקציה `switch_to()` מופעלת באמצעות קפיצה פקודת (`jmp`) ולא באמצעות קריאה רגילה פקודת (`call`) ?

• תשובה: פקודת `call` דוחפת למחסנית את כתובות החזירה של השורה הבאה לביוץ, ואנחנו רוצים כתובות חזרה שונה בהתאם ל蹶ה:  
 - במקרה המקורי: הכתובת היא `ret_from_fork` וכי שנראה בהמשך.

## מערכות הפעלה

131

- במקרה הסטנדרטי: הכתובת היא כתובת החזקה מ-`switch_to_asm()`.

- בשני המקרים, מכיוון ש-`switch_to()` מוגדרת כפונקציה לCALL דבר, הקוד שלו מסתיים בפקודת `ret` ששולפת כתובת חזקה מהמחסנית (של התהיליך הבא לבייעו) וקופצת אליה.

תקציר למה שלמדנו. התחלנו מתוכנית הכללית שהשלי החלפת הקשר יש לנו context-switch שהיא שער היחיד להחלפת הקשר בLINOKS ולא תלוית ארכיטקטורה והוא מחייב בין תהליכיים ואז עבר ל`switch_to_asm()` שהוא כתובה באסמבלי, קוד סימלאר בשפה של רגיסטרים והוא קוראת ל`switch_to_s()` שהוא פונקציה סי כי מתכסקת עם הסדרקט שנקרה *TSS* ראיינו שהוא `switch_to_asm()` הוא הבא, שומרת את הרגיסטרים *r15 - rbp* שעולמים להשתנות ובאחריות הפונקציה הנקרה ושומרת רותם כי מיד עושים החלפת הקשר לטהיליך *next* שימושה באיזה רגיסטרים שהוא רוזה, ראיינו שהוא יכול להיות אחד משני מקרים תהיליך שכבר רץ בעבר ועבר החלפת הקשר או לא אם כן עבר אז מחסנית שלו נראית בערך כמו *prev* וזהו שומר את הרגיסטרים שלו *r15 - rbp* אך מה ש`switch_to_asm()` זה לשולף רגיסטרים אלה לחזור לקוד ש`next` שביעע לפני שעבר החלפת הקשר ועשה `ret` זהה `pop` לכתובת ווחזרת לפונקציה של `next` כדי לחזור אליה ואז אחרי עושים את זה קוראים לפונקציה `switch_to_s()` משלימים חתיכה אחרתה בפואז של החלפת הקשר שהיא לחתול *TSS* להציביע לבסיס המחסנית של `next` אז שמסתיימת החלפת הקשר *TSS* מהציביע לבסיס המחסנית של `next` ו-`rsp` מציביע לתוך מחסנית הגרעין של `next` ו-`rip` מכיל כתובת החזקה מ-`switch_to_asm()` והקוד ממשיך כרגע שהוא המקום בקוד שבו קראה ל`switch_to_asm()` כתובת context-switch כלומר פקודה הבאה לבייעו אחרי `switch_to_asm()` בקוד הבא:

## מערכות הפעלה

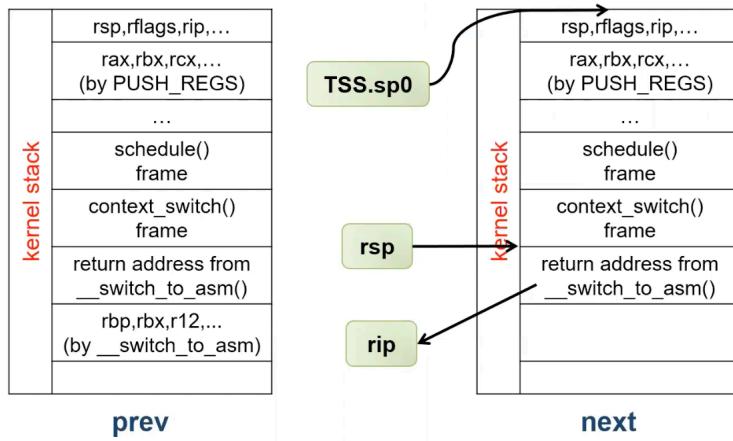
132

### הfonקציה () context\_switch

```
מצביע למתאר התהילך הנוכחי, שמופיע על המעבך
context_switch(..., struct task_struct *prev,
 struct task_struct *next, ...)
...
...
switch_mm(oldmm, mm, ...);
...
__switch_to_asm(prev, next);
...
}
המצביע למתאר התהילך הבא, שמקבל את המעבך
החלפת מרחבי הזיכרון
המאקרו ששמור את ההקשר
הנוכחי וטוען את ההקשר החדש
```

از מי שקרה לה זה *context\_switch* איז פקודת הבאה תהיה  
פקודת סי שורה אחריו אפשר לחשוב על זה כהמשך הקוד של תהליך  
*next* כי הוא עבר החלפת הקשר בעבר והוא קרא ל-*context\_switch* ה-*context\_switch* הוא צריך  
לבדוק אם ה-*rip* הינו בשלב הצב כי שרואים  
שלו ומשם הוא ימשיך וזה מה שרגם *rip* יחזיק בשלהב הצב כי שרואים  
פה.

### תמונה מצב אחריו () \_\_switch\_to()



:do\_fork

- קריית המערכת *fork()* משתמש בפונקציה פנימית של הגרעין  
הקרויה *do\_fork()* לבנית הקשר של התהילך החדש.

## מערכות הפעלה

133

- גם קריאות מערכת אחראות ליצירת תהליכיים (לדוגמה `clone()`)  
קוראות ל-`do_fork()`.
- הפקציה `do_fork()` מבצעת את השלבים הבאים:

- (1) מקצת *PCB* חדש ומחסנית גרעין חדשה עבור תהליך הבן.
- (2) קוראת לפקציה `copy_thread()`, אשר ממלאת את מחסנית הגרעין של תהליך הבן כך שייראה כאילו הוא קרא לקריאת המערכת `switch_to_asm()` וזו לפקציה `fork()` מעתיקת לתהליך הבן את מרבית הנתונים מ-*PCB* האב.
- (3) מושיפה את טבלת הקבצים הפתוחים (*file descriptors*)
- (4) ומשול את שגרות הטיפול בסיגנלים.

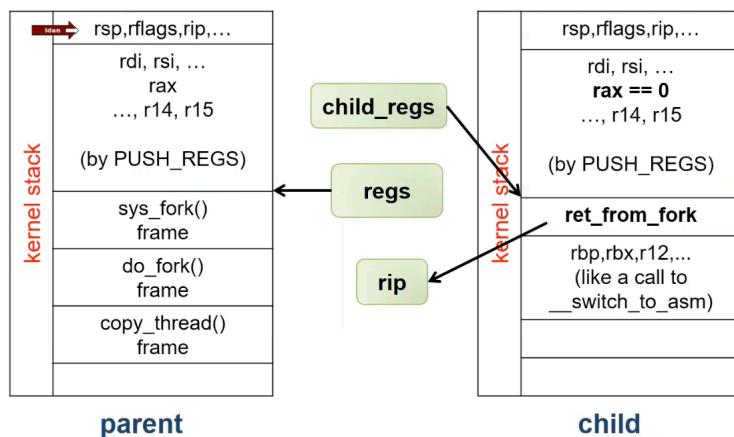
(5) העתקת תכולת הזיכרון מתבצעת בשיטת *COW*.

- (6) מקשרת את תהליך הבן ל"בני משפחתו".
- (7) מוסיפה את תהליך הבן לרשימת התהליכים הగזובאלית וגם לטבלת *PID → PCB* הערבול.

- (8) מעבירה את תהליך הבן למצב *TASK\_RUNNING* ומכניסה אותו ל-*runqueue*.

- (9) לסיום, הפקציה מחרירה את ה-*pid* של תהליך הבן, וערוך זה מוחזר גם מתהליך האב.

### הפקציה `copy_thread()`



## מערכות הפעלה

134

מה הפונקציה *parent, child copy-thread*? עושה עכשו יישן *fork*? להבדל מקודם שם היה *prev, next* ניתן לראות שבבסיס של מחסנית של האבא יש את הרגיטרים שהוא שמר אז *rsp, rflags...etc* וואז פתח מסגרת חדשה ל *sys\_fork* שהיא קראה ל *do\_fork* ושקרהה ל *copy-thread* שבעצם יוצרת את המחסנית של תהליך הבן ואז תהליך הבן מהנדסת אותו כך שהוא גם קרא ל *fork* בעצמו כי גם שהבן חוזר ממצב משתמש נראה שהוא חוזר מ אבל ההבדל בין אבא לבן שחזור מ *fork* זה ערך חזרה הבן מקבל 0 והאבא מקבל את ה *pid* של הבן ניתן לראות בתרשים למטה שאצל הבן הערך זהה הוא *rax* יש 0 וחוץ מזה אין נשמר על המחסנית של הבן את הרגיטרים.. *r12, rbp, rbx, r12..* כי אין זוכרים שעוברים למחסנית חדשה בהחלהת הקשר אנו שולפים אותם ואז שומרים כתובות חוזרת חזרה *ret\_from\_fork* שהיא הולכת ל*hishlq* לתוכו *rip* שאנו נעשה *ret* בסוף הפונקציה *switch\_to* שראינו קודם. לכולנו ברור *ret\_from\_fork* מחייב צריכה להיראות כך, מה *ret* אמורה לעשות? תעשה *pop* לרגיטרים ותחזור למצב משתמש. בעצם לקובד שבו בוצע את ה *fork*, עכשו השאלה היא זה למה היא מחסנית הבן רוצה להיראות כמו שנראית כי היא רוצה להיראות כמו מחסנית גראין לתהליך שעבור החלהת הקשר בעבר אז אם התהליך עבר החלהת הקשר בעבר הוא פעם כאילו שמר הרגיטרים האלה אז האבא יהיה אחראי לשים פב את הרגיטרים האלה ולשלוף אותם ואז לשים מעלייהם *ret\_from\_fork* כדי שנעשה החלהת הקשר לבן ישולף הפונקציה הזאת וקוד של האבא אחראי לשים ב *rax* את 0 כי שהבן ישולף את הרגיטרים ויחזור למצב משתמש הבן יחזיר מ *fork* עם ערך חזרה 0 והאבא והבן לא צריכים להיות זהים למשול הבן אין *sys\_fork, copy-thread* כל המסגרות האלה עצמן האבא כי הוא יוציא את המחסנית של הבן. נשים לב ש ב *ret\_from\_fork* היא פונקציה שעושה *pop regs* ששולפת כל הרגיטרים מעל כולל *rip* שמופיע למטה שומר את המקוד בקובד משתמש של הבן שאליו נחזור והוא יהיה אותו *rip* גם עצם האבא גם עצם הבן כי אין מעתיקים הרגיטרים מאבא של הבן.

## מערכות הפעלה

135

זכור ש `m_swtch_to_asm` ש��פקיד שלה לשלו. כל מה שנשמר

|  
תזכורת: מחסנית הגרעין

לאחר המאקרו  
**PUSH\_REGS**

| Kernel stack |
|--------------|
| ss           |
| rsp          |
| rflags       |
| cs           |
| rip          |
| orig_rax     |
| rdi          |
| rsi          |
| rdx          |
| rcx          |
| rax          |
| r8           |
| r9           |
| ...          |
| r15          |



על המחסנית.

על ידי סטרuktur שומר את כל הרגסטרים שנשמרים על המחסנית

**struct pt\_regs**

```
struct pt_regs {
 unsigned long r15;
 unsigned long r14;
 unsigned long r13;
 unsigned long r12;
 unsigned long rbp;
 unsigned long rbx;
 unsigned long r11;
 unsigned long r10;
 unsigned long r9;
 unsigned long r8;
 ...
 unsigned long rip;
 unsigned long cs;
 unsigned long flags;
 unsigned long rsp;
 unsigned long ss;
};
```

כתובות גבוחות

**struct pt\_regs\***

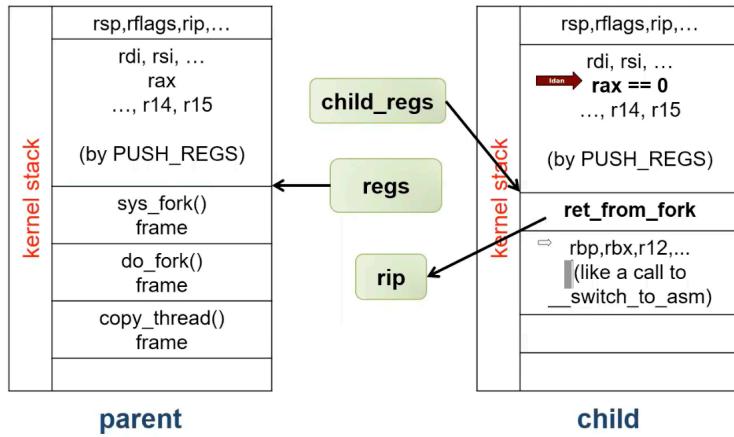
| Kernel stack |
|--------------|
| ss           |
| rsp          |
| rflags       |
| cs           |
| rip          |
| orig_rax     |
| rdi          |
| rsi          |
| rdx          |
| rcx          |
| rax          |
| r8           |
| r9           |
| ...          |
| r15          |

שהוא מוצבע על ידי `regs`, `child_regs` אבל למביעים את זה? כי העתקה של סטרuktur בשני שיעושים שובב בין סטרוקטים בסיסי ההשמה היא העתקה של כל השדות וכן מרוויחים העתרה של כל הרגסטרים בביטחון. השם `regs`, `child_regs` מופיעים פה:

## מערכות הפעלה

136

### הfonקציה () copy\_thread



### שלבי ה Fonקציה () copy\_thread

```
int copy_thread(..., struct task_struct * p, ...) {
```

מצביע ל-PCB של תחילת הפונקציה

- מוצאת את מבנה pt\_regs אצל תחילת הפון:

```
struct pt_regs* childregs = task_pt_regs(p);
```

- מוצאת את מבנה pt\_regs אצל תחילת האב:

```
struct pt_regs* regs = task_pt_regs(current);
```

- מעתקה את הרגיסטרים של האב לבן:

```
*childregs = *regs;
```

שלבי ה Fonקציה :  
copy\_thread

הסביר לך. ניתן לך ראות בתרמונה למעלה אבל יותר מאשר אחד שזה לשנות רגיסטר אצל הבן שהוא לא כמו אבא זהה *xax* א; ניגשים לרי

## מערכות הפעלה

137

דעתן 07 ונשים לובשrip לא משנה כי הבן ואבא חזרים

### שלבי הפונקציה () copy\_thread

- מעדכנת את ערכו של rax ל-0 בתחילת הבן:

```
childregs->rax = 0;
```

למה?

- מעדכנת את כתובת החזורה השמורה על המחסנית:

```
struct inactive_task_frame* frame =
 (struct inactive_task_frame*) (childregs) - 1;
frame->ret_addr = (unsigned long) ret_from_fork;
```

- מצביעת את p->thread.rsp לראש מחסנית הגרעין של הבן:

```
p->thread.rsp = (unsigned long) frame;
```

מ בקוד ואז יש חישוב לכתובת החזורה.

### הפונקציה :ret\_from\_fork

- פונקציה זו מופעלת כאשר תחיליק הבן מזמן לראשוña למבוד במהלך החלפת הקשר.
- המאקרו *POP\_REGS* שולף את כל הרегистרים מהמחסנית.
- אחר כך פקודה המכונה *sysret* קופצת חזרה לקוד המשתמש.
- למעשה, ביצוע הקוד ב *ret\_from\_fork* יגרום לסיום הקריאה(*fork()*).  
בתחיליק הבן עם ערך מוחזר 0.

```
ret_from_fork:
 ...
 POP_REGS
 sysret
 ↴
```

## מערכות הפעלה

138

חותמים.

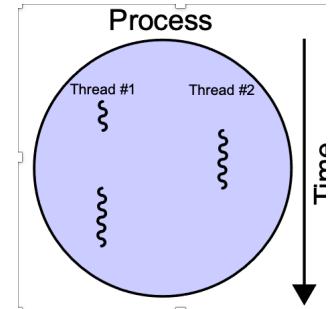
תקני. נושא חדש לגמרי שזה סנכרון ותכנות מקבילית. המוטיבציה  
הוא שיש מעבדים מרובי ליבות למשל בפלפון יש 4  
LIBOT, באמצעות נספר את הביצועים, למשל אחד הבעיות  
לחשוב עליה זה מין דרך – *sort* – *merge* – למשל  
נתן לכל חוט למין חלק מהמערך, ותתי שימושות אלה  
הם במקביל וכן ניתן לשפר ביצועים לכוארה אלו ידיעם  
אפשר לחתם למבקרים אחרים לעשות עוד עבודה למשל  
דרך תהליכיים אבל הבעייה שם לא משתפים זיכרון ולמה  
זה פחות נוח למשל בדוגמה המיין של המערך וכן הפתרון  
הוא חוטים, שהם בעצם תהליכיים קטנים בתוך תהליכי.  
כלחוט מבצע קוד בצורה עצמאית וכלם רואים את אותו  
זיכרון, והם יעזרו לנו לפטור תתי שימושות במקביל של  
אותה משימה גודלה של אותו תהליכי. כМОבן ששיטתן  
זכרון בין חוטים הוא זה שמאפשר להם להיות כל כך  
מוסילים אבל יכול ליצור בעיות, לתוכנת באמצעות חוטים  
זה לא כל כך קל והסיבה מרכזית היא היעדר אוטומיות  
בגישה למשתנים מסוימים. מה שקרה זה שחותמים לא  
מודעים אחד לשני והם ניגשים למשתנים מסוימים ויכולים  
להרוס אחד את השני, אלו נתמודד עם הבעייה הזאת באמצעות  
מנגנון שנקרו מנעולים.

- מעבדים מודרניים הם מרובי ליבות. איך אפשר לנצל אותם כדי  
לשפר ביצועים? לדוגמה, מין מערך גדול ע"י: חלוקה לשניים, מין  
כל חצי מערך בנפרד, ולבסוף מיזוג.
- תהליכיים לא משתנים זיכרון, וכך הם פחות מתאימים לתוכנות  
מקבילי. חוטים, (*threads*, לעומת זאת, פועלים למרחב זיכרון  
משותף. חוט הוא יחידת ביצוע עצמאית בתוך תהליכי ("Lightweight process").  
כל תהליכי יכול להכיל מספר חוטים שיורכו במקביל.

## מערכות הפעלה

139

- אבל שיתוף זיכרון בין חוטים יוצר גם בעיות, שאחת הנפוצות בהן היא: היעדר אוטומטיות בגישה למשתנים משותפים. נלמד איך להתגבר על הבעיה באמצעות מנעולים (mutex/spinlock).



```
void quick_sort(int a[], int length);

int* parallel_sort(int a[], int length) {
 int* b = (int*) malloc(length * sizeof(int));
 pid_t p = fork();
 if (p == 0) { // son
 quick_sort(a, N / 2);
 } else { // father
 quick_sort(a + N / 2, N / 2);
 wait(NULL);
 // now merge the two subarrays into b
 }
 return b;
}
```

למה המימוש הזה לא עובד?

זה: מין מקבילי של מערכ.

למשל בתמונה למטה שימוש תהליכים אינו ממש פיתרון למין. הקוד הוא דוגמא לטובה בשימוש בתהליכים, כמו שאמרנו תהליכים הם לא פיתרון לבעיה שדורשת שיתוף זיכרון. נשים לב שיש לנו פונקציה `parallel_sort` שמקבלת מערך  $A$  ואת הגודל שלו  $length$  ומקצת זיכרון ואז הבן ממיין חצי מערך הראשון ובaba חצי השני ואזABA מהכח שהבן מסיים וממזרג את מערך הגודל  $b$  שהקצינו. מה הבעיה פה? יכול להיות ששניהם ירצו על אותו מעבד ואז לא ממש שיפרנו ביצועים. נניח שהזה לא המצביע, אז מה היה קורא? אז מה שקרה זה שעושים `fork` נוותנים לבן עותק של תהליךABA אבל מאותו רגע הם העותקים נפרדים

## מערכות הפעלה

140

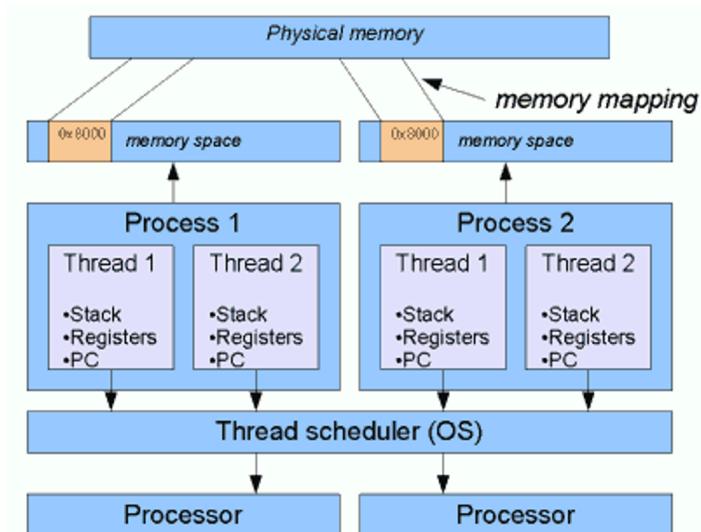
לగמרי אז הבן ימיין חצי מערך שלו וABA לא יראה את חצי מערך שהבן מימיין אלה יראה חצי מערך עליון שהוא מימיין כי הם חיים במרחב זיכרון נפרדים לוגמרי כי אזABA לא ידע לגשת למרחב זיכרון של הבן כדי למאגר. לפטור את הבעיה אפשר לעשות תקשורת בין תהליכיים למשל בpipe להעביר מידע מבן לאבא אבל זה רעיון פחות יעיל. لكن פיתרון טוב זה חוטים כי הם משתפים זיכרון ביניהם.

חוטים (*threads*). חוטים לא משתפים סט רסטרים שלהם, כל חוט מרגיש שיש לו סט רגיסטרים משלהו. זה מעניין כי מעכשו מה שscheduler עשוה ומזמן ל্‏רוצה זה חוטים ולא תהליכיים כלומר החלוקת הקשר גם קורותת בין חוטים, כלומר יכול לזמן חוטא של תהליך או תהליך אחר עם חוט אחד או כמה חוטים אבל מבחינת מערכת הפעלה היא יכולה לזמן למעבד חוטים.

- חוט הוא יחידת ביצוע בתחום תהליך. באנגלית נקרא גם: *Lightweight Process* בעברית נקרא גם: תהליכון.
- תהליך בלינוקס יכול לככלול מספר חוטים המשתפים ביניהם את כל משאבי התהליך: מרחב הזיכרון, גישה לקבצים והתקני חומרה, ועוד.
- למרות שהוט הוא רכיב של תהליך, כל חוט הוא יחידת ביצוע עצמאית שניית להריצתו על מעבד ללא קשר לשאר החוטים. כל חוט מהוות הקשר ביצוע נפרד - כל חוט מחסנית ורגיסטרים משלהו. ה-scheduler מזמן לריצה חוטים ולא תהליכיים.

## מערכות הפעלה

141



חותמים מול תהליכיים.

יש מה דיאגנומת שמתארת את העולם יש פה שני תהליכיים ולכל אחד יש מרחב זיכרון משלהו ולכל אחד שני חוטים וככל אחד מחותמים חי במרחב זיכרון עצמאי (מרחב זיכרון וירטואלי) ובוסףו של דבר כל מרחב ממופה למרחב זיכרון הפיזי שיש במחשב וכן שרואים *scheduler* מקבל כקהל חוטים ולא תהליכיים.

חותמים יכולים לשפר ביצועים.

- במערכת מרובת מעבדים: כל חוט יפותר חלק מהמשימה של אותו תהליך, והחותמים ירוצו במקביל על מעבדים שונים. אבל זה דורש לפרק את הבעיה לחתתי-בעיות בלתי תלויות - משימה לא טריוויאלית למתכנת...
- גם במחשב עם מעבד יחיד ניתן לשפר ביצועים באמצעות שימוש בריבוי חוטים: חוט אחד יכול לוויטר על המעבד (למשל כדי להמתין לנטונים מהרשთ) וחוט אחר ימשיך בביצוע חלק אחר של התוכנית.
- יתרון נוסף לחוטים: יצירת חוט זולה מעט מיצירת תהליך חדש, מפני שאינה כרוכה בהעתיקת משאבים כמו טבלת הדפים וטבלת הקבצים הפתוחים.

