7,125

LAB 3

grep - copy
sed - search and replace

*regular expressions
*learn regular expressions table for test week 6
*teaching notes 4.1

* is a wild card, you can use it as grep " " *.txt -> meaning search in any file in the given directory that ends with .txt

GREP
-> case sensitive
-> use -i for it to be case insensitive

-match all lines that contain ana

```
grep "ana" input.txt
    ana
    1234ana
    stii unde e anabel
    afbhik kna anastasia acbdi

grep -i "Ana" input.txt
    ana
    1234ana
    stii unde e anabel
    afbhik kna anastasia acbdi
```

SED
-> just outputs the result, without replacing the original file
-> s/ is for substitute
-> /d if like a filter, deletes all files containing the given expression

```
sed "s/ana/maria/" input.txt
    maria
    1234maria
    stii unde e mariabel
    afbhik kna mariastasia acbdi

    nothing
    12345

sed "/ana/d" input.txt

    nothing
    12345

sed 'y/aei/@3!/' input.txt
    @n@
    1234@n@
    st!! und3 3 @n@b3l
    @fbh!k kn@ @n@st@s!@ @cbd!
```

```
    noth!ng
    12345
```

LAB 4

```
[abc] - takes them with an OR between them -> will return a line that
contains any of these characters
[a-z0-9] *matches ONE SINGLE character
+ -> one or more times
* -> zero or more times
    -> can be looked at like a while loop
? -> zero or more timess, binary -> yes or no
[a-z]{3} -> the sequence has to be matched at least 3 times
^ -> beginning of the line or not (^ana or [^a-z])

grep -E '/^ana/' n.txt -> searches for ana at the beginning of the
line
grep -E '/^.{3}$/' n.txt -> any line that has exactly 3 characters
    -> '/^...$' does exactly the same

cat /etc/passwd | wc -l -> takes the output and counts the lines

grep -E "^[a-z]{2}[02468]+:" /etc/passwd
grep -E "^..[02468]+:" /ect/passwd
    -> find usernames that contain only even numbers

grep -E "^([^:]*[aeiou][^:]*){2}:" /etc/passwd
    -> username has at least 2 vowels

grep -E "^([^:]+:){4}[^:]*\<ION\>" /etc/passwd
    -> find the students with the name ION
```

LAB 5

```
SED
    > sed -E    's/ ________ / ________ /'
                ^ search   ^ replace
    > sed -E 's/ [A-Z] / - / ' -> replace the first occurence on each
line
    > sed -E 's / [A-Z] / - / g' -> the flag "g" stands for global, it
replaces every occurence
    > sed -E 's / [A-Z] / - / gi' -> replace everything, case
insensitive
        -> using -i replaces the content of a file, not just prints it


* replace vowels with *
> sed -E 's/[aeiou]/*/' file.txt -> replaces just the first occurence
> sed -E 's/[aeiou]/*/g' file.txt -> replaces all of them globally
> sed -E 's/[aeiou]/*/gi' file.txt -> replaces everything case
insenstively

* add before each vowel !
    -> grouping 'sed/([aeioy])()()/*\1\2/'
    > sed -E 's/([aeiou])/*\1/gi' file.txt  -> adds before every vowel
*
```

```
* swap uppercase-lowercase groups
    > sed -E 's/([A-Z])([a-z])/\2\1/g' file.txt  *WITHOUT i

* double each vowel
    > sed -E 's/([aeiou])/\1\1/gi' file.txt

operator y/
    -> y/aeiou/@3!0w/
    -> mapping? not string replacement, but rather a correspondence

delete
    > sed -E /Ana/d file.txt -> deletes all lines that contain Ana


AWK
    -> basically a mini programming language
    -> print $smt prints a token that's split by a given delimitator

> awk '{print $2}' file.txt
    -> if that word doesn't exists, it prints a new line
    -> every line gets applied this print $2

* telling awk to define tokens by my given delimitator
> awk -F: '{print $2}' file.txt
> awk '{print NR, $0}' file.txt -> NR is the num of the line
> awk '{print NF, $0}' file.txt -> NF is the number of fileds
> awk '{print "start"} {print NF} {print "end"}' file.txt -> between
'' we can have as many bodies as we want
> awk 'BEGIN{print "start"} {print NF} END{print "end"}' file.txt ->
BEGIN prints it just at the start and END prints it once at the end

You can have a  __.awk file to format it better
> awk -f try.awk file.txt
  1  BEGIN{
  2    print "Welcome this is the beginning of the file"
  3 }
  4
  5 NR%2==0{print $0}
  6
  7 END{
  8     print "Okay byeee"
  9 }
    -> print all lines that are even

  1 BEGIN{
  2    print "Welcome this is the beginning of the file"
  3 }
  4
  5 NR%2==0&&length($2)>5{print $0}
  6
  7 END{
  8     print "Okay byeee"
  9 }
```

```
    -> print all even lines for which the second token is > 5
```

* write an awk script that will count how many users are 914 and print it

```
  1 BEGIN{
  2     cnt=0
  3 }
  4 $5 ~ / 914 / {
  5     cnt++
  6     print $5
  7 }
  8
  9 END{
 10     print cnt
 11 }
```

* piping, get the last name of people in 914 and replace vowels with *

```
> awk -F: -f ex.awk /etc/passwd | awk '{print $5}' | sed -E
's/[aeiou]/*/gi'
```

LAB 6

we have to be super precies with the interpreter

* for permision (user, group or other), use chmod +x filename or chmod 755 filename

```
 #!/bin/bash -> this