חותמים כموון יכולים לשפר ביצועים והסיבה היא שבמערכת מרובת מעבדים היינו רוצים שכל חוט יפותר חלק מהמשימה וירנו על מעבד אחרת

## מערכות הפעלה

142

במקביל לשאר חוטים וכן נפתחה הבעיה יותר מהר בתיאוריה זה נשמע טוב אבל יש כמה קורסים (תכנות מקבילי מבודח) ומערכות מבוזרות שמראות למה זה לא זה פשוט והסיבה היא שלא קל לפרק בעיה לחתמי בעיות בלתי תלויות למשל יש אלגורתמים שקל לחשב עליהם בצורה מקבiliar שליהם למשל  $BFS$  זהה חיפוש לעומק ברור לנו שאפשר לעשות את זה כחוטים שהולכים על כמה מסלולים שונים אבל  $DFS$  למשל לא ברור איך לעשות אותו בצורה מקבiliar גם אם ייתנו כמה מעבדים שרצו זה לא יהיה קל, גם אם יש מעבד אחד במערכת כדי להשתמש בחוטים כי חיטים למשל יכולם לקרוא לקריאות עצרכות חוסמות אז למשל אם חוט עושה קיאה לדיסק אז הוא ימתין וחוט אחר ימשיך לבצע פעולות אחרות.

מתי כדאי/לא כדאי להשתמש בחוטים? מתי כדאי?

אם יש לנו תוכנית חשובה מאוד כבده ש策רכה כמה שיותר זמן מעבד איז ברור שהיינו רוצים להשתמש בחוטים כדי לחתם לה להשתמש בליבות מעבד שונות למשל חיזוי מג אוויר או כל מטריצות זהה סימולציות פיזיקליות לצורכות המון הזמן המבוקש למשל חיזוי מג אוויר יכול להיעז שרצה שעה על מעבד אחד ואם יש 4 או 15 דק יקח עם ריבוי מעבדים אן להריץ משימב בלתי תלויות למשל להדפס מסמך בחוט שרצ בפרק בזמן ערכתו.

- תהליכי חישובים "כבדים" (משל חיזוי מג האויר) יכולם לנצל יותר ליבوت כדי לשפר את הביצועים.
- חוטים יכולם להריץ משימות בלתי תלויות, למשל הדפסת מסמך במקביל לערכתו.
- אפשרויות שרף יכולות ליצור חוט חדש לכל בקשה כדי לטפל במספר בקשות בו-זמןית.

מתי לא כדאי?

## מערכות הפעלה

143

הם תוכניות דyi קצורה זהה ממש לא ייעיל לתוכנית איז עדיף לוותר או למשל על מעבד יחיד שMRIICIM Chisobim CI AZ Yish TAKORA של החלפת הקשר.

- תוכניות קטנות ופשיות עלולות לסייע מתקורה מיותרת: עלות יצירת חוטים חדשים.
- תMRIICIM Chisobim במערכת מעבד יחיד עלולים לסייע מתקורה מיותרת: עלות החלפות הקשר.

הערה. החלפת הקשר בין חטימSEL אותו תMRIICIN היא יותר מהירה משני חוטים על תMRIICIM שונים CI משתפים אותו זיכרון AZ לא צריך לחמם מטמוניים בעצם תMRIICIM משתמשים במטמוניים של המעבד ואם חולקים אותו מרחב זיכרון AZ הם יכולים להנוט MAZHA בשימוש באותו מידע במטמוניים זה גם קשור ל nice, goodness שזה אלגוריתם זמן הישן שלמדנו בהרצאה למשמעות תMRIICIM וחוטים משולביםivid ביחס בתו הריצה וביחס המתנה AZ שוב תוך הריצה שומר רק חוטים מוכנים לריצה ומערכותיו אף פעם לא שומרים בתו הריצה תMRIICIN. עד היום למדנו שתMRIICIM היה להם חוט אחד ואין כזה דבר מהיות על תMRIICIF יש לפחות חוט אחד החוט הראשי ואז הוא נשמר בתו הריצה או המתנה. האם ב () fork נוצר חוט או תMRIICIN? נוצר תMRIICIN חדש שיוצרים לו חוט ראשי מיד.

תקשורת בין חוטים. השיתוף מיידע בין חוטים הוא קבוע בהם CI בתם משתפים מרחב זיכרון וכל אחד מריצ פונקציה AZ כל פעם שיוצרים חוט חדש חייבים להעביר לו את המשתנים המשותפים שעובדים עליהם בתו פרמטרים לחוט בזמן היצירה שלו או להשתמש במשתנים גלובלים זהה בדרך כלל מה שעושים ואז כל תMRIICIM יכול לגשת אליהם. שיתוף זיכרון גם חיסרון שלא ידרכו חוטים אחד על השני.

- חוטים צריים בדרך כלל לתקשר ולהחלף ביניהם מיידע. אמצעי התקשורת הוא פשוט ביותר: קריאה וכתיבה למשתנים משותפים.

## מערכות הפעלה

144

המשתנים המשותפים צריכים ליהיות גלובליים או ליהיות מועברים כפרמטרים לחוטים.

- משתנים משותפים הם גם חסرون: יש לחתם את הגישות אליהם על- מנת למנוע את שיבוש הנתונים.

- ישנו מצבים שבהם חוטים לא נדרשים לשתחוו בהם מידע.

- לדוגמה: חיפוש מילה ספציפית במספר גדול של קבצים. כל חוט יקרא קובץ וידפיס למשך את המופיעים של המילה בקובץ שהוא בדק.

- בעיות שלא דורשות תקשורת בין החוטים נקראות *embarrassingly parallel*

## ספריית *pthreads*

- בשנת 1995 הוצר תקן *POSIX threads* (*POSIX threads*), המגדיר אוסף טיפוסים ופונקציות המאפשרים עבודה עם חוטים במערכות תואמות *Unix*. הטיפוסים והפונקציות מוגדרים בקובץ *pthreads* (*/usr/include/pthread.h*) (ספרייה משתמש כמו *.libc*).

- כדי להשתמש בספריית *pthreads* יש להוסיף קובץ *header* בתחלת קוד :

```
include <pthread.h>
```

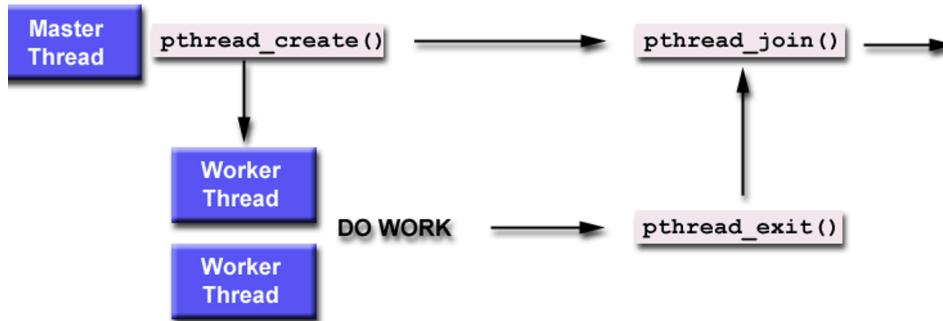
להוסיף את הדגל –*pthread* – במהלך ההידור

```
: gcc myprog.c -pthread -o myprog
```

הדגל מגדיר מספר *macros* ומקשר את התוכנית עם הספרייה הנחוצה.

## מערכות הפעלה

145



- שימוש לבב: שילוב בין תהליכיים לחוטים הוא אפשרי אך אינו מומלץ.

סכמת עבודה.

הfonקציות המعنיניות הן `create`, `join`, `exit` כדי ליצר חוט קוראים ו`fork` שהוא שקולה של `pthread_create` כМОון בשבל לנקוט תהליך חדש קוראים ל`wait` שפה זה `join` בהקשר של חוטים שממתינה `return` חוט וחוטים מסתירים מכם או 0 `exit` ייפעם

### דוגמה לייצרת חוטים

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void* print_m(void* ptr) {
 char* m = (char*)ptr; // cast the thread argument
 printf("%s PID = %d, pthread ID = %ld\n",
 m, getpid(), pthread_self());
 return NULL;
}

int main() {
 pthread_t t1;
 const char* m = "I'm Thread 1!";
 pthread_create(&t1, NULL, print_m, (void*)m);
 // we assume pthread_create() didn't fail
 pthread_join(t1, NULL);
 return 0;
}
```

מה ידפיס הקוד הבא?

יצירת חוט - דוגמא.

למעלה מוצר קוד סימני פשוט במרחב המשמש ואז מגדרים עוד פונקציה חדשה `m_print` שתהיה פונקציה של החוט החדש שהולכים ליצור

## מערכות הפעלה

146

ומה ש *main* עושה להגדיר חוט חדש היא מדירה 1 מטיפוס *t* מטיפוס *pthread\_t* שעד לא איתחולנו אותו ומה לחת לו לעשות, בפונקציה *pthread\_create* מקבלת מצביע ל משتمה 1 שיחזיק את המידע על החוט ועוד כמה ארגומנטים השלישי *print\_m* היא הפונקציה שהוחוט ירעץ ואחרון ארגומנט שהפונקציה תקבל נשים לב שפונקטייה של החוט מקבלת \* *void* ומחזירה \* *void* ולמה עובדת כך? בגלל להבטיח גנריות. נשים לב שחותם מרים רק פונקציה ולא קוד שלו. עוד מהו לכל החוטים יהיה אותו *pid* שהוא מסותף לכל החוטים כי ככל תחת אותו תהליך וכדי לקבל אותו משתמש ב *(pthread\_self()*. נשים לב שב *join* הפעמטר השני הוא עורך החזרה. והוא מעבירה *NULL* כלומר לא מעניין מה הוא מוחזיר.

```
- int pthread_create(pthread_t *thread,
 pthread_attr_t *attr,
 void* (*start_routine)(void*),
 void *arg);
```

יצירת חוט חדש.

- פועלה: יוצרת חוט חדש המבוצע במקביל לחוט הקורא בתוך אותו תהליך. החוט החדש מתחילה לבצע את הפונקציה המופיעה בפעמטר *start\_routine* וממת בסיום ביצוע הפונקציה.

• ערך מוחזר:

- o במקרה של הצלחה. כמו כן, מזהה החוט החדש נכתב למשתנה המוצבע ע"י *thread*.
- ערך שגיאה שונה מ-o במקרה של כישלון

• פערמטרים:

- *thread* - מצביע למקום בו יאותן מזהה החוט החדש במקרה של סיום הפונקציה בהצלחה.
- *attr* - מאפיינים המתארים את תוכנות החוט החדש, כגון אם החוט הוא חוט גרעין או חוט משותם, האם ניתן לבצע לו *join*, כלומר להמתין לסיומו, וכו'.

## מערכות הפעלה

147

- בד"כ נספק ערך *NULL* המציין חוט מערך שנitin להמתין לסיומו.
- *void \* (\*start\_routine)(void \*arg)* - מצביע לפונקציה שתהווה את קוד החוט. הערך המוחזר מפונקציה זו במקרה של סיום הטעבי הינו ערך הסיום של החוט.
- פרמטר שיסופק לפונקציה עם הפעלה - *arg* -

קובלחת מזהה החוט

*pthread\_t pthread\_self();*

- פעולה: מחרירה לחוט הקורא את המזהה שלו עצמו. מזהה זה הוא פנימי לספרייה *pthread* וaino קשור ל-*PID* של החוט.
- מתוך *the man page* :

*Thread identifiers should be considered opaque :*

*any attempt to use a thread ID other than in pthreads calls is nonportable and can lead to unspecified results."*

סיום חוט

*void pthread\_exit(void \*retval);*

- פעולה: מסיימת את פעולות החוט הקורא.
- ערך הסיום יוחזר לחוט שימתין לסיום חוט זה.
- פרמטרים: *.exit()* - ערך סיום (בדומה לזה של *retval*)

## מערכות הפעלה

148

הriegת חוט.

*int pthread\_cancel(pthread\_t thread);*

חוט אחד הורג חוט אחר

- פועלה: מסימנת את ביצוע החוט *thread* באמצעות סיגנל ייודי.
- ערך סיום הביצוע של החוט שנרג היה *PTHREAD\_CANCELED*.
- פרמטרים:
  - מזאה החוט המיועד לסיום. ערך מוחזר:
  - \* במקרה של הצלחה.
  - \* ערך שגיאה שונה מ-0 במקרה של כישלון.

המתנה לסיום חוט.

*int pthread\_join(pthread\_t thread, void\*\*thread\_return);*

- פועלה: גורמת לחוט הקורא להמתין לסיום החוט המזוהה ע"י *.thread*
- ניתן להמתין על סיום אותו חוט פעם אחת לכל היתר - ביצוע *pthread\_join()*
- כל חוט יכול להמתין לסיום כל חוט אחר באותו תהליך.
- המתנה על סיום החוט משחררת את מידע הניהול של החוט בرمת הספירה *threads* וברמת הגרען.
- פרמטרים:
  - מזאה החוט שממתינים לסיומו. לא ניתן להמתין ל"סיום חוט כלשהו" בדומה ל-*.wait()*
  - *thread\_return* - מצביע למקום בו יוחסן ערך הסיום של החוט עבורו ממתיינים.
  - ניתן לציין *NULL* כדי להתעלם מערך הסיום.

## מערכות הפעלה

- ערך מוחזר: *o* במקורה של הצלחה, וערך שונה מ-*o* במקרה כישלון.  
כמו כן, במקרה של הצלחה ערך הסיום נכתב למשתנה המוצבע ע"י *thread\_return* ).*NULL* (אם אין

הערה. חוט יכול לקבל סיגנלים גם בדומה לתהלהן אך זה לא שימושי או מעניין.

- חוט יכול להסתיים במספר דרכים שונות:

| אם התהלייר<br>מסתויים?                                                                  | אם החוט<br>מסתויים? | ס-בת הסיום                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------|---------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| לא בהכרח (רק אם החוט שהסתויים היה לאחריו)                                               | C                   | קריאה ל- <code>pthread_exit()</code> בתוך קוד החוט<br>קריאה ל- <code>pthread_cancel()</code> מחוץ אחר<br>חזקה מהפונקציה <code>start_routine()</code> (הפונקציה המבצעת של החוט) |
|                                                                                         |                     | קריאה לקריאת מערכת <code>exit()</code> ע"י חוט כלשהו<br>בקבוצה של החוט המזبور                                                                                                  |
|                                                                                         |                     | פעולה לא חוקית באחד החוטים<br>(למשל, חלוקה באפס)                                                                                                                               |
| <pre>int __libc_start_main(...)<br/>{<br/>    ....<br/>    exit(main(...));<br/>}</pre> | זיכרון:             | חזקה מהפונקציה <code>main()</code> של החוט הראשי<br>סקול לקריאה ל- <code>(exit())</code>                                                                                       |

## סיום חוטים ותהליכיים.

וחוט יכול להסתיים בהרבה מצבים לסירוגין חוט קודם כל חוט pthread\_rxit לסיים עצמו או לקבל cancel מחוט אחר שמשים אותו או אחריו ריצת הפונקציה שהוא מקבל כפרמטר שיועשה return. נשים לב שהתחלון מסתיים אם על החוטים בו מסתויימים, יש מקרים שבהם חוט מסוים אז גם התחלון למשל אם אחד מהחוטים קורא ל exit אז כל תהילון מסתויימים או למשל אם אחד מהם מחזק בו, או כל מיני שגיאות שגורמות לתחילון מסתויים, וגם אם הפונקציה main של החוט הראשי מסתויימת כי חזרה מ main שකוללה לקרוא ל exit כי היא עטופה כי exit עוטפת את pthread\_exit. כדי לסירוגין הוא ראשוני אפשר לעשות לו pthread\_main.

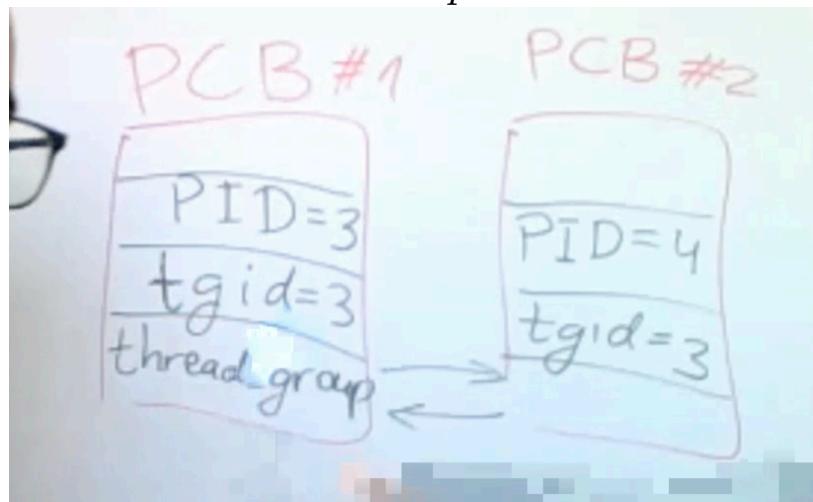
חוטים בגרעין לינוקס.

## מערכות הפעלה

150

- בגרעין לינוקס, חוטים ממומשים למשתמש כתהיליכים רגילים המשתפים ביניהם משאבים כגון זיכרון, גישה לקבצים וחומרה.
- כל תהליך נוצר עם חוט יחיד - החוט הראשי (*primary thread*)
  - באמצעות קריית המערכת(*fork()*).
- חוטים נוספים נוצרים באמצעות קריית המערכת(*clone()*). קריית מערכות זו היא הבסיס לתמיכת בחוטים.
  - בנגד לתהיליכים, אין קשרי משפחה בין החוטים.
  - אין חוט אב וחתוט בן.
- כל חוט יכול להמתין לסיום של חוט אחר כלשהו של אותו תהליך.
  - כל חוט יכול להרוג חוט אחר כלשהו של אותו תהליך.

קובוצת חוטים (*thread group*). נשים לב שעושים לוחוט אז אנו מקבלים את ב *tgid* של החוט. נשים לב שהחוט הראשי ה *tgid* שלו זה אותו דבר כמו ה *pid* שלו.



- לכל חוט, בהיותו תהליך רגיל, יש *PCB* משלהו ו- *PID*-*tgid* משלהו.
- עם זאת, המתקנות מצפה שלכל החוטים השبيחים לאותו תהליך נתן יהיה להתייחס דרך *PID* יחיד - של התהליך המכיל אותם.

## מערכות הפעלה

151

- פועלת () *getpid* תחזיר את אותו *PID* בכל החוטים של אותו תהילין.
- קריאות מערכת כמו (*kill*) הפעולות על ה-*PID* של התהילין צריכות להופיע על כל החוטים בתהילין.
- לכן, לינוקס מאחדת את כל החוטים של תהילין מסוים לקבוצה חוטים (*thread group*) כדי שאפשר יהיה להתייחס אליהם יחד
- השדה *tgid* במתאר התהילין מכיל את ה-*PID* המשותף לכל החוטים באותה קבוצה.
  - למעשה, זהו ערך ה-*PID* של החוט הראשון (הראשי) של התהילין. חוטים חדשים יקבלו ערך *PID* חדש וערך *TGID* זהה לחוט הראשון. קריאת המערכת (*getpid*) מחריצה למעשה *tgid* את *current*  $\rightarrow$  *tgid*.
- השדה *thread\_group* במתאר התהילין (*task\_struct*) הוא ראש הרשימה המקושרת של כל החוטים באותה קבוצה.
- פעולות על ה-*PID* המשותף מתורגמות לפעולה על קבוצת החוטים המתאימה ל-*PID*.
  - למשל: *kill(pid, SIGKILL)* הורגת את כל החוטים *tgid == pid* העורם

סיכום: *PID* מול *TID*

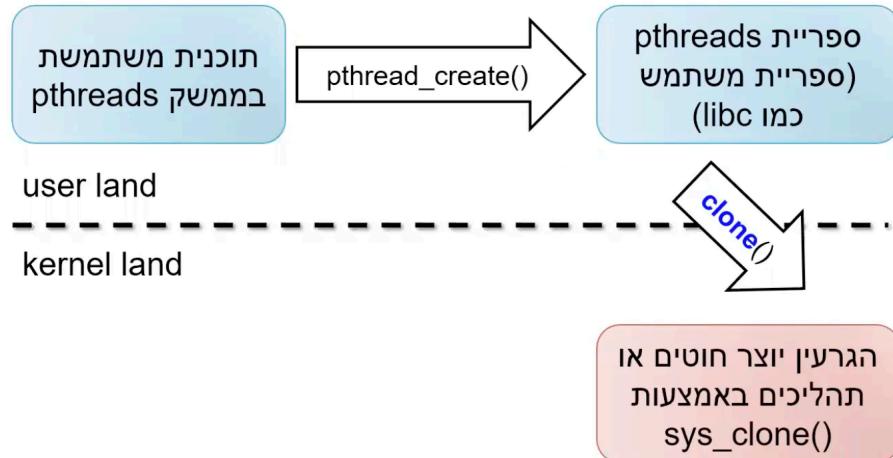
- המשתמש ומערכת ההפעלה משתמשים במסתכלים על חוטים באופן שונה:
  - תהילין מרובה חוטים מורכב מחוט ראשי וחוטים משנהים המרכיבים תחנת *PID* ייחיד (שהוא בעצם גם ה-*TGID*).
    - ניתן לקבל *TGID* זה באמצעות (*getpid*).
  - לכל חוט מסוימת נספּן הנקרא *TID* או *Thread ID* או *PID* האמתי של החוט.
    - ניתן לקבל *TID* זה באמצעות (*gettid*).
- מערכת ההפעלה:

## מערכות הפעלה

152

- מסכמת על כל חוט וחותם כהליין נפרד, בעלי *PID* משלה עצמו. קיימת "שירות גורלוות" של כל התהליכיים המרוכזים תחת אותו *TGID*.

## תרשים: חוטים במבט מערכת



נשים ליב לתרשים למעלה שיש קו מקווקו שמאפירד בין מרחב משתמש לגרעין מה שלמעלה הוא רץ בהרשאות נמוכות  $cpl == 3$  וכל מה שלמטה רץ בהרשאות נמוכות, נשים ליב ש *pthreads\_create* היא פונקציה כמו *printf* והוא קוראת ל *clone* שסומומשת בתוך קוד הגרעין.

### קריאה המערכת () clone

*int clone(int(\*fn)(void\*), void\*child\_stack, int flags, void\*arg);*

די דומה ל *pthread\_create* אבל לא בדיק לא אותו דבר. מקבל גם דגלים שמצוורפים בטבלה.

- פועלה: יוצרת תחליין בן המשתק עם תחליין האב משאבים ונתונים לפי בחירה.
- פרמטרים:

## מערכות הפעלה

153

- $fn$  - מצביע לפונקציה שתהוו את הקוד הראשי של התהילין החדש.
- $arg$  - הפרמטר המועבר לפונקציה  $(fn$  בתחילת ביצוע התהילין החדש.
- כשמבצע הפונקציה  $fn(arg)$  מסתים, נגמר התהילין החדש.
- $child_stack$  - מצביע לראש המחסנית של התהילין החדש.
  - תזכורת: המחסנית גדלה לכיוון הכתובות הנמוכות.
  - לכן  $child_stack$  עירך להצביע לסופם בлок הזיכרון המוקצה לטובת המחסנית
- $flags$  - מסכת דגלים הקובעת את צורת השיתוף בין התהילין הקורא והטהילין החדש. להלן מספר דגלים אופייניים:

|                                                                                                                   |                     |
|-------------------------------------------------------------------------------------------------------------------|---------------------|
| שיתוף מרחב הזיכרון                                                                                                | <b>CLONE_VM</b>     |
| שיתוף טבלת הקבצים הפתוחים                                                                                         | <b>CLONE_FILES</b>  |
| שיתוף טבלת נתוני עבודה עם קבצים, המכילה נתונים כגון ספרית העבודה הנוכחיית ועוד                                    | <b>CLONE_FS</b>     |
| לטהילר החדש יהיה אותו אב כמו התהילר הקורא (אחרת החדש יהיה הבן של הקורא)                                           | <b>CLONE_PARENT</b> |
| התהילר החדש הוא חוט באוותה קבוצת חוטים כמו התהילר הקורא (אותו <a href="#">tgid</a> ). גורר גם <b>CLONE_PARENT</b> | <b>CLONE_THREAD</b> |

מספר דגלים יחד באמצעות  $|$  או  $OR$  לוגי ביניהם, לדוגמה:  $CLONE_VM|CLONE_FS$ :

- ערך מוחזר: במקרה של הצלחה מוחזר ה- $PID$  של התהילין החדש, אחרת 1.

מיושן קריית המערכת ( $clone()$ : בתוך הגרעין,  $sys\_clone()$ ) שמשתמש בשירות בפונקציה ( $do_fork()$  עליה למדנו בתרגולים קודמים, ומעבירה לה את הדגלים על מנת לקבוע לכל משאב אם לשתח אותו או ליצור אותו מחדש.

## מערכות הפעלה

154

### תכנות לrogramma

הfonקציה () f תרצה  
1000 פעמים  
במקביל, ובכל פעם  
תملא איבר נוסף  
של המערך.

מה יהיה ערכו של  
[999] a בסיום  
התוכנית?

```
#include <pthread.h>
#include <stdio.h>
#define N 1000
int i = 0;
int a[N];

void* f(void* arg) {
 a[i] = i;
 i++;
 return NULL;
}

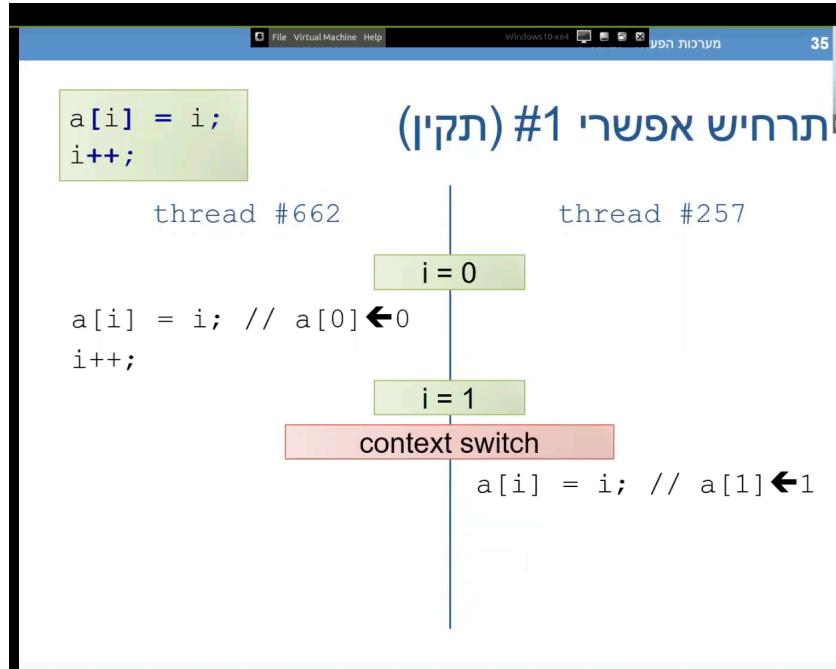
int main() {
 pthread_t threads[N];
 for (unsigned int i=0; i<N; i++)
 pthread_create(&threads[i], NULL, f, NULL);
 for (unsigned int i=0; i<N; i++)
 pthread_join(threads[i], NULL);
 printf("a[999] = %d\n", a[999]);
 return 0;
}
```

קובץ, באמצעות חוטים.

מה הבעיה בשימוש בחוטים. למשל למעלה יש לנו פונקציה *main* ויש מערך בגודל 1000 שהוא גלובלי מה ש *main* עושים זה ליצור 1000 חוטים בלולאה וכל אחד מוכניס לו פונקציה ממלאת מערך, במקומות ? ומקדמת את המשנה ב 1 אובי צה עוד בלולאה *pthread\_join* לוחכות לכל תהליכיים אלו שישתיימו.

## מערכות הפעלה

155



תרכיש למעלה נותן תוצאה תקינה. אבל; ה לא בדיק המצביען  
פלט התכנית לדוגמה

```
>> gcc -pthread -O2 main.c
>> ./a.out
a[999] = 999
>> ./a.out
a[999] = 999
>> ./a.out
a[999] = 0
```



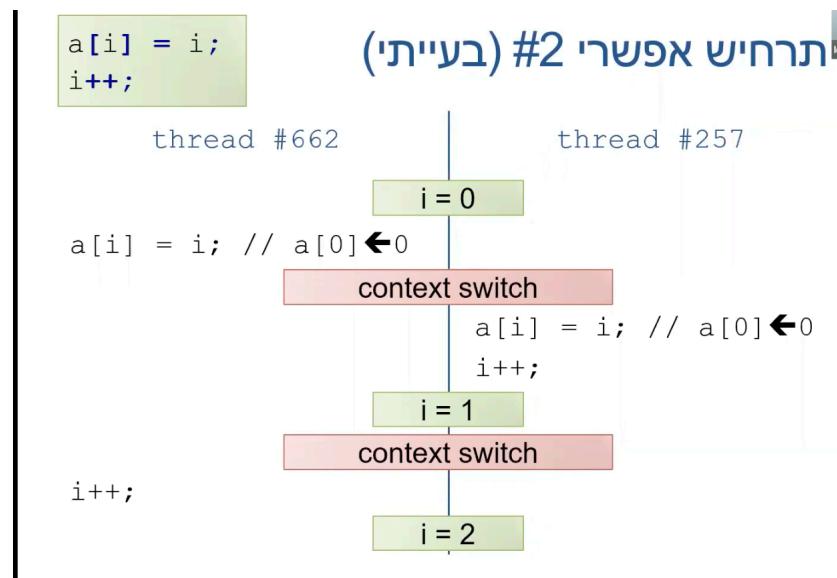
אוויז! התוכנית אינה דטרמיניסטית הרי קיבלנו תוצאה לא מצופה.  
(מצב זה נקרא *race condition*). כי הפלט שיוצא ממש צלוי בסדר  
של ריצת חוטים שמערכת הפעלה אחראית עליהם, אך חיבים לבנות  
תוכנית שאין בה פאג זהה.

## מערכות הפעלה

156

מה קורה כאן?

- החלפת הקשר בין החוטים יכולה לקרות בכל נקודת זמן, בפרט באמצע הפונקציה של חוט מסוים.
- פלט התכנית תלוי בתזמון של החוטים ובסדר בו הם מתבצעים - מצב שנקרא *race condition* (תנאי מרוץ).
- מכיוון שאנו לא שולטים בתזמון של תהליכיים (או חוטים), תכנית המCALLה *race condition* נחשבת תקולה (*buggy*).
- יותר דיק, תכנית צאת היא בעלת התנהגות לא מוגדרת.
- קשה לדבג תכניות כאלה כי קשה לשחזר את ההתנהגות הביעיתית.



מה שקרה פה זה שני חוטים קידמו את *i* ומילאו אותו מערך וכך בעצם *a[1]* יכיל ערך זבל.

אות *C* בשורת *race condition*

- שימושו ליב: החלפת הקשר יכולה לקרות גם באמצע שורת *C*, כי הקומpileר עשוי לתרגם שורת *C* אחת למספר פקודות אסמלבי.
- לכן, *race condition* יכול לקרות גם במקומות לא צפויים.
- לדוגמה, הקומpileר יכול להדר את השורה `++` לאחר האסמלבי

הבא

## מערכות הפעלה

157

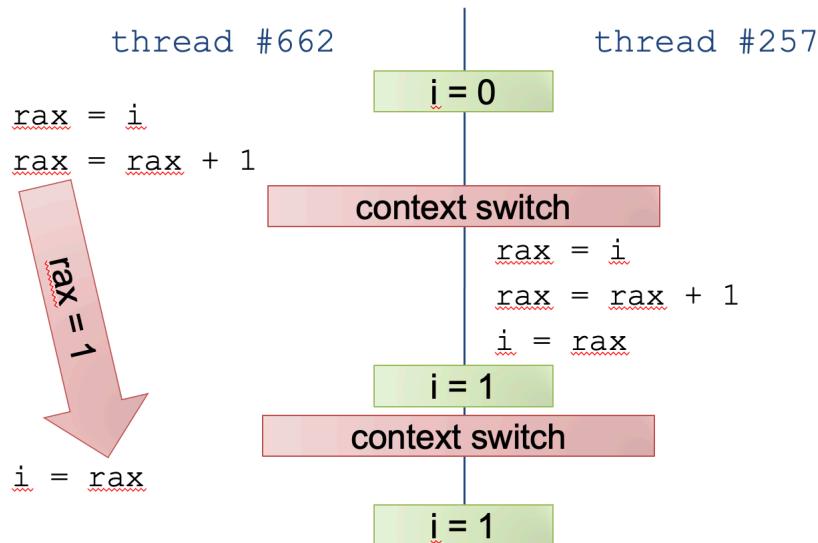
f:

```
...
 movl (%rbx), %rax
 addl $1, %rax
 movl %rax, (%rbx)
...

```

**גואידו-קוד:**  
`rax = i  
rax = rax + 1  
i = rax`

**תרחיש אפשרי #3 (בעיתי)**



נניח שיש לנו שני חוטים וכל אחד מרים  $i + 1$  מה נציג החלפת הקשר בנקודת הכיוון לא טובה שיכולה לקרוות חוט ראשון קודם כל קורא  $i$  לרגresar  $rax = rax + 1$  ואז  $rax = rax$  כלומר  $rax$  יוכל עכשו 1. עכשו קرتה החלפת הקשר במקום הכיוון לא טוב, עכשו 0  $i = rax = 1$  ואז  $rax = rax + 1$  ואז  $rax = rax = 0$  ואז  $rax = rax = 1$  ואז עושים  $i = 1$  ואז עושים שוב החלפת הקשר וכך עכשו 1  $i = 1$  כי נשים לב לפני החלפת הקשר ראשונה שומרנו כל הרגסטרים במחסנית הגרעין כולל  $rax$  וכך הרצינו פעמיים  $i + 1$  ואז  $i = 1$  כלומר חוט אחרון שרצף לעולם לא יגיע 999 במקרה זה, דהיינו זה

## מערכות הפעלה

158

- תרחיש די לא סביר ואסור לקרות בחים אמיתיים. נשים לבי שמשתנים גלובליים בסי תמיד מאוחלים ב 0 אוטומטי גם מערכיים. קטע קרייטי.

- קטע קוד הניגש למשאב משותף (למשל משתנה בזיכרון).
- ביצוע קטע קרייטי במקביל עלול לגרום להתקנות לא רצויה.
- תכנית תקינה צריכה להבטיח מניעה הדדיות - לפחות חוט אחד יריץ את הקטע הקרייטי בכל רגע נתון.

- במקרה אחרות: הפקדות בקטע הקרייטי צricsות להתבצע בצורה אוטומית ביחס לגישות אחרות למשאב המשותף - או שלל הפקדות בקטע הקרייטי ירצו יחד ויסטימנו, או שהן לא ירצו כלל.

```
void* f(void* arg) {
 a[i] = j;
 i++; }
 return NULL;
}
```

בדוגמה הקודמת: שתי השורות  
האלו הן קטע קרייטי

בדוגמה קודמת פונקציה *f* ניגשה למשתנים משותפים זהה קטע שצריך לזרע בצורה אוטומית. איך אפשר להבטיח שרק חוט אחד יריץ קטע קרייטי? מנעולים.

מנעולים. כדי לפטור את זה משתמשים במנעולים נחשוב על זה בטרמונוילגיה של דלת שמוגנת עם מנעול ובתוך המנעול יש מפתחץ יש לנו חדר ואני רוצים להטיח שרק בן אדם אחד בחדר ברגע נתון (כמו שירותים), שאם רואן מפתח במנעול הוא נכנס פנימה והעל מבפנים ואז אף אחד לא יוכל להיכנס ושהוא מסיים הוא יוצא ושם מפתח במנעול לטובת חוטים הבאים ולמנעול יש שצי פונקציות *lock*, *unlock*.

- מנעולים הם אחד המנגנוןים הבסיסיים למימוש מניעת הדדיות.
- האנלוגיה: קטע קרייטי חדר עם דלת מוגנת ע"י מנעול עם מפתח בפנים.

## מערכות הפעלה

159

- כדי להכנס לחדר (הקטע הקרייטי) צריך לנעול את המניעול ולשים את המפתח בכייס.
- ביציאה מהחדר יש לפתוח את המניעול ולהשאיר את המפתח בדלת (לטובת החוטים האחרים).
- שימוש לב:
- בכל רגע נתון, לכל היוטר חוט אחד יכול לתרוף / להחזיק / לנעול את המניעול.
- רק החוט המחזיק במניעול אמור לשחרר אותו (בעלות על המניעול).

## מניעולים pthreads

- בתקן *pthreads mutexes* מניעולים נקראים *mutual exclusion* (מניעה הדדית).
  - קיצור של *mutex* (鎖, מנגנון).
- נתקן את הדוגמה הקודמת בעזרת נעילה.

```
pthread_mutex_t m;

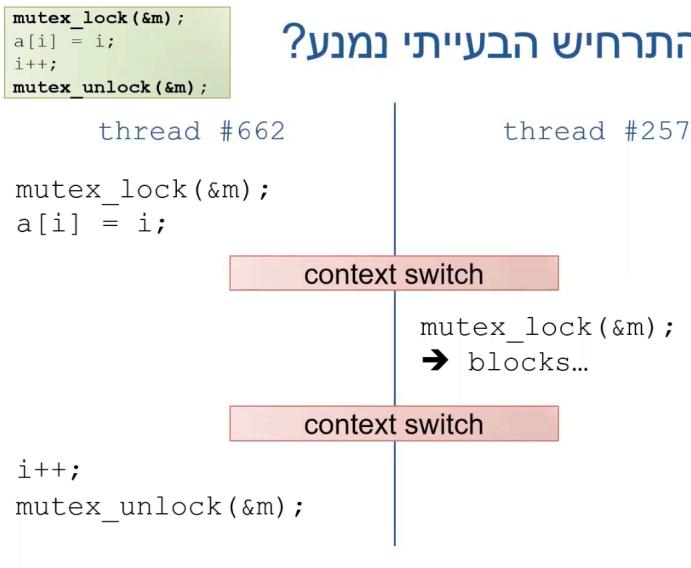
void* f(void* arg) {
 pthread_mutex_lock(&m);
 a[i] = i;
 i++;
 pthread_mutex_unlock(&m);
 return NULL;
}
```

- אם המניעול אינו נעול, החוט נועל אותו ונכנס לקטע הקרייטי.
- אם המניעול כבר נעול, החוט נחסם עד אשר המניעול ישוחרר.

## מערכות הפעלה

160

אי התרחיש הבעייתי נמנע?



הערה. המנוול חייב להיות גלובלי כי ככלם חייבים לאותו מנוול.

מנועלים עם/בלוי המתנה. *mutex*

- כאשר חוט מנשה לتفسס מנוול נועל, מערכת הפעלה תעביר אותו לתוך המתנה ותבצע חילפת הקשר.
- כאשר המנוול ישוחרר, מערכת הפעלה תuir את אחד החוטים המחייבים למןעוול.
- יתרון: תהליך חדש יכול לדרוש מיד, וכן לא מתבזבז זמן מעבד יקר. עדיף כאשר זמן המתנה המשוער גבוהה יחסית (כגון, כאשר הקטע הקריאה ארוך, כפי שקרה לרוב בקוד משתמש)

*spinlock*

- כאשר חוט מנשה לتفسס מנוול נועל, הוא בודק את ערך המנוול ללא הפסקה ולא מוותר על המעבד. טכניקה כזו נקראת גם: *polling*, *busy looping*, *busy waiting*.
- יתרון: התהליך יכול להמשיך לדרוש עוד בקצבנותם הנוכחיים, וכן לחסוך את התקורה על החילפת הקשר. עדיף כאשר זמן המתנה

## מערכות הפעלה

161

המשוער נמוך יותר מהמחיר של החילוף הקשור (כליום, כאשר הקטע הクリיטי קצר, כדי שקורחה לרוב בקצב גרעין).

נבחין בין שני טכניקות לנעילה, *mutex*, *spinlock*. ב *mutex* מה שעושים זה הבא, קודם כל אם חוט מנסה לסתום מנעול שנעול מערכת הפעלה מעבירה חוט זהה לתוך המתנה ועשה החילוף הקשור ושמנעול ישחרר מערכת הפעלה תעביר אותו מתוך המתנה, בשיטה זו אנו לא מבזבזים זמן מעבד, לעומת *spinlock* שימושיים להתקדנד בין חוטים (אולי נחזור על חוט שכבר בדקנו) עד שיגדו לנו לא. טכניקה זאת נקרא *busy waiting* והיא טוחנת זמן מעבד וחיסרון שלו היא לא משחררים המעבד לטעות חוטים אחרים מה שהיינו רוצחים שיקריה זה שאמ מנסים לסתום מנעול והוא נעה אין טעם למתין בכניסה לקטע קרייטי אז נשחרר את המעבד לשאר חוטים שירוץ ויישחררו את המנעול. אז זה שימושי כי לא נתקע בלולאה אינספורת כי תמיד יש חוטים שרוצים על מעבדים שונים אז חוט שרצ על ליבת אחרת הוא זה שיישחרר את המנעול.



### פעולות על מנעולי mutex

#### • גnewline mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

• הפעולה חוסמת עד שה mutex מתפנה ואז נעלמת אותו.

#### • ניסיין לנעילה mutex:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

• הפעולה נכשלה אם ה mutex כבר נעול, אחרת נעלמת אותו.

#### • שחרור mutex נעל:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

• ניסיין לשחרר מנעול שאינו נועל תביא להתנהגות לא מוגדרת.

#### • פינוי mutex בתום השימוש:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

• הפעולה נכשלה אם ה mutex לא מואתחל אבל נעול.



## ATCHOL MENUL mutex

```
#include <pthread.h>
int pthread_mutex_init(
 pthread_mutex_t *mutex,
 const pthread_mutex_attr_t *mutexattr);
```

- פרמטרים:
  - mutex – המנעל עליו מבוצעת הפעולה.
  - mutexattr – מגדרת תכונות mutex.
  - NULL – עברו סוג ברירת המחדל.
  - ניתן לקרוא על סוגים נוספים ב-pages.man.
- ערך מוחזר: 0 בהצלחה, אחר כישלון.

שימוש תקין במנעלים.

- בקוד תקין רק החוט שמחזיק במנעל הוא זה שמשחרר אותו.
- לעומת זאת, הפעולות הבאות יובילו להתקנות לא מוגדרת:
  - (1) נעליה חוזרת ע"י החוט שמחזיק במנעל.
  - (2) שחרור המנעל ע"י חוט שאינומחזיק במנעל.
  - (3) שחרור מנעל שאינו נעול.
- דיבוג קוד עם מנעלים הוא מטגר יותר ברגל שהקוד כבר לא>DTRMINISTI – בחילק מהריצות יש באג, בחילק מהריצות אין באג.
- יש כלים שעוזרים בעיה, לדוגמה *helgrind* או *sanitizers*.

שימוש של מנעל.

פקודות מכונה אוטומיות.

- פקודת מכונה אוטומית (*atomic operation*) היא פקודה המעדכנת את מצב המערכת (מעבד+זיכרון) בצורה אוטומית.

## מערכות הפעלה

163

- כלומר מעבדים אחרים אינם יכולים לראות מצבם בינויים של ביצוע הפקודה ואיינם יכולים לעדכן את מצב המערכת תוך כדי ביצוע הפקודה.
- במערכת מעבד יחיד: כל פקודת מכונה היא אוטומית מכיוון שהיא אינה יכולה להיקטע ע"י פסיקה.
- במערכת מרובת מעבדים: פקודת מכונה אוטומית חייבת לנעול את ערוץ הגישה של כל המעבדים האחרים לזכרון עד לסויומה.
- הנעה מתבצעת באמצעות הוספת קידומת *lock* לפקודה. למשל: *x lock; inc x*; היא פקודת מכונה המבצעת  $x++$  בצורה אוטומית.
- פקודת מכונה אוטומית מגירה למעשה קטע קריטי באורך פקודה אחת, וכן מאפשרת להימנע משימוש במגעול.
- בנוסף, פקודות מכונה אוטומיות מאפשרות למשם בצורה פשוטה יחסית אמצעי סינכרון כמו מנעולים.
- לדוגמה: ארכיטקטורת  $64x$  מציעה את פקודת (*bit test and set*), *BTS*(*bit test and set*), אשר מדליה בית מסויים ברגיסטר או כתובת בזכרון ומחזירה את ערכו הקודם בדגל *CF* ברגיסטר הדגלים.
- פונקציית הגרעין (*test\_and\_set\_bit()*) משתמשת בפקודת המכונה *BTS* כדי להציג משתנה ולהחזיר את ערכו הקודם בצורה אוטומית.

אמרנו החלפות הקשר יכולת לקשר בפקודות אSEMBLY וכן אפשרות *lock; inc* פקודות מכונה שהן מתחומות (אוטומיות) למשל אם נעשה *x* או *lock; inc* אז זאת פקודת שתבוצע בצורה אוטומית מבחינת גישה לזכרון שיש לה מנגן חומרתי שמריע אותה בצורה אוטומית ופקודת מכונה אוטומית היא קטע קריטי של פקודת אחת כלומר היא חוזרת יותר מאשר הפקודות. למשל למעלה *x lock; inc* היא מקדמת את *x* בצורה כך שמעבד שועל *x* הוא זה שיכל רק לגשת לזכרון באותו רגע זה הוא שעשה הנעה והעביד אחר לא יוכל לעשות זאת (זה ימנע מקטעים קריטיים).

## מערכות הפעלה

164

```
typedef struct lock {
 bool is_locked;
} lock_t;

void init(lock_t* l) {
 l->is_locked = 0;
}

void lock(lock_t* l) {
 while (test_and_set_bit
 (l->is_locked));
}

void unlock(lock_t* l) {
 l->is_locked = 0;
}
```

### סעיף ב'

- כתן מימוש באמצעות הפקודה האוטומטית `test_and_set_bit()` המدلילה ביט ומוחירה את ערכו הקודם.
- האם המימוש מבטיח מונע הרעה?
- כן. ההוכחה בעזרת נפנוי ידיהם...
- האם המימוש מונע הרעה?
- לא. ניתן דוגמה נגדית:
  - שני חוטים A,B רצים לסדרוגון.
  - חוט A תמיד תופס את המנעול לפני סיום הקוווטם שלו.
  - חוט B תמיד מכליה את הקוווטם שלו בהמתנה למנעל.

```
typedef struct lock {
 bool is_locked;
} lock_t;

void init(lock_t* l) {
 l->is_locked = 0;
}

void lock(lock_t* l) {
 while (l->is_locked);
 l->is_locked = 1;
}

void unlock(lock_t* l) {
 l->is_locked = 0;
}
```

השאלה

נשים לביות שפתרון ראשון לא ממש מניעול כי ברגע שני חוטים רצים על שני מניעולים במקביל ינעלו ואז שניהם יכנס לკטע קרייטי ואז התווצה לא תהיה צפואה. לשם כך בפתרון השני השתמשנו בפקודה אטמית מתורגם לשפט מכונה מה שעושה היא שברגע שנכנס אליה חוט היא משנה את הערך ומוחירה ערך קודם למשזה נניה שחותך ראשון נכנס הפ את ערך שם ל-1 כלומר נעל המניעול ואז יוצאים מה `while` כי הפקציה תחזיר 0 אחרי זה שעוד חוט ייכנס(אולי במקביל גם) הערך שMOVED ל-0 זה 1 אז הוא ישאר ב `while` לנצח עד שחותך ראשון ישחרר מניעול ואז יהפול ל-0. ובג卡尔 שהיא אוטומית כלומר מובטח שלא יהיה החלה הקלפה. הקשר תוכה לכך מובטח שמנגרון יעבד.

תרגול של סנכרון.

מימוש של מנעול דרך סימפור.

זכור שמנעל זה מגנו סנכרון שאמור להבטיח שרק חוט אחד יוכל לעבור את הקטע הקרייטי. אם נתחל למשל את ה סימפור ל 0 אז מה יהיה ב lock, unlock?

נתחיל למשל מ  $sem = 1$  וואז מה יהיה ב lock, unlock?

נשים למשל wait בתוך מימוש lock ובתוך מימוש של unlock נשים post. הבחירה הזאת לעובד למה? כי אפשר לחשב על זה בהרבה דרכים. דרך ראשונה יש לנו מנעל ואתחלנו את ה  $sem = 1$  עכשו חוט ראשון שמנסה לעשות נעליה למנעל עושה lock ואז wait ומוריד הערך של sem ל 0 ומיד נכנס לקטע הקרייטי. עכשו החוט שני שמנסה לעבור את הקטע הקרייטי יעשה lock ואז יגיע ל wait עכשו החוט הוא 0 לכן היא מעבירה את החוט השני לתור המתנה. ברגע שהחוט ראשון מסיים קטע קרייטי הוא עושה unlock שימושה על ידי post ואז היא מעירה את החוט השני מטור המתנה, והוא נכנס לקטע הקרייטי. נניח שאתחלנו את  $sem = 10$

עכשו המשמעות זה כמו מנעול מוכל (מה הבעה?) כי כמה חוטים יכולים להגיע  
לקטע קרייטי, אבל למשל לפעמים זה מה שנרצה נניח למסעדה שבה לא יכולה  
להכנס 10 אנשים בו זמן תואם אז במקרה הזה יכנסו חוט 1 ויעשה wait ויוריד sem  
ב1, יכנס עוד חוט ועוד פעם יוריד ב1, ואז יכנסו עוד ועוד עד שיגיע ל0. והלוקה  
11 ישינה לכטム למסעדה ייריץ lock ואז יתקע. ברגע שהוא אחד מס'ים הוא יעשה  
post וגרום לחוט השני להיכנס. لكن ניתן לעשות דרך סימפור ביןארי מנעול. או  
סימפור מוכל. נשים לב שמנעול ממומש דרך סימפור לא ממש דומה לmutex  
שראינו כי بما שראינו רק מי שעושה lock יכול לעשות לעצמו unlock לעותמת  
המימוש דרך סימפור שכל אחד מהחוטים יכול לעשות lock ו unlock ז"א אם חוט  
תפס מנעול ונעול אז חוט אחר יוכל לעשות לו unlock. בתור מדתמשים חייבים  
להשתמש בסימפור בזהירות כי למערכת הפעלה ממש לא אכפת. בכלל זאת למה  
כדי להשתמש במנעול?

## דוגמיה: סטפור בטור מנעול

```
sem_t sem;

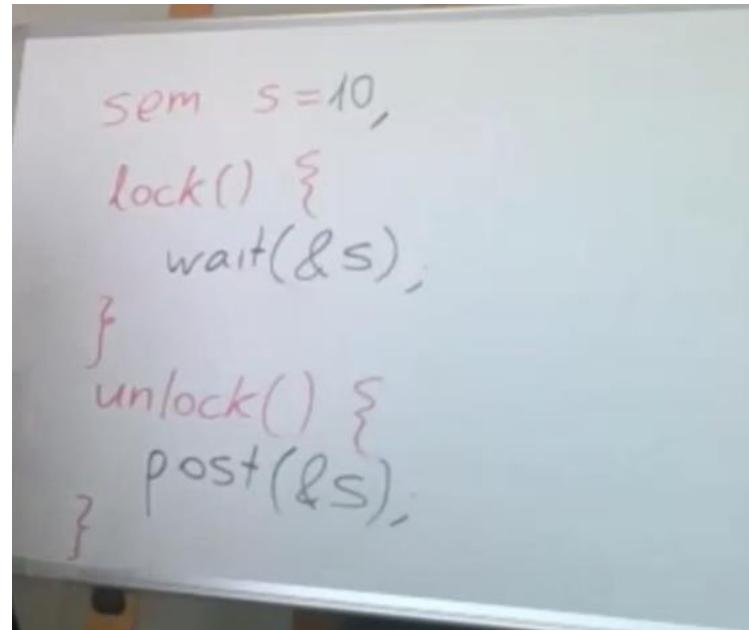
sem_init(&sem, 0, 1)

sem_wait(&sem);
// critical section
sem_post(&sem);
```

- סمفורה עם ערך התחלתי 1 נקראת סمفורה בינהריה.**

**סמפור בינהריה יכול לשמש להגנה על קטע קרייטי (ע"י מניעה הדדית בין החוטים הניגשים).**

**דגש: סمفורה בינהריה שונת מתגיאול mutex, משומש שכלי חוט יכול לבצע post על סمفורה, גם אם לא ביצע wait על הסمفורה קודם לכן (אין "בעלויות" על הסمفורה).**



נחזיר לדוגמא של הרשימה המקשורה על enqueue, dequeue לראות כמה זה נוח למשם עם סימפור.

**דוגמה: סימפור להבטחת סדר**

אם נאתחל את הסימפור ל-0, החוט השני ייכחה לראשונה.

השימוש בסימפור פשוט יותר מאשר במשתנה תנאי כי () post של סימפור לא הולך לאיבוד.

```

mutex_t m;
sem_t queue_size;
sem_init(&queue_size, 0, 0);

void enqueue(item x) {
 mutex_lock(&m);
 /* add x to tail */
 mutex_unlock(&m);
 sem_post(&queue_size);
}

item dequeue() {
 sem_wait(&queue_size);
 mutex_lock(&m);
 /* remove from head */
 mutex_unlock(&m);
}

```

אמרנו שמנעל זה סימפור שמאתחל ל-1 ומוכלל זה חמספר חיובי כלשהו, והאפ הוא מאתחל ל-0 אז עוזר לנו לעשות סדר. למשל נחזיר לדוגמא ונאתחל את הסימפור ל-0

שהוא ישמור לנו מספר האיברים ברשימה, כעת אפשר לשים לב שסימלור פותר לנו את הבעיה שהיא היה איבוד סיגנלים בצורה די עדינה ואלגנטית. למשל ניתן לראות ש ב `dequeue` חוט לא יכול להיכנס לעולם אף הרשימה ריקה, לעומת מה שהוא מקודף שחוות היה צריך להיכנס ואז לנעול מעול ולעשות `while` והרבה משתני תנאי, כל זה נחסך על ידי ה `wait` של הסימפור פשוט אם יש שם 0 בסימפור אז הוא ייכה עד שימושו יכנס ויעשה `post` ואז יעיר את החוט ההוא שנמצא בראשימת המתנה, ואז ינעל ויעשה הקטע קרייטי ויישחרר. למעשה יש פה סדר די ברור, ולא צריך להתעסק עם באגים. לכן יותר נוח להשתמש בסימפור ממשתני תנאי. למה בכלל זאת השימוש במנעולים? כי יכול להיות שם לא להשתמש במנעול יכנסו כמה חוטים לקטע הקרייטי עברו ערך סימפור גדול מ 1 ואז יש לנו בעיה הרי הם יכולים לdroos לאחד את השני, או להכנס ביחיד איבר ואז יד בעיה די גדולה. לכן מנעול זה חובה כדי לשמר על סדר ולהבטיח ש רק חוט אחר נכנס לקטע הקרייטי.

**דוגמא: מימוש מנעול קוראים כתובים.**

## מנעול קוראים-כותבים

ldan

- מנגנון סנכרון המאפשר להגן על מבנה נתונים באופן בו-זמני.
- מספר חוטים יכולים לקרוא את המידע (ambil' לשנות אותו) בו-זמני.
- כאשר חוט רוצה לעדכן את המידע, הוא צריך גישה בלבד למבנה הנתונים.

- לדוגמה, כדי להגן על משתנה  $x$  הנגיש ממספר חוטים:

| reader thread                                | writer thread                                      |
|----------------------------------------------|----------------------------------------------------|
| read_lock();<br>$y = 2*x;$<br>read_unlock(); | write_lock();<br>$x = 5*x + 1;$<br>write_unlock(); |

- בשקפים הבאים נדגים כיצד ניתן למשתמש במנעול קוראים-כותבים באמצעות משתני תנא.
- בהרצה ראיתם איך להשתמש בסמפור למטרה זו.

נרצה למשתמש בניו נתונים שרק חלק מהחווטים קוראים ממנו וחלק כותבים אליו. בשביל כך נדרש נעלמה, למשל הנעה למעלה קצר קשוחה מדי, אפשר לעדן אותה. אפשר לחת לקוראים להכינס ביחד הם לא משפיעים כלום, הם פשוט קוראים ולא משנים שום מצב. אבל אם אחד שרוצה לכתוב או הוא היחיד שרוצה להיות בקטע הקרייטי. אסור לנו מצב שבו קורא עם כותב, הכתיבה אמורה להתבצע אך ורק על ידי חוט אחד בלבד. למשל פה ניתן לראות שיש רק משתנה  $x$  אבל מה אם היה עז שרוצים לכתוב ולקרוא ממנו. במקרה זה, אם כותבים וקוראים יחד, אז יכולים להיות שני חוות נני אחד כותב שהוא מוסיף לעת איבר או משנה ערך של צומת ספציפי ותוך כדי חוט אחר מהפץ בעז או מה שקורא זה שקורא ערך לא מעודכן וזה לא מה שרצו.

מיושן של קוראים כותבים. (באמצעות משתני תנא)

## מנעול קוראים-כותבים

- ממשו מנגנון קוראים-כותבים בעזרת משתני תנאי ומנעולי mutex בלבד (בשונה מהIMPLEMENTATION שראיתם בהרצאה באמצעות סטפסורים).
- יש לממש את 5 הfonקציות הבאות:
  - 1 reader\_lock()
  - 2 reader\_unlock()
  - 3 writer\_lock()
  - 4 writer\_unlock()
  - 5 readers\_writers\_init()

---

### IMPLEMENTATION (1)

```
int readers_inside, writers_inside;
cond_t read_allowed;
cond_t write_allowed;
mutex_t global_lock;

void readers_writers_init() {
 readers_inside = 0;
 writers_inside = 0;
 cond_init(&read_allowed, NULL);
 cond_init(&write_allowed, NULL);
 mutex_init(&global_lock, NULL);
}
```

למעלה יש לנו 2 משתני תנאי שזהם(read\_allowed, write\_allowed) ששנייהם אחד אומרים אם ניתן לכתוב והשני אם ניתן לקרוא. ויש פה מנגנון global\_lock

כזכור משתנה תנאי תמיד בא עם מנעול. מה שנעשה בתזוז זה אתחול, ובהתחלה אין לנו קוראים או כותבים لكن שמיים 0. נתחיל עם `read_lock` מה אמורה לעשות? על איזה תנאי אמורה להמתין? אנו נמאמ למתנה אם יש כותבים.

## שימוש מנעול קוראים-כותבים (2)

```
void reader_lock() {
 mutex_lock(&global_lock);
 while (writers_inside > 0)
 cond_wait(&read_allowed, &global_lock);
 readers_inside++;
 mutex_unlock(&global_lock);
}

void reader_unlock() {
 mutex_lock(&global_lock);
 readers_inside--;
 if (readers_inside == 0)
 cond_signal(&write_allowed);
 mutex_unlock(&global_lock);
}
```

קודם כל `reader_lock` תופסת את המנעול הגלובלי, ובודקת אם יש כותבים, אם כן יש או יוצאים למתנה על מנתה התנאי `read_allowed`. וברגע שעברנו לה `while` אוז יודעים שיש קורא בפנים, لكن מגדילים מספר הקוראים ב 1 ואוז משחררים את המנעול. מה היה קורה אם לא היו משחררים? אוז זה היה מפר וזה שיש כמה קוראים יכולים לקרוא במקביל. אני רוצים לאפשר כמה חוטים לעבור את ה `reader_lock` או עוד לפני ש `reader_unlock` יכול לבוא עוד `reader_lock` ויעבור אותו וזה מה שהוא חשוב. קלומר בין `reader_lock` `reader_unlock` יכול להיות פעולה ארוכה של חיפוש ביןארי ואוז נרצה כמה חילושים לקרות בו זמן וכאן מה שהוא חשוב לנו שכמה חוטים יעברו את `reader_unlock` וזה כרגע מתקיים במימוש הזה, כל חוט יכנס ויתפס מנעול ויבדוק

את התנאי ויראה שאין כותבים ואז מחרר מנעול גלובלי ואז עוד reader יכול לכנס. בכל זאת למה צריכים מנעול? כי אז יכול להיות שני קוראים נוכנים ביהם ואז הערך של reader\_inside לא ברור אז אנחנו כןאפשרים שיכנסו קוראים במקביל אבל לא בו זמנית. מה reader\_unlock עשו? באופן דומה, היא מורידה ב 1 reader\_inside ובודקת אם הוא 0 אם כן אז אין קוראים لكن מעירה כותב אחרת משחררים המנעול כך שעוד קורא יכול לכנס.

### IMPLEMENTATION-KORAIM-COTBIM (3)

```

void writer_lock() {
 mutex_lock(&global_lock);
 while (writers_inside + readers_inside > 0)
 cond_wait(&write_allowed, &global_lock);
 writers_inside++;
 mutex_unlock(&global_lock);
}

void writer_unlock() {
 mutex_lock(&global_lock);
 writers_inside--;
 if (writers_inside == 0) {
 cond_broadcast(&read_allowed);
 cond_signal(&write_allowed);
 }
 mutex_unlock(&global_lock);
}

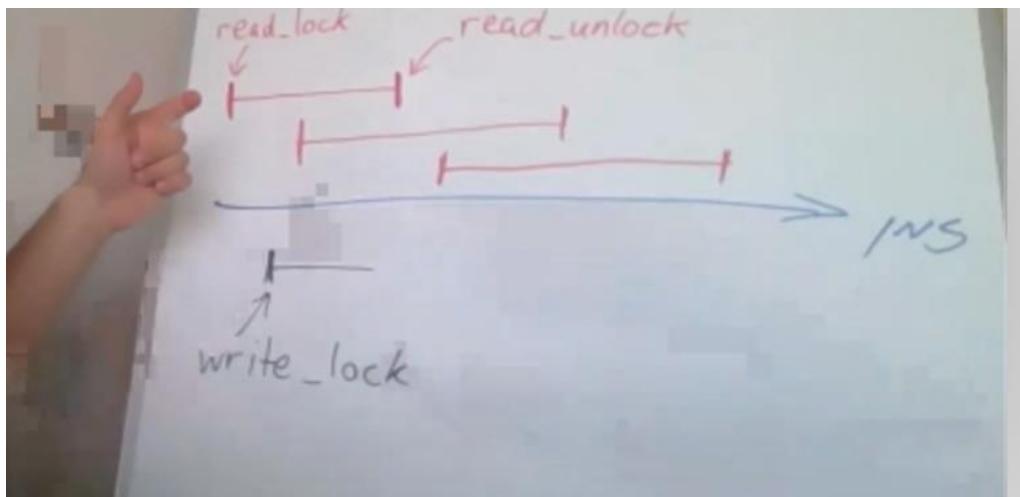
```

נסתכל על write\_lock היא בודקת אם כל עוד יש קוראים או כותבים יצא להמתנה כי יש כבר קוראים או כותבים שרצו לרווח ואז שוחזרים מהמתנה (מיישו כנראה קורא מעיר החוט שנכנס עכשו מהמתנה) מגדילים את מספר הכותבים ומשחרר מנעול. עכשו writer\_unlock תופסת המנעול ומקטינה מספר כותבים ואם הוא מגיע ל 0 אז היא מעירה כותב אחד וכל הקוראים, נשים לב ש cond\_boradcast ומעירם את כל הקוראים כי רוצים לעיר יכול שיכנסו ומעירם רק אחד מהכותבים כי אפשר רק כותב אחד בו זמנית. אם היינו עושים cond\_boradcast ל

האם זה באג? לא, כי בסופו של דבר בwriter\_lock רק אחד יוער והשאר יכנס להמתנה. האם צריך את ה if? לא! כי אם עושים writer\_unlock אז לכל היותר הwriter\_inside יהיה 0 או 1 אז אם עושים writer\_unlock אז נראה שזה הכותב היחיד במערכת והוא זה שמשחרר את המניעול ומוריד אותו ל0 לכן הif תמיד יתקיים כי תמיד יתקיים אז זה לא כזה משנה. האם הwriter\_lock ב while? כן, כבר רأינו שזה חשוב קודם (כי יכול להיות שכן העירו אותו לפני אבל זה לא אומר שכרגע זה נכון, אנו צריכים שייעירו אותו כרגע ולהוכיח עד שזה יקרה מחווטים שבאים בעתיד כי אם הוא יכנס אז יהיה לנו שני חוטים שמוחקקים בתור המקבilia וזה בעיה רצינית).

## חרונות של המימוש

- **הרעות כתבים וחוסר הוגנות:** כל עוד המניעול אצל הקוראים, קורא חדש שmagiu יוכל להיכנס ויעקוף כתבים שהגינו לפניו.
- **חוסר סדר:** לא ניתן לדעת האם הקוראים או הכותב יכנסו לקטע הקritis.
- תלוי מי יוכל לתפוס ראשון את המניעול global\_lock\_global שמשחרר בסיום writer\_unlock().
- איך אפשר לפתור בעיות אלו?



ניתן לראות שקורא נכנס שוב ושוב והכותב מורעב, אבל אנו רוצים הוגנות. אז ניתן מימוש משופר שפותר את הבעיה.

## מימוש מניעול קוראים-כותבים (1)

```

int readers_inside, writers_inside, writers_waiting;
cond_t read_allowed;
cond_t write_allowed;
mutex_t global_lock;

void readers_writers_init() {
 readers_inside = 0;
 writers_inside = 0;
 writers_waiting = 0;
 cond_init(&read_allowed, NULL);
 cond_init(&write_allowed, NULL);
 mutex_init(&global_lock, NULL);
}

```

הוספנו משתנה חדש שנקרא **writer waiting**, כך שאם לנו כותב ממתין אנו לא ניתן לקוראים להיכנס, כמוון נתחל אותו ל-0.

## מועד א', אביב 2008, שאלה 1

```
void reader_lock() {
 mutex_lock(&global_lock);
 while (writers_inside > 0 || writers_waiting > 0)
 cond_wait(&read_allowed, &global_lock);
 readers_inside++;
 mutex_unlock(&global_lock);
}

void reader_unlock() {
 mutex_lock(&global_lock);
 readers_inside--;
 if (readers_inside == 0)
 cond_signal(&write_allowed);
 mutex_unlock(&global_lock);
}
```



מוקודם קוראים היו ננסים חופשי, עכשו אנו לא רוצים לעשות את זה. קלומר אם יש כותב ממתין או אף כותב לא יכנס אלא יצא להמתנה.

## א', אביב 2008, שאלה 1

```
void reader_lock() {
 mutex_lock(&global_lock);
 while (writers_inside > 0 || writers_waiting >
 cond_wait(&read_allowed, &global_lock);
 readers_inside++;
 mutex_unlock(&global_lock);
}

void reader_unlock() {
 mutex_lock(&global_lock);
 readers_inside--;
 if (readers_inside == 0)
 cond_signal(&write_allowed);
 mutex_unlock(&global_lock);
}
```

נשים לב ש ה writers\_waiting נעדכן אוטם אך ורק ב writer\_lock כלומר תופסים מנעול וברגע שתופסים אותו אז זה אומר שאין כותב ממתין ואז קוראים לא יכולים לכנס כי הם רואים שיש כותב ממתין ואז כותב בודק את התנאי ונניח שמתקיים אז באיזשהו שלב יעירו אותו והוא לא ממתין אז מוריד 1 מ writers\_writing. האם קורא יכול לכנס באמצעות ב writer\_lock ב''י writer\_waiting ו ++writer\_waiting התשובה לא כי אנחנו תחת מנעול שתופסים אותו.

נניח שיש לנו 3 חוטים 2 קוראים מואחד כותב, קודם, כותב כל כניסה הכתוב ואו הממצינים  
הפק ל 1 לאן אם כניסה קורא או הוא נהייה במתנה לקורא ואז כניסה עוזנקורא הוא  
בודק ה while הבדיקה מצילהה לנכון מכניס אותו ומגדיל הכותבים בפנים ואז משחרר  
המנעל, עכשו עושה lockatch וואז תפוס ממועל ומקטין כותבים בפנים ומעיר את  
הכותבים, שוב הכותבים לא יכולים להכניס כי יש אחד בפנים, לנכון מעיר את  
הקוראים שהיו במתנה ואז מסתירים הריצה.  
יש פה טריד אוף, נכוון שעכשו כן מריעבים את הקוראים אבל הם יכולים לרווח  
במקביל וזה טוב.

**סינכרון בגרעין לינוקס.**

## אנלוגיות המסעה



- דמיינו מסעה ובה המלצר מטפל בשני סוגים של קוחות:

| לקוחות VIP                                                                    | לקוחות רגילים                                                                                                                |
|-------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------|
| המלצר מטפל מיד בכלLKUCHOT_VIP שmagiu, גם אם צריך לעזוב באמצעות LKUCHOT_REGIL. | אם המלצר פניו ומגיע לKUCHOT_REGIL, אז המלצר עובר לטפל בו.                                                                    |
| LKUCHOT_VIP לעולם לא ישחרר את המלצר שמטפל בו כרגע.                            | LKUCHOT_REGIL יכול לשחרר את המלצר שמטפל בו כרגע לטובת LKUCHOT_ANOTHER.                                                       |
| המלצר לא יעזוב LKUCHOT_VIP לטובת LKUCHOT_REGIL.                               | המלצר יכול לעזוב LKUCHOT_REGIL לטובת LKUCHOT_VIP שmaguiim. לאחר הטיפול בלקוחות VIP, המלצר יכול לחזור לטפל בלקוחות REGIL אחר. |

## הגרעין הוא מלצר

- הגרעין מטפל בשני סוגים בקשות (לקוחות):

| פונקיות חומרה                                                                                            | קריאות מערכת / חריגות                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| הגרעין מטפל מיד בכל פסיקת חומרה שmagiu, גם אם הוא באמצעות טיפול בחיריה / קריית מערכת / פסיקת חומרה אחרת. | אם המעבד מריץ קוד משתמש ומגיעה קריית מערכת / חריגת גז הגרעין עובר לטפל בה.                                                                                  |
| שגרת טיפול בפסיקת חומרה לעולם לא תוותר על המעבד.                                                         | קריאות מערכת יכולות לוותר על המעבד, לדוגמה (), wait().                                                                                                      |
| שגרת טיפול בפסיקת חומרה לעולם לא תקרה לקריאת מערכת או תיצור חריגה (למעט חריגת דף - page fault).          | הגרעין יכול לסתוע טיפול בקריאת מערכת / חריגת לטובת פסיקות חומרה שmagiuot. לאחר הטיפול בפסיקות החומרה, הגרעין יכול לעבור לטפל בתהילך אחר מזה שרצה קודם קודם. |

נשים מערכת הפעלה אפשר להחשב עליה כמלצר והוא רק משרתת בקשורת או פסיקות חומרה מה שחשוב להבין שיש כמה חוקים, אם המעבד נגיד מריצ' קוד משתמש ו Mageira קריאת מערכת אז הוא מטפל בה. כמובן אם גם אם מגיע פסיקת חומרה אז גם מטפל בה, אבל ההבדל הוא שפסיקת חומרה יכולה לקטוע קריאת מערכת או חריגה, ההפק לא יכול ל��רות, כמובן לא ניתן שבטיבול בפסקת חומרה תגיע קיראת מערכת כי אנחנו כבר בגרעין והוא לא יוזם קריאת מערכת. (רק קריאות משתמש). נשים לב שפסיקות חומרה אך פעם לא יכולה יותר על המעבד לעומת קריאות מערכת. לעומת זאת הכרעין כן יכול לקטוע קריאות מערכת לטפל בפסקות. האם פסיקת מקלדת גורמת לקריאת מערכת? ממש לא, פסיקת מקלדת מה שעשו היא כותבת ל buffer של המקלדת בגודל מסוים, אבל אם אף תהליך לא יעשה write מהמקלדת (נזכיר שהוא מחוברת לתיבת כניסה מס' 0 בטבלת FDT ואם אף תהליך לא יעשה write מהמקלדת זה שיגיע פסיקת מקלדת זה לא יעניינו אותנו) דוגמא קנוונית לכך אם פותחים שולחן עבודה וכותבים משהו אז כלום לא יקרה, אלא אם כן הם מנוטים למשהו שהוא תהליך וממתין אליהם. ואם סלאן מקלידים הם ישארו ב buffer של המקלדת.

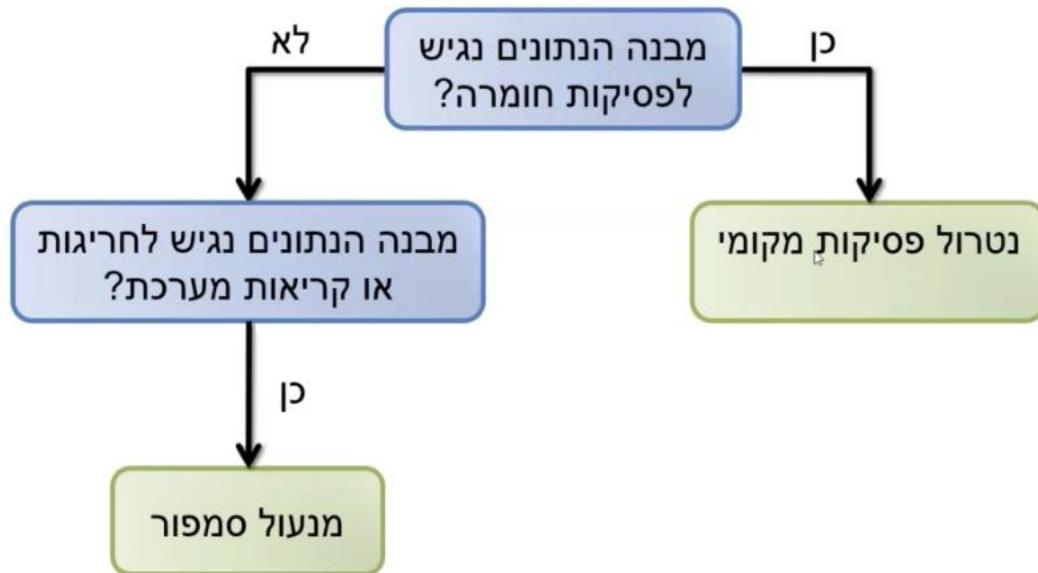
למה זה חשוב?

## איך זה קשור לביעות סנכרון?

- נסתכל על התרחיש הביעיתי הבא:
  - במסעדה יש ערכת תה אחת המורכבת ממספר חלקים (קנקן, כוסות, ...).
  - ללקוח רגיל נכנס למסעדה וմבקש תה.
  - המלצר מתחילה לעבוד ומגיש ללקוח את הקנקן.
  - לפטעה נכנס ללקוח VIP וגם מבקש תה. המלצר כמושן ניגש לשרת אותו מיד.
  - המלצר מעביר לו את הכוסות, אבל הקנקן עדיין אצל הלוקו הקודם.
  - כל לקוח מחזיק חלק מהערכתה בגל שהמלצר לא הביא אותה **בצורה אוטומטית**.
- ההקללה לגרעין המשרת פסיקות: **בעית אוטומיות בגישה לששתנים משותפים**.

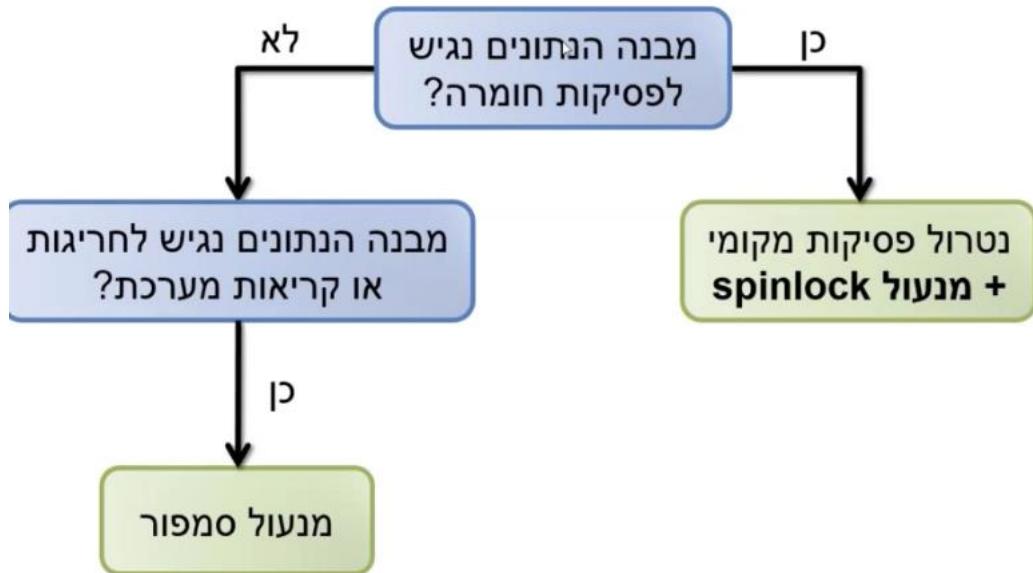
למשל אם יש תרחיש שבו גרעין מטפל בקריאה מערכת על מעבד אחד, ובאמצעע קריאה מערכת מגיעה פסיקת חומרה שמנסה לגשת לאותם נחונים של קריאה מערכת, שעלול לגעת באותו מבני נתונים. אפילו אם יש כמה מעבדים, נניח שמריצים אותה קריאת מערכת מהתהליכים השונים על מעבדים שונים ואז קריאת המערכת **fork** ש策ריכה להוסיף לתור הריצה אז יכול להיות שני מעבדים שונים שעושים Fork בו זמנית לשני התהליכים שונים ושניהם מנסים להוסיף דברים לתור הריצה ואם לא תהיה נעליה אז יהיה לנו בעיות בכך חייב להיות לנו אוטומיות כי אז תור הריצה יכול להשתבש.

## אמצעי הגנה במערכת מעבד יחיד



יש לה דיאגרמה שמתארת את הפיתרון לבעה, למשל אל אני ובעבד יחיד אז היא עשויה סדר. למשל אל מבני הנתונים נגיש לפסיקות חומרה או היא ייבת לנטרל פסיקות מקומי אחרה הוא רק נגיש לחריגות וקריאות מערכת אז משתמשים בסימפור (בثور מנעול).

## אמצעי הגנה במערכת מרובת מעבדים



אם יש לנו מרובה מעבדים או משתמשים ב נטרול פסיקות מקומי ו `spinlock`.

## פסיקות חומרה במערכת מעבד יחיד

- פסיקת חומרה חדשה יכולה להגיע תוך- כדי ביצוע טיפול בפסיקת חומרה אחרת (קינון פסיקות חומרה).
- פסיקת חומרה יכולה להגיע גם תוך כדי טיפול בחירה.
- לבסוף, יש להגן על מבני נתונים הנגישים לפסיקות חומרה באמצעות מנעולים. בפועל, נעלמה היא בעיתית במערכת עם **מעבד יחיד**. נדגים באמצעות התרחיש הבא:

  - מסלול בקלה #1 מטפל בפסיקת חומרה כלשהי ומחזיק מנעול.
  - פסיקת חומרה אחרת מגיעה ומתופלת מיד במסלול בקלה #2 (אשר קוטע את מסלול בקלה #1).
  - מסלול בקלה #2 מנסה לתפוס את המנעול, ולכן הוא ממתיין לסיום מסלול #1.
  - אבל גם מסלול בקלה #1 ממתיין לסיום מסלול #2 לפני שיחזור לרצף.
  - **קיבלו deadlock**.

נשים לב שפסיקות חומרה יכולות להיות מקווננות זאת אומרת שתוך כדי שמטפלים באחת יכולה להגיע עוד אחת (שמטופלה אותה דבר) אז מה הבעיה לפתור דרך מנעול? אסור פה ולמה? כי נניח שיש לנו קוד טיפול בפסקת חומרה בגרעין שהוא מחייב מנעול כי נכנס לקטע קריטי תוך כדי שהוא בקטע קריטי מגיעה פסקה חומרה שגמ רוצחה לכטע קריטי, אם מנסה לפרק את המנעול אז היא חייבת להמתין עד שקטע קודם יסתהים אבל הוא לא יסתהים עד שמטפלים בפסקת חומרה החדש שהגיעה לכן יש לנו פה deadlock. (שני קטעי קוד ממתיינים אחד לשני) לכן הדרך למנוע את זה פשוט לעצור פקודות לומרה מקווננות בקטעים מסוימים ככלומר אם הגרעין בקטע קוד של פסקת חומרה והוא חייב להגן על משתנה משותף. אז הוא מנטרל פסיקות מקומיות ככלומר מנטרל מטרל כל הפסיקות שmagiuot על המעבד המקומיי ככלומר מקרים הדגל F ברגסטר הדגלים של המעבד. (נשים לב שהפסיקות מנטרלות נעלמות לעולם אלא יועברו לעתיד שהמעבד יקבל פסיקות).

לכן אנו מנטרלים לאורך זמן די קצר, אבל צריך להבין שהן פיתרון לא מספיק למרובה מעבדים כי הם יכולים להריץ אותה פסקה חומרה בו זמנית וכך הנטרול של הפסיקות הוא מקומי על אותו מעבד זה לא מספיק צריך גם לנטרל פסיקות מקומיות וגם לתפוס מנעול.

ניהול זיכרון.

תרגול 9.

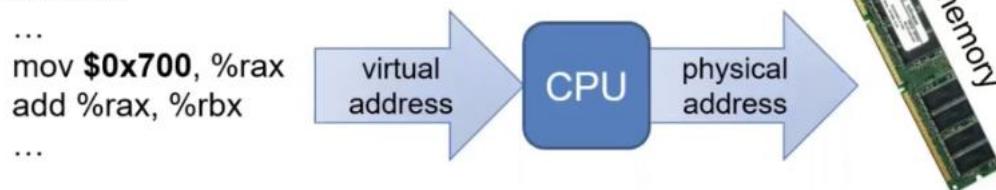
איך ניתן לזכור? למרות שזיכרון הוא אמצע התקן פיזי אנחנו לא ניתן לזכור לפי כתובות פיזיות כי זה היה יוצר הרבה בעיות. או כל הכתובות שהתוכנית ניגשת אליהם הן וירטואליות ככלומר לא אמיתיות מה שמעבד צריך לעשות זה בכלל פקודה שצריך לגשת לזכור זה לתרגם כתובות הווירטואלית לפיזית. למשל כמו שרואים בשקף אם יש לנו תחילה והוא מבקש לקרוא את התוכן בתא מספר 700 בזיכרון לתוך רגסטר התא הזה הוא לא פיזי, המעבד לוקח את 700 מתרגם אותה לכתובות

פיזית בעצמו למשל יכולה להיות מתורגמת ל-1300 למשל ודרך ניגש לכתוב פיזי ושם קורא המידע או כותב. nimrim לב שהתרגם עלול להיות יקר ובסוף התרגול נגיד איך יכולם להביע אותו.



- גישה ישירה לזיכרון הפיזי הייתה יוצרת הרבה בעיות: מחסום בזכרון רציף, היעדר בידוד בין תהליכיים, מגבלה על מרחב הזיכרון האפשרי.
- ה抽象ציה שפותרת את כל הבעיות הללו היא **זיכרון וירטואלי**.

process A:



- אבל אין ארכיטקטורה חינמית: **זיכרון וירטואלי פוגע ביציאות**.
- כל פקודה גישה לזיכרון דורשת תרגום יקר: וירטואלי ← פיזי.

---

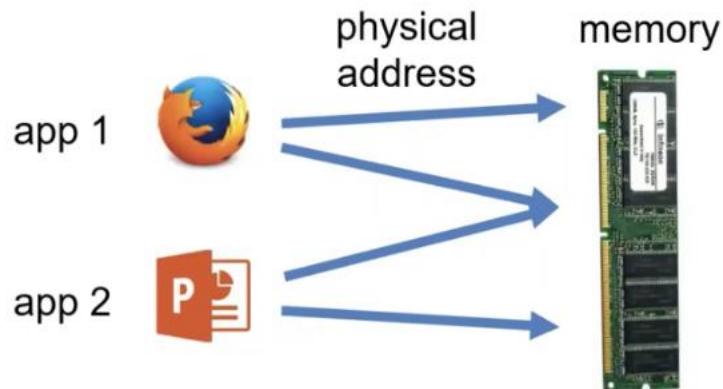
למה צריך זיכרון וירטואלי? אז בגדול זה רק התגלמות נוספת של עיקרונו הווירטואלי. יש לנו משאב פיזי מוגבל שהוא הזיכרון ורוצים למספר אותו להרבה תהליכיים בצורה מאובטחת בלי שיפריעו אחד לשנו או יסכו אחד את השני. איך עושים את זה?

## זיכרון פיזי

- התקני האחסון במחשב נחלקים לשניים:
  - זיכרון (DRAM) – אחסון נדייף, קטן יותר ו מהיר יותר (סדר גודל ms100).
  - דיסק קשיח – אחסון עמיד, גדול יותר ואיטי יותר (סדר גודל ms1).
- פקודות מכונה יכולות לפעול רק על רגיסטרים /או נתונים **בזיכרון**.
  - הקוד המבוצע ע"י המעבד והנתונים החדשים לביצוע הקוד חייבים להיות בזיכרון בזמן הביצוע.
  - אם רוצים לעבוד מידע מהdisk, יש להביא אותו לזכרון, לעבוד אותו, ולכתוב את התוצאה חזרה לדיסק.
- הגישה לזכרון היא, בסופו של דבר, באמצעות **כטבות פיזיות** של בתים (bytes). למשל, עבור זיכרון בגודל 8GB הכתובות הפיזיות הן מספרים שלמים בתחום [1 - 8GB, 0].
- עם זאת, יש חסכנות רבים לגישה ישירה באמצעות כתבות פיזיות.

מה זה זיכרון פיזי? במחשב שלנו יש שני התקני אחסון. הראשון זה זיכרון DRAM והוא אחסון נדייף ו מהיר וקטן, לעומת דיסק שהוא גדול ועמיד. שמכבים את החישמל על הזיכרון המידע מתנדף אבל על הדיסק זה נשמר, אבל על הזיכרון בגלל שהוא הרבה יותר מהיר אז שומרים עליו את המזוניות ערכימה, וכך טענים מידע מדיסק לזכרון ומעבדים אותו בזיכרון ומהזרים אותו חזרה לדיסק. אפשר לחשב על זיכרון כמערך של בתים. נזכר שכתובתם הם של בתים. נזכר שלכל בית בזיכרון יש כתובה [GB0,8] והיא פיזית, למה לא לתת למחשב לבקש מידע מכתובת פיזיות?

## חומרן #1: היעדר בידוד/הגנה בין תהליכיים

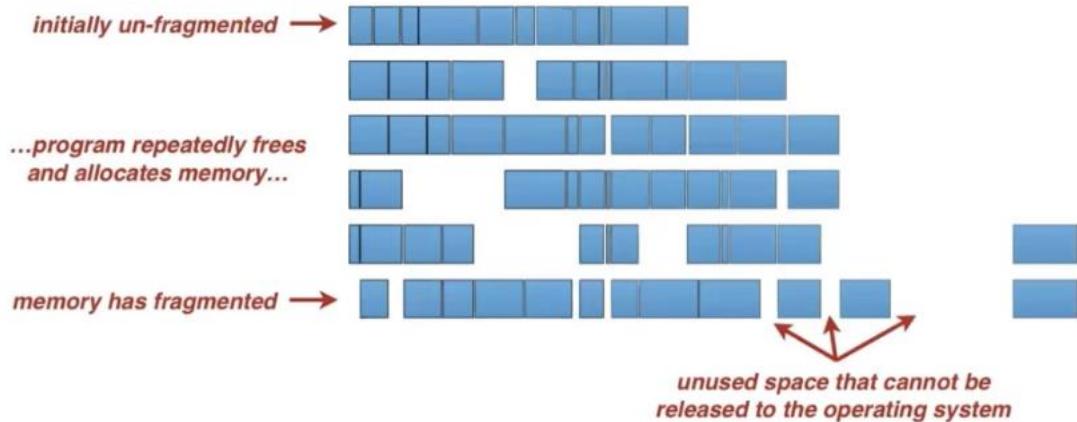


- אין הגנה על המידע – תהליך א' יכול לגשת לזיכרון של תהליך ב'.

אם כל תהליך יבקש כתובות פיזיות לגשת אליה או אין לנו הגנה ובידוד בין תהליכיים גם אין הפרדה של מידע מהתהליך אחד לאחרר. כמו שוראים למעלה יש שני תהליכיים פירופקס ו גם מצגת, אז נניח שפתחו אתר זמני אז הוא יכול לגשת לזיכרון לכתוב למרחב הזיכרון של המצגת ולהרווס. בטח שתקייפות סייבר הן די נפוצות.

## חומרון 2#: מחסור בזכרון רציף

- במערכת אמיתית, הזיכרון עובר קיטוע (fragmentation):



- גם כאשר יש מספיק זיכרון במערכת, הוא "שבור" לרסיסים.

המחשב שלנו דיב מהיר המעבד עושה מיליון פעולות malloc, free בשניה וכך מה שזה יגרום לקיטוע או פרגמננטציה ניתן לראות שורה ראשונה זה נראה כמו בשורה ראשונה כאשר כחול מסמל בלוקים שתפוסים ולבן בלוקים שלא תפוסים בזכרון. ואז אחרי פרק זמן הזכרון הוא שבור לרסיסים ונניח שרוצים מערך של 1 גיגה בזכרון, יוכל להיות שיש אבל נזכיר שמדובר בהיות רציף אבל בغالל השבריריים שנוצרו יש לנו מקום ל 1 גיגה אז זה לא רציף.

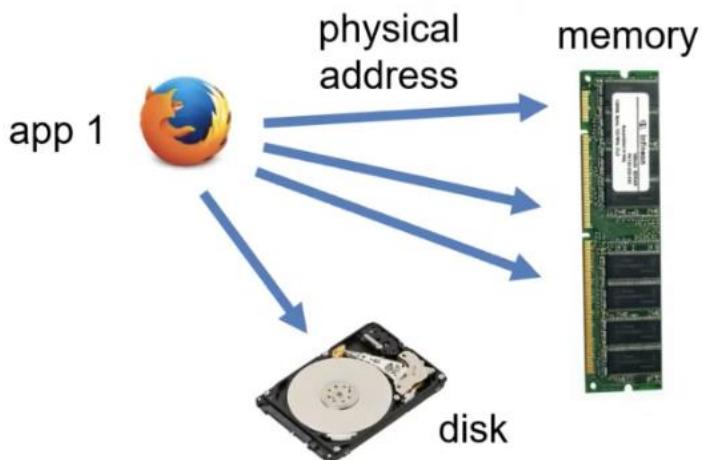
## פרגמנטציה (fragmentation)

- בזיכרון זיכרון כתוצאה מאופן שימוש לא יעיל.
- יש שני סוגים פרגמנטציה:
- **פרגמנטציה חיצונית** (external fragmentation) – בזיכרון מחוץ למקטעי הזיכרון בגלל שהם מפוזרים למרחב.
  - לדוגמה: מערך C חייב להיות מוקצה בצורה רציפה בזכרון.
  - יתכן כי לא נצליח להקצות מערך בגודל נתון למרוחות שיש מספיק זיכרון – סכום כל החוריות למרחב גדול מספיק, אבל החוריות לא מאורגנות באופן רציף.
- **פרגמנטציה פנימית** (internal fragmentation) – בזיכרון בתוך מקטעי הזיכרון כתוצאה מהקצת יתר.
  - לדוגמה: מהדר מסויים מיישר כל הקצת זיכרון לכפולה של בלוק בגודל N.
  - אם המשתמש מבקש זיכרון בגודל  $> N$ , שאר הזיכרון יתזבז.

---

יש לנו שני סוגי של פרגמנטציות חיצוני שזה מה שראינו קודם או פנימי שזה עשינו malloc ל גיגה בית והשתמשנו רק מיגה ולמערכת הפעלה לא יודעת כמה אנחנו רוצים להשתמש, יוכל להיות שבזבזו הרבה זיכרון.

## חומרן 3#: מגבלת זיכרון



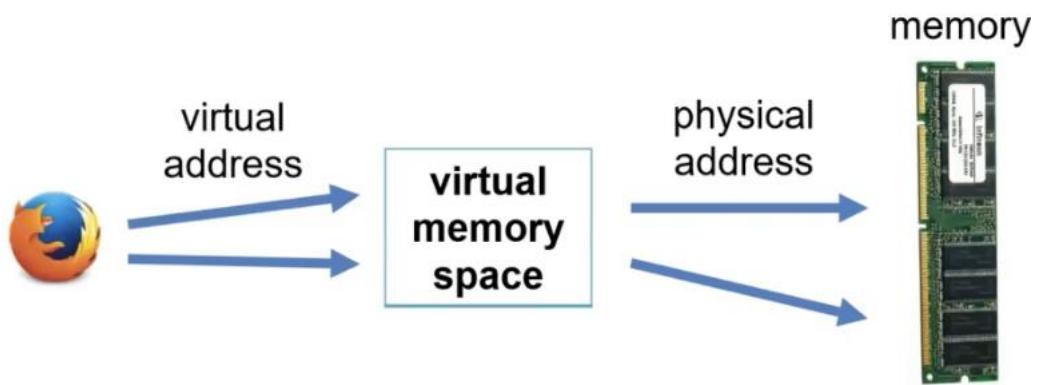
- הינו רוצים להשתמש גם בשטח האחסון שקיים בדיסק,  
**בצורה שקופה לקוד האפליקציה.**

---

מגבלה זיכרון. נניח שיש לנו זיכרון בגודל GB8 ואפליקציה שלנו צריכה GB12 מצד אחד יכולים להגיד לתוכנית שאין מקום לרוץ אבל מצד שני יש לנו את הדיסק שהוא גם אמצעי אחסון שיוכולים לשמר עליו הרבה מידע אומנם הוא איטי. הינו רוצים גם אמצעי אחסון קומביינה כדי להשתמש בדיסק.  
הפתרונות. זיכרון וירטואלי

## הפתרון: זיכרון וירטואלי

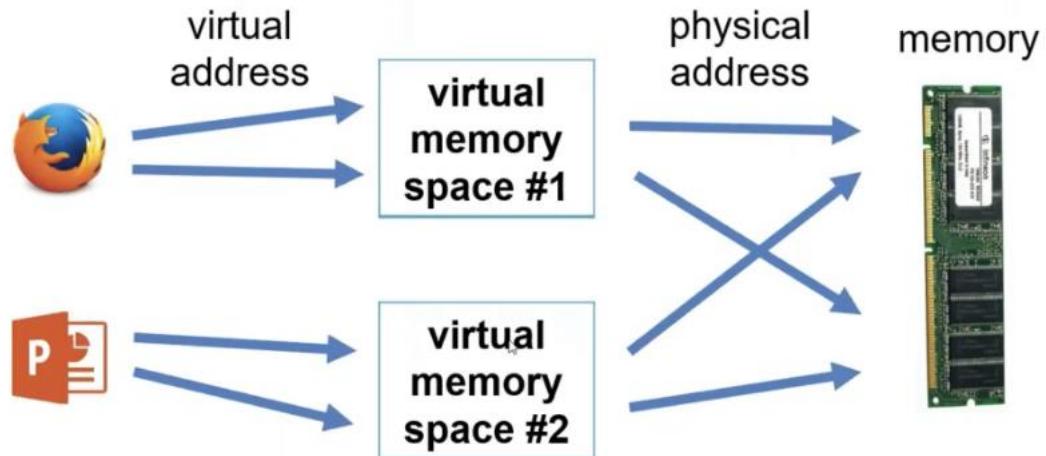
- פקודות המכונה ייגשו אך ורק לכתובות זיכרון וירטואליות.
- מערכת הפעלה תגדיר מיפוי (= פונקציה) בין כתובות וירטואליות לפיזיות: לכל כתובה וירטואלית מתאימה בדיקת כתובה פיזית אחת.
- המעבד יתרגם כתובה וירטואלית  $\leftarrow$  פיזית בזמן הגישה.



כעת התוכנית רואה רוחב וירטואלי ויהי לנו מיפוי ממוחב הזיכרון הווירטואלי למוחב הזיכרון הפיזי, התוכנית מאותו רגע רואה רק כתובות וירטואליות למשל firefox אם יבקש כתובה 2000 או 1000 זה יהיה במוחב הווירטואלי שלו, יש פונקציה שמערכת הפעלה מגדרה ממוחב הווירטואלי למוחב הפיזי והתהליך כל פעם שהוא רץ או כל פעם שמעבד מנסה לגשת לזכרון האונרואה בפקודה המcona איזשהו רכיב של כתובה הוא מתרגם אותה לכתובה פיזית.

## זיכרון וירטואלי נתן בידוד/הגנה

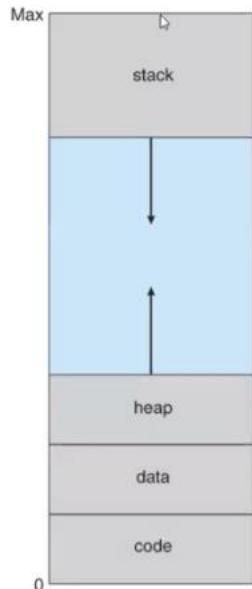
- תהליך יכול למסח זיכרון וירטואלי שלו עצמו.
- כל תהליך מקבל אשלה שהוא בלבד במערכת.



כרגע כל אפליקציה יש לה מרחב משלה, כך יכולים להבטיח בידוד ש לכל תהליך, וכל תהליך מקבל אשלה שהוא היחיד רץ במערכת.



## זיכרון וירטואלי מספק רציפות

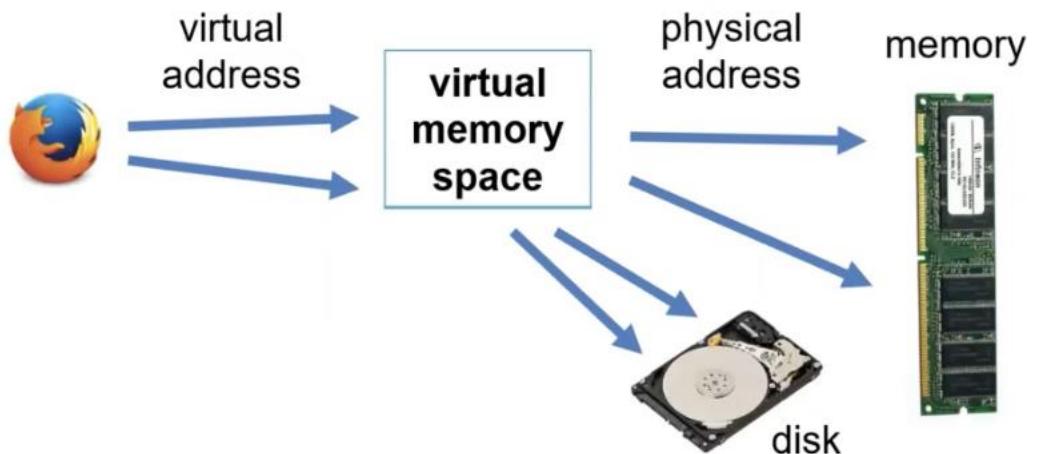


- תהליך חדש מקבל מרחב זיכרון וירטואלי "נקי" ורציף.
- בנוסף, מרחב הזיכרון הווירטואלי של תהליך יכול להיות גדול הרבה יותר מהזיכרון הפיזי הקיים.
- ← מערכת הפעלה תוכל למצוא בקלות יותר זיכרון רציף במרחב הווירטואלי.
- שימוש לב: הזיכרון הפיזי המתאים לא חייב להיות רציף!

כעת יש לנו רציפות, כי כל תהליך מקבל מרחב זיכרון וירטואלי נקי ורציף, וכי יכולים לחתם למרחב הווירטואלי להיות יותר גדול ממרחב הפיזי, נשים לה שהרציפות לא חייבת להיות רציפה במקום הפיזי כי יש לנו מיפוי, העיקר שהרציפות יהיה במרחב הווירטואלי.

## זיכרון וירטואלי אפשר swapping

- ניתן למפות חלקים מהזיכרון הווירטואלי אל הזיכרון או אל הדיסק.
- ← המשמש יראה יותר זיכרון ממה שיש במאגר המערכת.



כעת ניתן למפות זיכרון וירטואלי גם לדיסק אז אני לא מוגבלים עכשו בזכרון, יש לנו זיכרון עצום וזה בדיסק למרות שהוא יותר איטי. זה עדיף מאשר לא לחתה לתהlikך לרגע, אומנם נסבול עם איטיות אבל נסתדר עם זה.

זיכרון וירטואלי הוא די כדי קיים בכל מערכות המחשבים, שרתים של גугл, שעוני אפל, וכל מה שנרצה יש בהימוש של זיכרון וירטואלי. אנו נלמד כמה מימושים של זיכרון וירטואלי, והם קיימים בכל מיני מקומות.

**זכרון וירטואלי מציע יתרונות נוספים**

- **demand paging** – חסכו של זיכרון פיזי ע"י הקצתתו רק בגישה הראשונה לזכרון הווירטואלי.
  - למשל: אם הקצנו מערך גדול באמצעות (malloc) ולא ניגשנו לחלקים ממנו, החלקים האלה לא יהיו מגובים בזכרון הפיזי.
  - **deduplication** – חסכו של זיכרון פיזי במידה ואפליקציות שונות משתמשות באותו מידע **לקרייה בלבד**.
    - למשל, מרבית התהליכים משתמשים במידע של ספריית libc (לקרייה בלבד).
    - לכל תהליך מרחיב זיכרון וירטואלי שונה, אבל כולם יכולים למפותו לאוטו אזור פיזי שבו יושבת הספרייה libc.
  - **copy-on-write** – מנגנון לחיסכון של זיכרון פיזי ולמניעת העתקות מידע מיותרות – נראה בתרגול הבא.
    - ועוד יתרונות רבים אחרים...

## איך ממשים זיכרון וירטואלי?

**עוברת לע פיזית. מה הבעיה עם מימוש זה?**

הבעיה הוא שהמייפוי ישמור יותר מדי מקום ומבזבז יותר מדי מקום אז לא שיפרנו כלום.

**אופציה שנייה.** (מה שעושים באינטלו).

מחלקים את הזיכרון הווירטואלי לפיזים לבLOCKIM

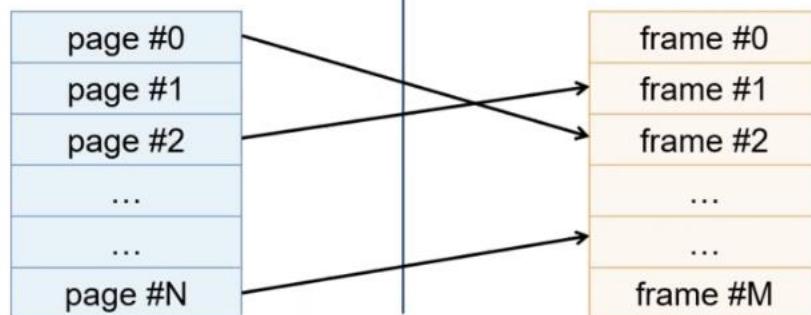
## 14 דפים ומסגרות

מרחב הזיכרון הווירטואלי

- מחולק לדפים (pages).
- גודל דף == גודל מסגרת (4KB).
- הדפים מיושרים בזיכרון הווירטואלי.

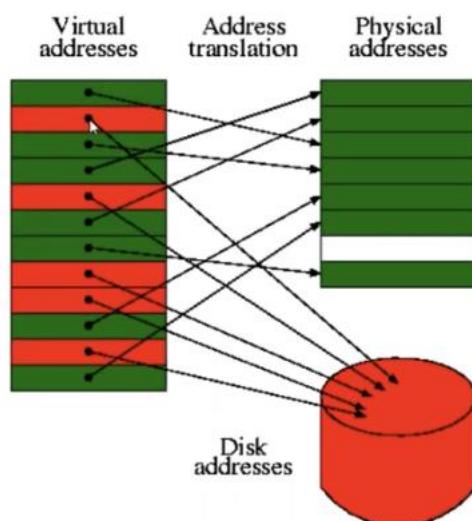
מרחב הזיכרון הפיזי

- מחולק למסגרות (frames).
- בלוקים עוקבים בגודל קבוע (4KB בארכיטקטורת 32-A).
- המסגרות מיושרות בזיכרון הפיזי.



דף אומרים לבлок בזיכרון הווירטואלי ומסגרת זה בלוק בזיכרון הפיזי.

## לא כל הדפים ממופים למסגרות פיזיות!



- חלק מהדפים לא ממופים כלל, כלומר הדף לא מגובה בזיכרון ולא בדיסק.
- מטעמי חיסכון בזיכרון ובזמן, אין טעם להקצות מראש את כל המרחב הווירטואלי של תהליכי.
- חלק מהדפים יכולים להיות מגובים בדיסק.
- נאמר כי הדפים **swapped out**.
- פרטים נוספים בתרגול על מטמון הדפים.



לא כל הדפים ממופים לזיכרון הפיזי. יש דפים שלא ממופים לאפ' מקום והתכונת לא יכולה לגשת אליהם למשל כתובת 0 כי שם יש NULL (תוכנית מיגשת רק לכתובות וירטואליות لكن אם ניגש לכתובת 0 וירטואלית מקבל segmentation fault).

האם תוכנית יכולה לגשת לכתובת 100? לא! כי 100 הינו נמצאת בדף של ה-0 ומעכשו מערכת הפעלה ממפה דפים למסגרות, לכן אם מעכשו אסור לגשת לכתובות 0 אז כל הדף הראשון לכל דף מס' אף אסור לגשת יותר מדויק כל הכתובות שנמצאות בין 0- 4095 אסור לגשת אליהם. נשים לב שלא מבוז כי זה זיכרון וירטואלי והוא לא מוקצת בזיכרון הפיזי. נשים לב שדף ראשון שכולל את ב-0 הוא מלא יכול להיות ממופה לזכרון הפיזי כי אסור לגשת אליו لكن אם נרצה נגד כל הסיכויים כן להקצות פה אז אנחנו נקבע מקום בזיכרון ואז יש מצב ש-0 קיבל כתובת פיזית וזה אסור להחולטיין כי היא שמורה לוונח ואז אין לנו את ה-0 הווה שכל כך רוצים אותו. האם המיפוי ממומש במערכת הפעלה או בחומרה? מערכת

הפעלה מגדרה המיפוי והמעבד זה שמתרגם. המיפוי הוא מבני נתונים בעצמו, מערכת הפעלה עושה `add`, `remove` וכי שעובד `search` זה המעבד באופן אוטומטי כי יש הרבה פעולות של חיפוש מאשר הכנסה הוצאה. כמה דפים יש במרחב בוירטואלי, קודם כל צריך לדעת הגודל של מרחב הווירטואלי.

## 32-A. דפים ומסגרות בארכיטקטורת IA-32

- בארQUITקטורת 32-A (ארQUITקטורת 32 בית של אינטל):
  - הזיכרון הווירטואלי הוא ברוחב 32 בית.
  - הזיכרון הפיזי הוא ברוחב 24 בית.

- מה מספר הדפים במרחב הווירטואלי?

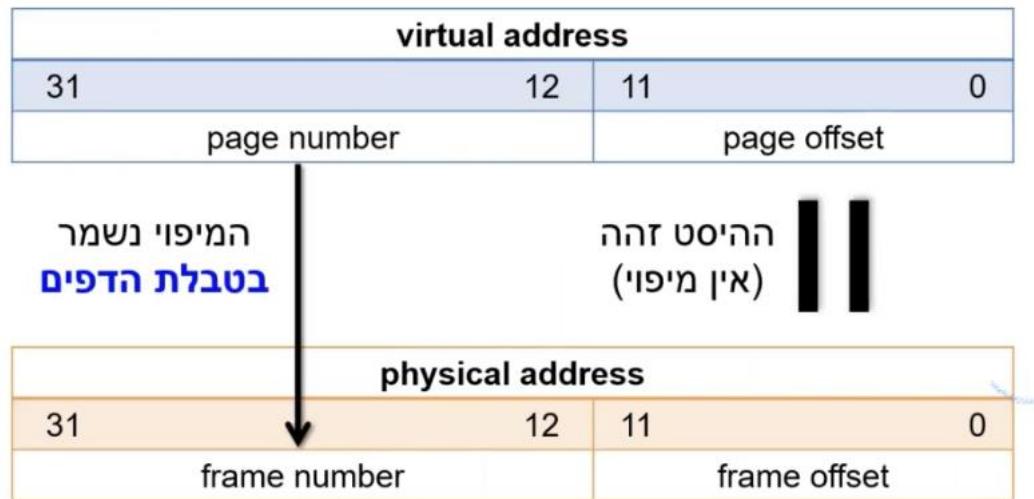
$$\frac{\text{space size}}{\text{page size}} = \frac{2^{32}}{2^{12}} = \frac{4\text{GB}}{4\text{KB}} = 1\text{M} \cong 1,000,000$$

- מה מספר המסגרות במרחב הפיזי?
  - כנ"ל (הчисוב זהה).

אז 1 Megabyte מיליאון קלומר בזיכרון הווירטואלי יש בערך מיליאון דפים ובפיזי בערך מיליאון מסגרת.

## מיפוי דפים למסגרות

- בכל גישה לזכרון, המעבד מתרגם את הכתובת הווירטואלית לכתובת פיזית באופן הבא:



המעבד מקבל כתובת וירטואלית של 32 ביט שanfordים אותה לשני שדות. 20 ביטים

- עליהם הם מספר הדף ותחתונים זה offset בתוך הדף וזה גם offset במסגרת.
- אך חלק שכן עובר זה מספר הדף שצורך לעבור למספר מסגרת. והמילוי הזה נשמר בטבלת הדפים. טבלת הדפים היא מבנה נתונים ששומר מיפוי בין מספרי דפים למספרי מסגרות. מה מבנה נתונים הכii נוח? טבלת ערבול? בוואו השתמש במערך.

## ניסוי 1# טבלת דפים ליניארית

- מערך שבו הכניסה ה-k מכילה את המיפוי של הדף ה-k.

- כל כניסה מכילה את:

- מספר המסגרת המתאימה לדף.

- סיביות בקרה:

- אם הדף בזיכרון? אם הוא בדיסק?

- אם הדף לקריאה בלבד?

- ועוד...

| #0  | מספר מסגרת | סיביות בקרה |
|-----|------------|-------------|
| #1  |            |             |
| #2  |            |             |
| ... |            |             |
| ... |            |             |
| #k  |            |             |
| ... |            |             |
| ... |            |             |
| ... |            |             |
| #N  |            |             |

כניסה k במערך אומרת מה המיפוי של הדף ה k וגם עוד כמה סיביות בקרה, במקרה שהדף לא בזיכרון כלומר לא ממפוה או בדיסק או יש סיבית dirty או valid כמו שראינו בהרצאה. האם נגיע לתהליכיים או רק לגרעין וכל מיני ביטים ששומרים. מה הגודל של המערך?

מספר כניסה במערך  $2^{20}$  שווה מיליון וכל שורה לקחת 20 ביטים לסיביות מסגרת ו 12 ביטים לסיביות בקרה,

## ניסוי 1#: טבלת דפים ליניארית

- עברו 100 תהליכי, התקורה על הזיכרון הפיזי היא MB 400.
- בפועל, תהליכי קטנים (ויש הרבה כאלה) ניגשים רק לחלק קטן של מרחב הזיכרון הוירטואלי, ולכן זה בזבזני להחזק את הטבלה כולה.
- יתרה מזאת, טבלת דפים ליניארית פשוט אינה ישימה בארכיטקטורות חדשות.
- למשל, נניח כי רוחב של כתובות וירטואלית ופיזית הוא 48 ביטים.
- מרחב הזיכרון הוירטואלי הוא בגודל  $TB = 2^{48} \cdot B = 2^{36}$ .
- גודל דף הוא 4KB והוא יש  $= 2^{12}$ .  $2^{48}/2^{12} = 2^{36}$  כניסה במערכת.
- כל כניסה היא בגודל 8 בתים.
- ↪ גודל טבלת הדפים יהיה **512GB** לכל תהליך!

זה טוב מאוד עם חזקה של 2. עכשיו יודעים שככל כניסה במערך זה 4 בתים. אז מה גודל הטבלה? אמרנו שיש מiga ביבט מספר דפים וכל אחד תופס כתובות וככיסות בקרה של 4 בתים כלומר 4MB.

נשים לב שלכל תהליך יש מרחב זיכרון וירטואלי لكن תהליכי שונים יש להם טבלה דפים שונה אבל חוטים משתפים אותה טבלת דפים כי מרחב הזיכרון משותף ביניהם.

## ניסוי 1#: טבלת דפים ליניארית

- עברו 100 תהליכי, התקורה על הזיכרון הפיזי היא MB 400.
  - בפועל, תהליכי קטנים (ויש הרבה כאלה) ניגשים רק לחלק קטן של מרחב הזיכרון הווירטואלי, ולכן זה בזבזני להחזיק את הטעלה כולה.
- יתרה מזאת, טבלת דפים ליניארית פשוט אינה ישימה בארכיטקטורות חדשות.
- למשל, נניח כי רוחב של כתובות וירטואלית ופיזית הוא 48 ביטים.
  - מרחב הזיכרון הווירטואלי הוא בגודל  $TB = 2^{48} \cdot 2^{36}$ .
  - גודל דף הוא 4KB והוא יש  $= 2^{12} / 2^{48}$  כניסה במערכת.
  - כל כניסה היא בגודל 8 בתים.
- ← גודל טבלת הדפים יהיה **512GB** לכל תהליך!

נשתמש באבחנה מאוד חשובה למערכות אמיתיות. ההבחנה היא שתהליכי קטנים ניגשים רק לחלק קטן של מרחב זיכרון וירטואלי ולכן זה בזבזני להחזיק הטעלה כולה.

למה הכוונה?

## ניסוי #2:

### טבלת דפים היררכית

|      |   |
|------|---|
| 0    |   |
| ...  |   |
| ...  |   |
| 1023 |   |
| 1024 | ↓ |
| ...  |   |
| 2047 |   |
| ...  |   |
| ...  |   |
| ...  |   |
| ...  |   |
| ...  |   |
| ...  |   |
| ...  |   |
| ...  |   |
| 1M   |   |

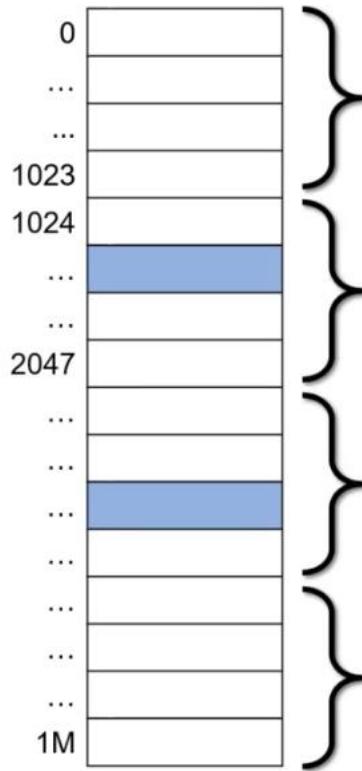
- נניח כי תהיליך מסויים  $\mathbb{P}$  ניגש לשני דפים בלבד.

- טבלת הדפים היליניארית של התהיליך  $\mathbb{P}$  משתמש בשתי כניסה בלבד, כפי שקרה התרשים:

• קל לראות את בזבוז הזיכרון...

למשל ניתן לראות שיש לנו תהיליך והוא ניגש רק לשני דפים מכל המרחב הווירטואלי. יש רק שני כניסה שהן באמת מモפות וכל שאר הכניסות בעצם מצביעות שהדף לא ממופף כלומר ה-`valid` שלהם כבוי. כלומר הוא 0. וראויים שרק שתי כניסה תפוסות בטבלת המיפוי. ניתן לראות פה את בזבוז הזיכרון.

## ניסיון #2: טבלת דפים היררכית



- נחלק את הכניסות בטבלה לבlokים בגודל מסגרת פיזית.

- כמה כניסה יהיו בכל בלוק?

$$\frac{\text{frame size}}{\text{entry size}} = \frac{4\text{KB}}{4\text{B}} = 1024$$

- כמה בלוקים יהיו?

$$\frac{1\text{M}}{1024} = 1024$$

נחלק את המערכת שלנו כל מיליון ה כניסה לבlokים, כל מסגרת היא KB4 וכל כניסה זה 4 בתים לכן כל בלוק הוא 1024 כניסה. בעצם נשים לב שרוב הבלוקים הינם ריקים לוגרי. גם מרחיב הזיכרון שלנו זה מיליון כניסה וכל בלוק זה 1024 כניסה לכן סך הכל יש 1024 בלוקים.

## ניסויון 2#:

### טבלת דפים היררכית

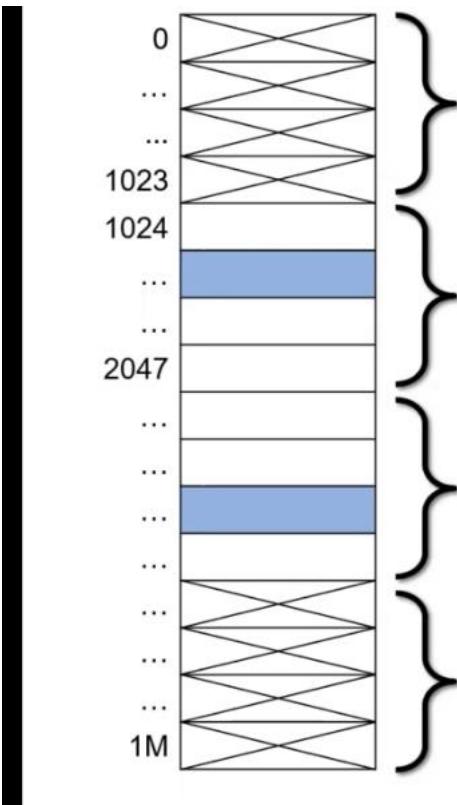
- נשים לב כי מרבית הבלוקים ריקים לגמרי...

• ולכן לא כדאי להקצתות אותם.

- נשמר טבלה נוספת עם 1024 כניסה כנימות אשר תצביע לבלוקים:

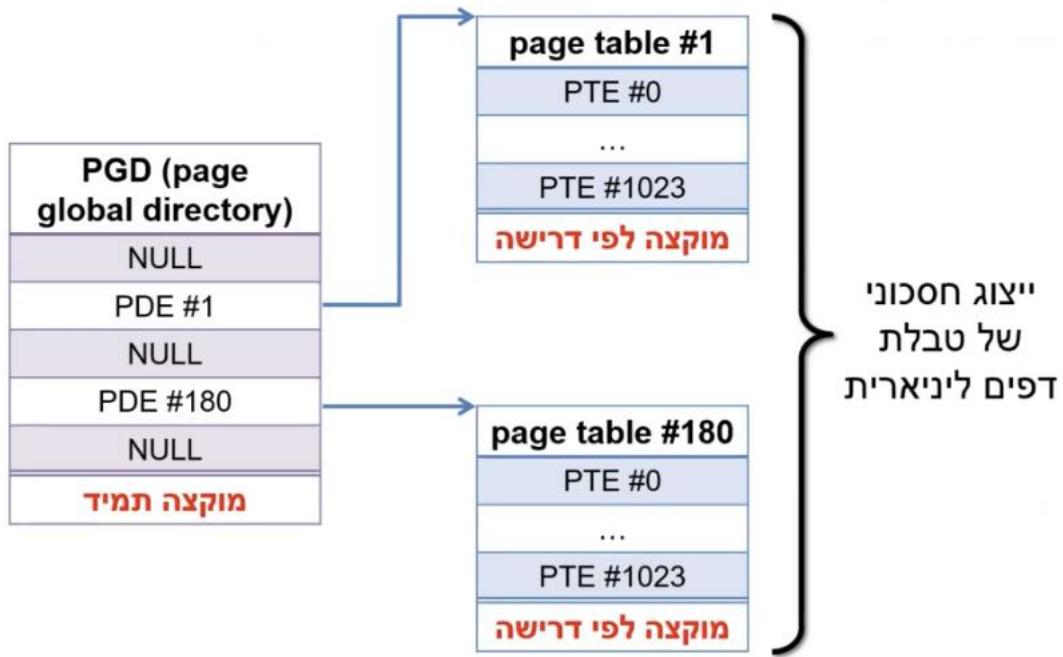
• NULL עבור בלוק ריק.

• כתובת פיזית עבור בלוק מוקצה.



נשימים לב שמרבית הבלוקים ריקים לגמרי כלומר רוב הבלוקים מכילים 1024 כניסה ריקות לחלווטין. אז עדיף לא להקצתות את הבלוקים האלה. אבל איך נדע איזה בלוקים בקצנו ואיזה לא? אנחנו נשמר טבלה לצד קטנה נוספת של 1024 כניסה. והטבלה תצביע לבלוקים ברמה הבאה. אף ברמה הראשונה לא מוקצה כלום כלומר בלוק ריק אז תצביע לאוнач. ואם כן מוקצה אז נשמר כתובת פיזית שם.

## ניסוי 2# טבלת דפים היררכית



למשל רואים שיש פה טבלה אחד גדולה שנקראה PGD שייהי בה כניסה. רוב הכניסות בה יהיה Null. כניסה Null אומר שככל ה-1024 כניסה שמתאימות לכינסה זו ברמה לא יהיה מוקצים. לאחרת ברמה הבאה יש חלק ממיפויים שקיים או נקבע מרמה עליונה לרמה הבאה אז היא תקבע להתחלה שלהם בזיכרון הפיזי ובתוך הבלוקים האלה נשמר את המיפוי. עכשו נשים לב שבמבנה זה מוקצים רק שלוש בלוקים אז במקום 1024 בлокים הקצנו רק 3. נניח עכשו שמקבלים כתובת וירטואלית ורוצים לתרגם אותה לכתובת פיזי איך מוצאים אותה דרך המבנה? קודם כל עושים מודולו 1024 כדי לדעת באיזה בלוק נמצאת או כניסה שווה שקל לעשوت shift 12. אחרי שמצאנו אותו נעשה מודולו 1024.

## תהליך תרגום כתובות וירטואלית



- איך המעבד מתרגם כתובות וירטואלית V לכתובת פיזית?
- בשלב הראשון, המעבד מחשב את מספר הדף הווירטואלי:

$$P = (V >> 12)$$

- בטבלה דפים ליניארית:

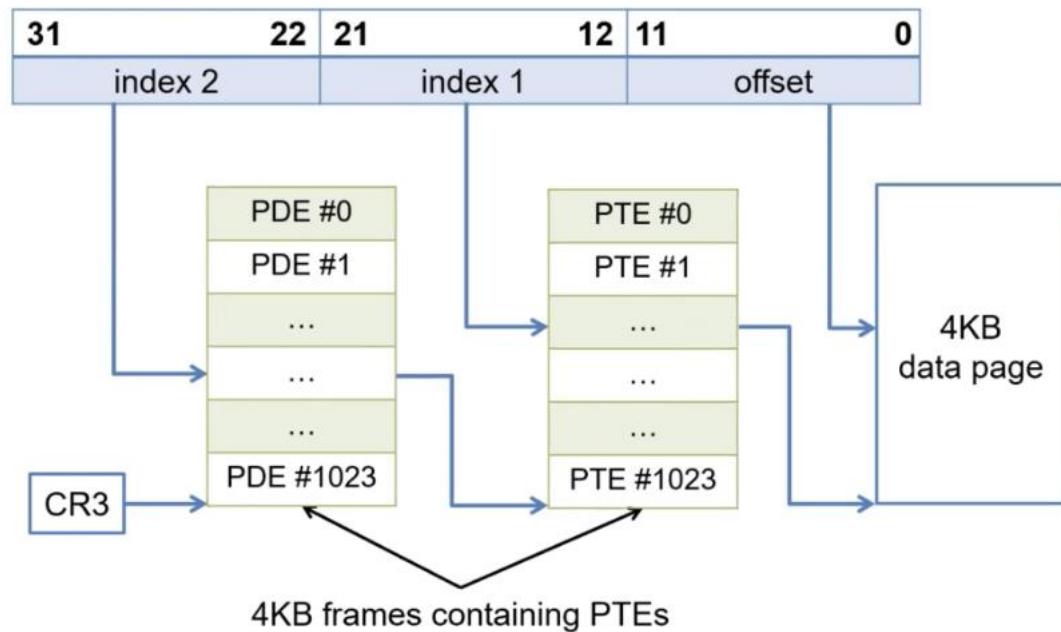
1. התרגום נמצא בכניסה P במערך.

- בטבלה דפים היררכית:

1. המעבד קורא את הכניסה 1024 / P בrama העלינה של העץ.
2. אם הכניסה זו == NULL, אז אין תרגום (דף לא בזיכרון).
3. אחרת, התרגום נמצא בכניסה 1024 % P בrama התחתונה של העץ.



## פירוק כתובות וירטואלית לשדות



ניתן לראות פה הפרוק לשדות ה-12 ביטים תחתונים לא מתייחסים אליהם בכלל כי הם ה offset בתחום המסגרת. עכשו ניתן לראות שיש לנו קודם כל 1024 בלוקים בטבלת ה-*ci* משמאלי שנקרה *pte* והוא תמיד מוקצת וממי שמצביע אליו זה רגסטר מיוחד שנקרא CR3 בטבלה הזאת יכול להיות שלכל כניסה מוקצת בלוק, אנחנו לא מזוהים יודעים מה מומפה ומה לא. נניח שיש איזשהו בלוק מומפה אז יהיה מוצבע על ידי איזשהו כמיסה מהטבלה, והבלוק הזה יכול 1024 כניסה כבר אמרנו למה, עכשו זה מצביע לאיזשהו בלוק בזיכרון הפיזי בגודל KB4, שניתן לדעת דרך offset לאן מומפה ה-*pte*. לכן יש לה בעצם שלוש שלבים של חישוב. חלוקה, מודולו, offset. נשים לב שימושית אם מקצים על הבלוקים אז לא הזכו זיכרון אל הוסףנו עבור הטבלאות אבל זה די לא מעשי, רוב התהליכיים משתמשים בכניסות בודדות.

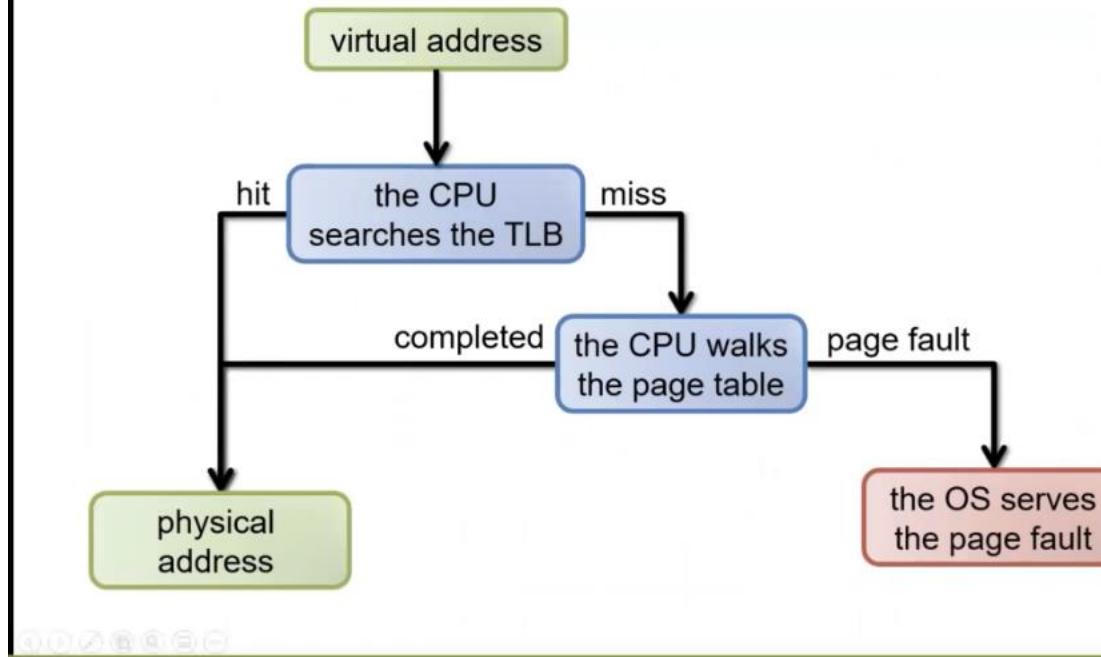
## TLB - Translation Lookaside Buffer

- תרגום כתובת ורטואלית לפיזית קורה כל גישה לזיכרון.
- 30% מהפקודות בתכנית ממוצעת ניגשות לזיכרון ↪ תקורה גבוהה.
- כדי לשפר את הביצועים, מעבדי אינטל מכילים מטען (cache) מיוחד, ה-**TLB**, אשר מכיל את התרגומים האחרונים בהם השתמשו.
- המעבד מחפש ב-TLB לפני החיפוש בטבלת הדפים. אם התרגום המבוקש נמצא ב-TLB, נחסכו גישות יקרות לזיכרון (מספר הרמות בהיררכיה).
- אם התרגום המבוקש לא נמצא ב-TLB, המעבד פונה לחפש בטבלת הדפים ואז מוסיף ל-TLB את התרגום החדש (לטובת הגישות הבאות לזיכרון).

| page number | frame number | flags           |
|-------------|--------------|-----------------|
| 12          | 25           | r/w, accessed   |
| 55          | 93           | accessed, dirty |
| ...         | ...          |                 |

נשים לב שעכשו במקומות גישה אחות לזכרון עכשו יש 3 גישות זהה מאוד מאוד בזבוני. אז אולי תהליך לא ניגש לזכרון הרבה? ממש לא, בתוכנית ממוצעת, היא ניגשת לזכרון כ 50-30% מכל פקודות המעבד הן גישה לזכרון. לכן זה היה מאיית את התוכנית שלנו לכן מה שעושים באינטל זה cache מטמון שמאיצן את תהליך התרגום. אז הרעיון הוא האם תרגמנו כתובות אז סיכוי גדול שהוא גם תהיה בעתיד ובשביל זה יש את ה bit dirty. למשל נדמיין שיש לנו מבני נתונים std::vector ושיושב באופן רציף בזיכרון שנרצה לאתחל אותו באפסים וכל איטרציה ניגשים לזכרון וכתובות אלו הן סמכות לנו כולם באותו דף אז יש סיכוי טוב שויקטור הזה יושב בדף אחד. פעם ראשונה שניגשים למערך תא ראשון אנו נכנס לכתובת 1 ופעם שנייה תא שנייה 2 אבל זה אותו דף אז חמה לעשות תרגום הדף פעמיים? איך נחשוך? מה שנעשה זה נשמר תרגום אחרון במטמון קטן ומהיר ליד המעבד והוא משתמש בו נקרא TLB והוא רושם שם תרגומים אחרונים, והוא ישמר מספר הדף את מספר מסגרת שהוא ממופה אליה ואת הדגלים. נשים לב אני משתמשיםஇதை 20-30 תרגומים אחרונים וכן חוסכים כמעט 90% מגישות.

## ויכום: תהליך התרגום במעבד אינטל



מעכשיו כל פעם התרגול יהיה כך, המעבד מקבלת כותבת וירטואלית הוא ממחפש אותה בTLB אם מצא מספר הדף זהו שיחק אותה אז יש כתיבה פיזית. אבל אם פספסנו אז הולך ועושה את כל **page walk** בטבלת הדפים וניגש לזכרון עוד פעמיים, אם סיים בהצלחה אז יש לו עוד פעם כתובות פיזית אבל יכול לגלוות שהיא לא ממפות בכלל ואז ייחזר שגיאה של **page fault** שזה **seg fault**. איך החיפוש בTLB מתבצע? זה מהهو בתוך מבנה פנימי של מעבד והוא יודע לעשות את זה.

## פסילת תוכן TLB

- ה-TLB מכיל עותק חלקי של המידע הקיים בטבלת הדפים, ולכן מערכת הפעלה אחראית לשמר על קוי הרנטיטות המידע ב-TLB.
- הגሩין חייב לפסול (invalidate) את תוכן TLB במקרים מסוימים, לדוגמה:
  1. כאשר הגሩין מוחק כניסה בטבלת הדפים (כדי לפנות מסגרת מהזיכרון לדיסק), הוא מוחק גם את הכניסה המתאימה ב-TLB. אחרת, התהילה עשוי למשוך לא מעודכן, בעוד TLB עדין מצביע למסגרת שכבר פונתה מהזיכרון.
  2. בעת החלפת הקשר, הגሩין מוחק את תוכן TLB כלו.
    - התהילה הבא לביצוע ייגש למסגרות של התהילה שרצ לפנוי.

נשים לב ש ה-TLB מכיל עותק חלקי של המידע מטבלת הדפים וכך לפעם יכול להציג מידע לא רלוונטי. למשל בהחלפת הקשר לכל תהליך יש טבלת דפים משלה ומשעושים החלפת הקשר מחלפים כתובת שאלת מצביע רגיסטר CR3 ועכשו טבלת דפים קודמת לא רלוונטית כולל גם את ה-TLB אז בשביל כך עושים flash קלומר מוחקים מה שיש במטמון וזה קורה כל פעם שנעשה החלפה הקשר. ויש עוד פעמים שעושים את זה למשל אף הייתה בטבלת הדפים מיפוי שנמחק אז חיברים למחוק גל מ-TB כי המעבד בודק אותו לפני טבלת הדפים אז יכול להיות שהחשב שיש מיפוי עדין וכך מונעים את זה.

או כל פעם בונים אותה מחדש שעושים החלפת הקשר קלומר מחממים את המטען ויבנה לאט את התוכן שלו, וכך תרגומים ראשוניים יהיה יקרים כי הם יחפשו בטבלת הדפים עד שישים את התרגום ב-TLB כלומר החלפות הקשר יש יפגעו ביצועים כי יקח זמן לחם המטען וכך נרצה שיקירה כמה שפחות. נשים לב שבחלפת הקשר של שני הוטים לא מוחקים את ה-TLB (יש להם זיכרון), מצב שני שלא מוחקים TLB זה אם עוברים תהליך גሩין שאין לו מרחב זיכרון משלה

והוא פועל למרחב הזיכרון של הגרעין لكن הוא מנצח טבלה דפימ של התחלה שרצ לפניו.

מעבר עכשו למעבדי אינטל 64 ביט.

## גודל מרחב הזיכרון

- במעבדי 64 ביט של אינטל (ארכיטקטורת 64x), משתמשים בכתובות וירטואליות של 48 ביט בלבד (מתוך 64 אפשריים). מה גודל מרחב הזיכרון הוירטואלי?

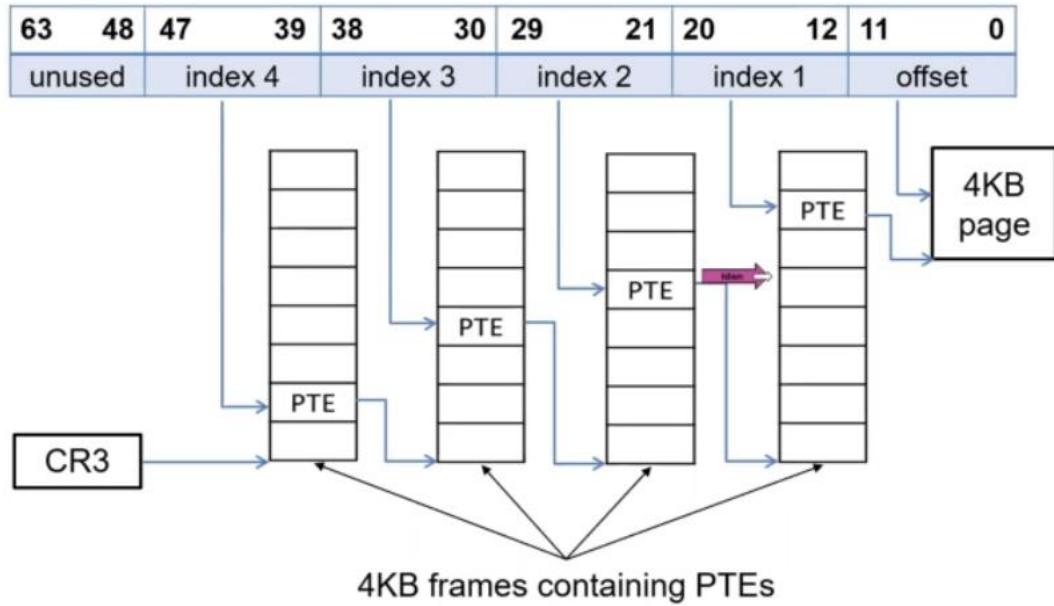
$$2^{48} B = 256 TB$$

- מה היה גודל מרחב הזיכרון אם היו משתמשים בכל 64 הביטים לייצוג כתובות?  $2^{64} = 16 \text{ Exabyte}$

- מדוע, אם כן, משתמשים רק ב-48 ביט?
- מעבר האפליקציות הקיימות היום, אין צורך למרחב וירטואלי גדול כל כך.

כעת הכתובת הפיזית היא 8 בתים ולא 4 שכן בכל דף יש 512 כבישות.

## page table walk



עכשו משאנו ממש מבלב נשים לה ש ה offset קובע לאיזה בית אנו ניגש בזיכרון. כלומר במקרה פה יש לנו  $2^{12}$  אופציות ליצג מספר וכל אחד מייצג בית בזיכרון שכן יש לנו לנו 4 קילו בית. עכשו נשים לב שהטבלה מחולקת לאינדיקטורים של אחד 9 ביטים וזה כדי להתביע על כניסה בטבלה מתוך ה 512 כלומר מספר רק 9 ביטים ליצג מספר בין 0-512.

מה אם נרצה לנוהל הזיכרון בדףים יותר גדולים או הם משניף את כל החלוקת שלנו לשדות כלומר תחילה היה יכול להיות יותר מסובך. פשוט מה שעשו זה הגדילו ה offset עכשו ניתן ליצג יותר מספר גדול שכן יש יותר ביטים בזיכרון ומצד שני מספר האנדקסים עכשו ירד ל 2.

ו גודל המסגרת עכשו זה MB2. ואפשר לעשות את זה עד דפים שונים لكن בזיכרון יכול להיות מסגרות של 2 מגה, 4 קילו בית אפילו גיגה וכולם אחד ליד השני בזיכרון.

נשים לב ש TLB בדפים יותר גדולים מכשה יותר מדפים קטנים. למשל בדפים גדולים אם יש לנו 10 כניסה וכל מיפוי הוא 1 גיגה אז TLB יכשה זיכרון של 1 גיגה. וניתן לשים לב שם מקזרים את ה pagewalk כי ירדנו במספר האינדקסים. אבל החיסרונו שלהם זה שהם תופסים מקום גדול רציף בזכרון וזה יכול לעשות בעית פרגמנטציה כלומר מהسور בזכרון רציף לשאר התהליכיים.

## תרגול 10

ראינו ש זיכרון וירטואלי הסיפור דלו בגודל שאחנו חינו באשליה כלומר מה שבשבילנו היה זיכרון פיזי הינו זיכרון וירטואלי, ומעכשיו כל פעם שתוכנית המשתמש ניגשת לזכרון היא עוברת דרך כתובת וירטואלית שעוברת תרגום כמו שרואים לעלה. בירוק יש כתובת וירטואלית ועובד מנסה לבצע איזשהו פקודה שניגשת לכתובת הווירטואלית זו, מה שיעשה זה יחשב ב TLP אם הייתה פגעה ב TLP אז מיד מתקבלת כתובת פיזית ותרגם מסתיים אחרה יש החטלה ואז המעבד ממשיך ומהפש בטבלה בדפים היררכית שמסודרת כמו עץ ושוב אם התהליך מסתיים בהצלחה אז בידים של המעבד יש כתובת פיזית ויכול להמשיך אבל יכול להיות שלא מסתיים בהצלחה מכמה סיבות, למשל אם present bit, valid וכו.. והרבה דברים שיכולים לגרום לשגיאה. במקרה הזה מקבלים page fault ואז מערכת הפעלה מתעוררת, והיא מנסה לפטור את ה page fault שהיא חריגה לכל דבר אבל יש הבדל משמעותי שגורם לה להיות מיוחד שהוא לא תמיד שגיאה, ומערכת הפעלה משתמשת בה כדי לנצל את הזיכרון.

בגודל בציור זה נראה כך, זה נראה כאילו שיש חריגת דף מערכת הפעלה צריכה לסוג אותה לחוקית או לא חוקית אם לא חוקית אז הרגת התהיליך אבל הרבה מקרים שמערכת הפעלה מסווגת אותה כחוקית וואז תוסיף או תתקן מיפויים בטבלת הדפים ותסייע החריגת והחריגת זו אם תסתיים בהצלחה המעבד מנסה לבצע אותה שוב. בתרגול זהה נדבר על שלבים אלו.

נדבר על מגווןניים עצלניים שמערכת הפעלה משתמשת בהם לניהל את הזיכרון. הערה: לא להתבלבל בין מונח cache לבין TLB זה זיכרון די קרוב למעבד שלפעמים שומרים עליו את טבלת הדפים כדי להסוך גישה לזכרון בפרט למשל שעושיפ pagewalk או בודקים את ב caches קודם ואז משם מתקדמים. TLB הוא כן cache אבל הוא לא כמו  $2|1|2|1|1|1$  אלא מטמון מסוג אחר ולא cache רגיל.

מבנה נתונים בגרעין לניהל זיכרון.  
ראינו חלקם למשל טבלאות דפים, מרחבי זיכרון, אזורים זיכרון

אנו משתמשים במעבד של 64 ביטים מעבדים של intel, amd אבל בפועל משתמשים רק ב 48 ביטים שהוא בעצם 256 טירה בית של זיכרון וירטואלי. אותם מחלקים לשני חלקים 128 ו 128 המתחננים כפי שהראים בציור הם למרחב המשמש בלבד, והעלויונים הם לגרעין בלבד, יש הפרדה ברורה למעשה. גם בתול מרחב המשמש יש חלוקה לאזור זיכרון לערימה, ועוד אזור זיכרון data, Bss,

code. כך נראה מרחב הזיכרון של כל אפליקציה, ונזכור שכל אחת לא יכולה לגשת למרחב הגרעין בגלל שיש CPL רגיסטר, אם הרגיסטר רואה שMRIIZ קוד משתמש, למשל הוא יודע שהוא MRIIZ קוד משתמש או גרעין ואם מזהה SMRIIZ קוד משתמש שמייל על טבלת הדפים אל רואה שאחד הביטים באחד הכניסות ה kernel user bit דלוק בה שהי אמר שכניסה זו היא במצב גרעין בלבד וכן מיד קורה חריגת דף אם הוא במצב משתמש. לא להתבלבל עם I/O זהה לא קשור לזכרון אלא להתקני קלט פלט. לפי הצירור ניתן לראות חור של כתובות לא מנוצלות

איפה שמסומן לעלה.

השוף הזה אומר מה שכבר אמרנו.

מה זה אומר מרחב הגרעין?

נשים לב שיש מרחב גרעין אחד והרבה תהליכיים אז יש לנו הרבה מרחבי זיכרון וכולם בחלק של הגרעין נראה אותו דבר למשל ניתן לראות שעוברים מתהליך אחד לאחר מצב הגרעין נשאר אותו דבר ככל מר 128 טרייה בית עליונים נשאר אותו דבר אך 128 תחוננים לא נראה אותו דבר. זה נוח כדי להבטיח שכחובות וירטואלית של גרעין תהיה זהה לכל התהליכים.

מרחב זיכרון בלינוקס נראה כך, הוא נמצא ככניסה בתחום ה PCB שנקרא `mm` וזה אומר memory management והוא מצביעה ל סטרקט. נתון אחר שלא דיברנו עליינו זה נקרא `mmlist` ששומר כל הנתונים הקשורים לניהול זיכרון של תהליך ויישם את ה `pgd` הזה מצביע לטבלת הדפים בזיכרון לתהליך (יש טבלה לכל תהליך).

נתון שלא דיברנו עליו זה mmap כלומר רשות אזור הזיכרון, מה זה בדיק? נשים לה שלא מספיק טבלת הדפים צרייך גם את רשות אזור זיכרון.

מגדרים תהליך גרעין שמערכת הפעלה יוצרת ואין שום התgelמות למרחב המשתמש, אין להם מרחב משותם או מהסנית משותם או עירמה. הם חיים רק ב-128 טירה בתים עליונים כי למשל יותר נוח לפעמים לגרעין להוציא חלק מתחלכים שלו לתחלכי גרעין למשל תחלכים שטפלים בפסיקות, והם רק ניגשים למרחב גרעין, אין להם מרחב משותם لكن משותם למרחב הזיכרון לתהליך שרצ לפניו. כלומר לא מחליף טבלת דפים.

למרחב הזיכרון של תהליך יש לנו רשימה גlobלית של כל מתاري של מרחבי הזיכרון.

יש לנו pgd שהוא טבלת הדפים והוא זאת שנטענה לרוגסטר (ההתחלת שלה) טוענים אליו את הכתובית הפיזית של הטבלה, כדי שידעו איפה לעשות את ה walk.page

Mmap

זאת רשימה ממוינת של אזורי הזיכרון.

## Rss

זה מספר מסגרות בשימוש.

## Total\_vm

מספר דפים כולל באזור הזיכרון.

הערה: ניתן שיש כמה דפים מצבעים לאוთה מסגרת لكن rss>vm>tottal\_vm</rss

תהליך גרעין יש לו pcb ושהה pgd מצבע לahn כי אין לו טבלת דפים משלה.

אזור זיכרון של תהליך, דבר ראשון צריך להבין שם אק במרחבי משתמש כלומר קובצתה רציפה של דפים שמשמעותם פונקציונליות כלשהו, למשל מהסנית יכולה להיות 10 דפים וכל הרצף הזה זה אזור זיכרון שמיוצג על ידי vm\_area\_struct שבה יש התחלת וסוף כלומר כהתובת ההתחלת גודל וארוך של אזור זיכרון, יש אזור אחד למחסנית, ערימה, text וניתן ליצור חדש באמצעות mmap שככל פעם ספרייה דינמית נקשרת יהיה לה אזור זיכרון למרחב זיכרון של התוכנית, וכאל אזורים אלה נשמרים ברשימה ממוינת.

חוט גרעין אמרנו שהוא תהליך שנוצר על ידי מערכת הפעלה כדי ליצור משימה כלדיה של מערכת הפעלה, לכל אחד יש PCB אבל רץ בהרשות גרעין על מהסנית גרעין. למה זה חשוב?

כי יש שני שדות במתאר של מרחב זיכרון שלא דיברנו עליהם, כל מרחב זיכרון שומר עוד שני פרטי מידע mmuser, mmcount.

## Mmuser

הוא אומר כמה תהליכי משתמש חולקים את אותו מרחב זיכרון. זה יכול להיות באמות חוטים, כי חוטים ימשתפים מרחבי זיכרון.

## Mmcount

סופר כמה תהליכי גרעין + משתמש חולקים זיכרון אבל כל תהליכי משתמש בספרים כאחד למשל המעליה ניתן לראות שיש ר תהליכי גרעין שמצבעים על mmstruct ויש 3 תהליכי משתמש שמצבעים על mmuser. למה עושים את זה? כדי לדעת متى לשחרר אורי זיכרון האם רק של המשתמש לנוכח ש mmuser יורץ ל 0 יכולים לשחרר את כל אורי זיכרון. למשל עירמה, מהסנית, קוד, אבל עדין לא יכולים לשחרר את טבלת הדפים, כי אולי יש עדין תהליכי גרעין שמצבעים על מרחב זיכרון לנוכח mmcount יורץ ל 0 יכולים לשחרר את הכל כולל טבלת הדפים.

אורי זיכרון זה פשוט רצף של דפים, לא חופפים, כל אחד יש הרשות לקרוא וכתיבה וביצוע משלו, אל כתובות חוקית אז היא חייבת להיות באור זיכרון. והם מושרים לKB4. כל פרטי מידע שמופיעים בטבלת דפים מופיעים גם באור זיכרון. אז למה צריכים אותה? בהמשך נראה ש צריך גם אותה. אפילו עכשו יכולים להבין ש

עבור page fault מעאכנת הפעלה תתקן את טבלת הדפים שהיא אומר שהיו שלבים בሪיצה שאבלת הדפים לא מעודכנת אבל אזורי הזיכרון מעודכנת ואז יקרה page fault ומערכת הפעלה תתקן את טבלת הדפים. בנוסף יותר יעיל למצוא המידע באזורי זיכרון מאשר בטבלת דפים, כי הם שומרים רצף של דפים ששומרים כל המידע על הרשאות שלהם בבית אחד לנוכח זה מידע הרבה יותר גס וקל להפנשו. אזור זיכרון הוא רציף בזיכרון וירטואלי ולא פיזי, כלומר רצף של דפים).

הוא מוצג כל ידי סטקרט וניתן לראות את השדות שלו.

ניתן לראות שאחד משדות זה דלים שמצוין חכנות של אзор כלשהו למשל vmread כלומר האם ניתן לקרוא לכתוב או לביצוע. עוד דגלים מעניין שהוא vmshared שאומר אם ניתן לשותפ לדפים באזור זהה.

ברגע שתהליך נולד מקבל אזורי הזיכרון שרואים למעלה. והוא יכול להוסיך אזוריים נוספים עם קיראת מערכת mmap. אז התנונה בגודל בלינוקס זה כך,

או מה זה brk או שם אחר שקול sbbrk

אם מעבירים דלהא חיובי או מגדילים את הערימה או מקטינים בהתאם. אם ערכת הפעלה תרצה להקצות מהוז או היא משתמשת ב mmap.

يُؤثر أذون ذِكرٌ حَدْسٌ وَمَكْبَلَةٌ يُوَثِّرُ فَرْمَطَرِيمْ، الْفَدَاهِيَا كَتْوَبَةً، بِدَرْكٍ كُلَّ مَعْبِرِيمْ لِلَّهِ كَدِي شَاهِيَا تَحْزِيرَ كَتْوَبَةِ ذِكْرٌ وَيَرْتَوَالِيَّةِ مَعْصِمَهَا. كَمَوْبَنْ مَعْبِرِيمْ أَوْرَدْ المَبُوكَشْ. تَمِيدْ نَعْبِرِ لِفَدَهِيَّ 0 فَرْمَطَرِ 0 جَمْ لِ0 offset. mmap, brk, malloc מושתת באמצעות best fit, first fit שלמדנו בהם.

אמרנו שיש שני סוגי אזורי זיכרון, סוג ראשון זה אזור זיכרון מגובה קובץ כלומר מערכת הפעלה לינוקס נותנת למשתמש אופציית להחת קובץ רגיל שישבディיסק ולמפות אותו לזכרון ומאותו רגע גישה לזכרון תורגם לגישה לדיסק באופן ישיר למשתמש קובץ כזה נקרא מגובה לזכרון וכל פעולות קרייה וכן יתורגם לאזור המתאים בדיסק זה גם קורה בצורה עצלית כלומר שממפים קובץ מדיסק לזכרון לא מיד הולכים קוראים מדיסק את התוכן לזכרון אלא זוכרים את המיפוי זהה באשימת אזורי זיכרון, ברגע שמנסים לגשת לכתובות וירטואלית מתאימה קורה page fault ואז באמת את המידע מדיסק לזכרון. סוג שני זה אונוניימי כלומר איזור זיכרון איי לו שם שלא קשור אף קובץ למשל שעושים malloc לא מבקשים זיכרון לקובץ מסוים לנכון mmap משתמש ב mmap אונוניימי כלומר מעבירים לו בפרמטר 5 את fd ששווה -1 לאחר fd של קובץ שפתחנו.

כעת נלמד מוגננים עצנים לדוחות את העבודה כמה שאפשר.

### נתהיל מ copy on write

הוא קשור קשר הדוק לקריאת המערכת `fork`, נזכר ש`fork` משכפלת תהליך כוללם יוצרת עותק שלו וגם את מרחב זיכרון, כלומר הבן רוצה מרחב זיכרון משלו לכארה השימוש הנאיivi היה לבוא לחת את כל זיכרון של האבא ולהעתיק את זה לבן, העתקה הנאייבית הזאת הייתה בעיתוי מסותי סיבות כי היא איטית, למשל אם נרצה לשכפל את כל המידע והוא די גדול, או אם אין זיכרון, נזכר שהמטרה של הבן היא execv שהיטת טעינה קוד חדש لكن העתקה מיותרת לגמרי. מה שעושים להסוך זה copy and write, כלומר במקום להעתיק את כל הזיכרון מאבא לבן אנחנו געתיין "כайлר" כלומר נגרום לאבא ובן להציבו לאבא ובן לאותו זיכרון בדיק אבל גן עלייה כי יכולים לכתוב אליו, בהתחלה שנייהם מצביעים לאותם מסגרות פיזיות, אחד מהם מסה לכתב הזיכרון יהיה page fault ואז מערכת הפעלה מטפלת פה על ידי לחת את המסגרת המשותפת ותשכפל אותו ובאותו רגע האשלה חייבה להתנפץ, כלומר מה עושים זה פשוט page fault . ואז פתרנו בעיה של איטיות, גם הבעייה של execv ואל הבן יעשה אותו הוא יזרוק אותה ויתקן תוכנית חדשה לביצוע וחסכנו דyi טוב. נשים לב אל המסגרות הן לקריאה בלבד אז ניצחנו כי אך אחד לא יכתוב ואף פעם לא נעשה שכפול.

מתהיליפ מתהליק אבא ניתן לראות בהתחלה שיש שני מבני נתונים בגרעין לניהול זיכרון, טבלת דפים ורשימת זיכרון. נשים לב שטבלת דפים היררכית היא 4 רמות ולמעלה יש רק אחד סתם לנוחות. ניתן לראות שיש רמה תחתונה של טבלת הדפים

ודגל הכי חשוב לנו read/write ורואים שכניסה זו מחייבת למסגרת מס' 250 ומסגרת זו היא גם לקריאה וגם לכתיבה, חוץ מזה יש גם רשימת אזורי הזיכרון פה יש אзор מס' אחד שמכיל בתוכו את דף מס' 11 ומידע נשמר גם באזור וגם בדף.

אחרי fork קיבל משаг צזה בדיק,

וכל אחד יש לו טבלת דפים משלו, וכרגע בשלב הזה הם עותק זהה אחד של השני, בשביל להגן על מסגרת זו מניסיונית כתיבה מאב או בן או הדגל read/write נכבה גם באבא וגם בגין

ברשימה אזורי זיכרון אין שינוי עדיין, נשים לב שלמסגרת יש מונה שאומר כמה טבלת דפים מצביים אליה לנין ניתן לראות שמעלה יש 2. והוא נשמר בתוך מבנה שנראה טבלת המסגרות שנראית בהמשך. ניתן לראות שעכשיו בן אבא לא יכולים לכתוב חמסגרת זו, אם אחד מהם כן מנסה לכתוב מה שיקרה זה שבן יחשוף שלו מהסנית שלו או אבא והוא יחתוף page fault ורק מהו י└ך לטבלת דפים יראה שכתוב שם 0/w/a ואז מערכת הפעלה מקצת מסגרת חדשה כך

מעתיקה את התוכן אז משנה את המונה ומסדרת הרשאות של כתיבה וכתיבה של מי שחטא page fault למשל למעלה זה אבא ל1. נשים לב שהבן נשאר לא מעודכן כלומר 0 עושים את זה כך כי איןנו עצנים ולא מרצה לעשות עבודה שלא רוצחים, תמיד מנסים לא להתאמת ולא לדחות. אם בכלל זאת הבן בהמשך ינסה לכתוב למחזנית אז הוא גם יחתוף pagefault, אבל עכשיו אין טעם להעתיק את המסגרת אף לא צריך וכך בודקים אם המונה 1 ואם כן אז נראה שרק הוא מצביע אליה ואז משנים את הרשאה שלו כלומר z/w ל1 וסיימנו הטיפול.

נשים לב שימושו מאד חשוב אם למשל אחרי fork כלומר מצב הזה בהתחלב

ואז אבא מנסה לכתוב למסגרת 250 עכשו איך באבא יידע שזו מסגרת מוגנת בcopy and write רשות אורי הזיכרון, שאומרת שכן מותר לכתוב למסגרת זו למרות שבטבלת הדפים אסור ואז מביניפ שהה בגלל מגנון copy on write. לינוקס עושה את זה דרך רשות אורי הזיכרון.

מה היה קורה אם זה חוטים? אז כל אחד היה מציביע לאותו רשות זיכרון וגם אותו טבלת דפים אז אנחנו מסודרים.

המגנון הוא מחולק לשני שלבים, ההגנה היא שלב זול והשכפול הוא די יקר.  
האם בהכרח שיפרנו ביצועים?

לא תמיד אם הבן ואבא יכתבו לכל הדפים, אז לא בהכרח הגנו ואז אפילו בזבזנו זמן על הגנה, אבל מקרה מפוך שבן משתמש בexecu כן חסכנו עבודה די טובה.

האם מערכת הפעלה הייתה יכולה לתפוס את ה execv הראשון קצר בעיתוי  
כי לא יודעים כמה קדימה להסתכל ויכול להיות הרבה דברים בקובץ لكن לא עושים את זה אבל תיאורטי כן אפשר.

השיקף למעלה זה מה שכבר דיברנו, מתי לא צריכים להעתיק וכו'.

מנגנון די' חשוב.

נדמיין לנו שאנו בסוק סטטוס הצלחנו בהפעלה ורצינו צטום ואז מבינים ששכחנו להבין כספ', אז מתקשרים לאבא ומבקשים 10 אלף דולר בחשבון ונמשוך אותו מפה. ואז הוא אומר שם אבל כרגע הם וירטואלי, ככלומר אין לנו כספ' פיזי ביד, אז מנסים לرمות אותנו שהולכים לכיסופומט להוציא כספ' למשל רצינו 100 דולר מקבלים 100 דולר ולא ממש לוקחים הכל. וזה מה שבديוק מערכת הפעלה היא נותנת לזכרון מלא זיכרון וירטואלי מזכור גם שמערכת הפעלה מביאה זיכרון פיזי חיבור לאפס אליו כי אם למשל יש מידע של מישחו אחר אז יהיה בעיה security מאוד חמורה. לכן היא מביאב מסגרת מלאת אפסים שאין בה junk של מישמי אחר.

אית זה סותר `malloc` יכול להחזיר ערך זבל, וצריכים לעשות `free`, למרות שמערכת הפעלה מאפסת זיכרון שמקצת לנו, אז איך `malloc` יכולה להחזיר ערך זבל? היא קצת מרמה אותנו, למשל אם עושים מalloc ל-100 קילו בית ואז `free` שוב ושוב אז מערכת הפעלה אומרת לעצמה המשמש סתם מבלבל המוח, כל פעם מקצת ומשחרר וסתם מבזבז הרבה קריאות מערכת או היא עשו `malloc` ואז שעושים `free` היא אומרת לנו בסדר עשייתי אבל בפועל היא לא, לכן שאנו עושים מalloc שהקצתה בעבר יוכל להכיל ערך זבל אבל אף פעם לא נראה זבל של משתמש אחר.

נשים לה שיש `malloc` שלא קוראים ל`brk`, `mmal`, אלא משתמשת באזורי זיכרון שהקצתו בעבר ולא היו בשימוש.

דוגמא זואת מקצת מערך של chars בגודל 4 קיל בית כי כתוב 4096 ואנו חיבים כך בcplusplus של דף. גם מעבירים ||עט כדי שמערכת הפעלה תחזיר לנו כתובת זהה ב 6, גם הזיכרון הוא אונוני. היא לא באמת מקצת זיכרון פיזי אלא מעדכנת את רשימת אורי זיכרון כלומר מוסיפה אחד חדש ולא נוגעת בטבלת הדפים. אחרי כך ממשה לקרוא את הזיכרון, אנחנו חייבים לעדכן את טבלת הדפים, שבאים לקרוא זיכרון שהקצנו כרגע אמרורים לקרוא רק אפסים, אז במקום למצוא מסגרת פנوية לאפס אותה ולתת אותה לטליך אנו שומרים מסגרת של אפסים ונקרא לה ה zero frame. וברגע שמנסים לקרוא את הביצ' הראשון של המערך בתוך משתנה אנו הולך לטבלת דפים ונמלה את הדף הווירטואלי למסגרת אפסים, נשים לב שהມון תהליכיים שונים יכולים להציג למסגרת אפסים, אז אם טליך מנסה לקרוא מידע שהוא הקצתה הוא ייגש למסגרת האפסים, אבל מה אם הוא ינסה לכתוב אליה? אז מගנים עליה דרך copy and write מה שעושים זה אותו מנגן ונשים לב שבאור הזיכרון שמננו שהמסגרת ניתנת לכתיבה וטבלת הדפים תגיד שאי אפשר ואז משכפלים אותה לטליך ומעתיקים אפסים ומאותה רגע ניתן לכתוב אליה.

ניתן לראות שיש עוד דרך שאומרים לmap לא לא להיות עצמן כלומר להביא מסגרת ולאפס אותו במקום להשתמש במסגרת המאופסת מרוש וلتת לכל מיני תהליכיים להציג עליה ולהטוף page fault שירצוי לכתוב לזכרון המוקצת.

מתיוואר בו כל מה שכבר אמרנו, נסגור שיש mmap מגובת קובץ או אונוני, נזכיר שмагובת קובץ זה אומר שהזיכרון המוקצת הוא של קובץ ולא של משהו לא ידוע. אם למשל ננסה ב mmap ראשון לגבות קובץ ואז בקריאה ראשונה מערכת הפעלה מביאה מדיסק לזכרון את הקובץ, major page fault כזה נקרא page fault,

כִּי לוחקת הרבה מאייד זמן לעומת זאת חריגה שלא ניגשת לדיסק لكن אינה חוסמת את התהיליך ואז היא נקראת **minor page fault**.

מה היה קורה אם הכתיבה ישירה? היינו מקצים מסגרת יישירות, אין לנו `in copy` או `write` אז מערכת הפעלה מוציאה מסגרת מאפסת אותה ונוננת אותה.

איך מערכת הפעלה יודעת איזה סוג **page fault** זה כדי לטפל בה? או כל המידע האלה שומרים על מהסנית גרעין וגם על רגסטר מיוחד שאומר מה נסיבות החריגה,

או בגודול התרשים הזה מתארמתי **page fault** לא חוקי למשל שימושה מנסה לגשת לGrün או אפילו מערכת הפעלה מנסה לגשת לGrün ממצב גרעין ובמקרים כאלה בורגים תהליכיים ומערכת תקרום, מצבים אחרים הם שתהיליך מנסה לכתב למשל או לפתח קובך ואין זיכרון זהה חוקי לכל דבר ואז לטפל בזה משתמש ב.demand paging שיש למשל `fork` או `copy` and `write`

Present/Absent bit – Present or absent bit **says whether a particular page you are looking for is present or absent**. In case if it is not present, that is called Page Fault. It is set to 0 if the corresponding page is not in memory. Used to control page fault by the operating system to support virtual memory.

תרגול 11.

תרגול זה מכסה את התפר בין זיכרון לדיסק. נזכר שמערכת הפעלה חייבת לניהל 3 רכיבים שהם מעבד זיכרון ודיסק

זכור ש יש הבדל מהותי בין זיכרון לדיסק שהוא בזמן הגישה. למשל התקני אחסון כמו דיסק או התקני אחסון מטכנולוגיות חדשות יותר הם הרבה יותר איטיים מזכרון בסדרי גודל משמעותיים. והמשמעות אם התוכנית ניגשת לדיסק אז עלולה לסייע מזמן השהיה ארוכים ועיכוב בביצועים. מה שמערכות הפעלה עושים זה לשמר מבני נתונים מיוחד שנראה מטמון הדפים בזיכרון שהם בעצם מביא מידע מדיסק לזכרון ושומר אותו שם וזה יותק עיל בغال עיקרונו הלוקאליות. מסתבר שתוכניות ניגשנות שוב ושוב לאותו פרטי מידע גם שוב ושוב מבחינת הזמן וגם פרטי מידע סמכים שהוא לוקאליות למרחב. ולכן אם יהיה מטמון ששומר פרטי מידע הכיר הם מדיסק לזכרון אז זה יאיץ הביצועים.

אם זה קאצים 3|2, 1|1? התשובה היא לא, יש לנו המון מטמותנים וקרמים שהוא נושא מחקרי, נעשה קצת סדר. יש מטמוניים שהם בחומרה בלבד שמערכת הפעלה לא יודע. עליהם למשל 2|1|1 שהם מטמוני מעבד ומנהל אותם בלבד לא עזרת התוכנה או מערכת בפעלה או משתמש והפ קיימים ברמת מעבד. יש מטמוניים למשל `tlb` שמנוהלים בחומרה וגם על ידי מערכת הפעלה כלומר מערכת בפעלה מודעת אליהם ואיפלו מנקה את התוכן שלהם עשו להם `flush` ויש מטמוניים שלא קשורים לחומרה שמנוהלים אך ורק במערכת הפעלה למשל כמו מטמון הדפים שנלמד עליו והוא קובעת متى מכניס פרטי מידע אליו או מוציאים ממנו ו מבחינת מעבד הוא יכול לרווח הצורה תקינה לגמרי גם בלי מטמון דפים למערכת הפעלה, פשוט מערכות הפעלה גילו שהוא משפר ביצועים ולכן משתמשות בו.

מה זה אומר שבתקני אחסון יותר איטיים מזכרון, נכון לעכשיו גישה לזכרון זה 100 nano למשל `lps2` זמן גישה זה 16 mikro שניות, לעומת זו כונן של דיסק קשיח זה

3 מיל' שניות. זכרון יותר מהיר mass פֵי 100 או בכך כדי שייהה למערכת הפעלה פחו גישה להתקני אחסון על ידי שתשים הידע הכி חם במתמון גפים.

מתמון הגפים אוסף מסגרות בזיכרון הפיזי, (לא בהכרח רציפות).

כל פעולות הגישה לדיסק עוברו צדקה דרך מתומו הדיסק, למשל שהמערכת עושה read write מערכת הפעלה בודקת אם המידע נמצא במתמון הדפים אחרת אין ברירה הגרעין קודם כל צריך לקרוא מידע מdisk למטרון הגדים ולהוסיף את זה במתמון הדפים ואז לשרת את הבקשה במתמון הדפים. נשים לב שמערכת הפעלה מנהלת את כל מסגרות בזיכרון הפיזי ודואגת לשמר מסגרות למטרון הדפים. ובעצם יגיד גם כמה זמן המידע נשמר במתמון הדפים. בינהיים צריל לדעת כל פעולה קרייה או כתיבה היא במתמון הדפים. (מטרו הדפים זה לא אותו דבר כמו TLB).

### עיקנון הלוקאליות:

ניתן להפריד אותו לשני סוגים של עקרונות לקולאיות בזמן ובמרחב. בזמן אומר שכנראה ניגש לפרט מידע שוב ושוב בפרק זמן קרובים, אף זה כך כדי לשמר אותו במתמון הדפים. ואם זה כך אז כדאי לשמר אותו במתמון הדפים ואז אל נרצה לקרוא מהdisk אם המידע נמצא במתמון הדפים אז נחשוך גישה לדיסק ואם נרצה לכתוב לדיסק, אז לא חסכנו כלום כי מטרון הדפים לא עוזר לנו לכתוב לדיסק באמצעות לכתוב לדיסק, אבל עדין מטרון הדפים יכול לעזור כי גם במקרה של כתיבה יתכן שנעשה שלוש כתיבות ברצף ייכתוב 3 פעמים למטרון הגדים ורק פעם אחת לדיסק וגם כתיבה לדיסק ניתן לעשות רקע כלומר אסנכרונית טרי שכתבנו לזכרון. למרחב אומר שאנו בדרך כלל קוראים פרטי מידע סטטיים, למשל אם נרצה לקרוא קובץ שורה אחרי שורה או באיטרציה ראשונה מבאים מסגרת ראשונה 4 קילו בית מהו

הדיםק ומחזירים למשתמש את הראשונה הראשונה או שנרצה לקרוא שורה שנייה  
וכו הן יהיה באותה מסגרת שהבאו מדיםק לזכור لكن מטמון הדפים יעוזר לו  
בלוקאליות של מרחב.

הערה: אין קשר בין `cat` למטמון דפים אמרנו ש `cat` שומר כתובות וירטוואליות כדי  
להסוך גישה לדיסק ולא שומר ממש מידע שלו בדיסק. גפ הוא שונה מ-`3-1|a`  
קאצים שהם בכלל נמצאים על מעבד.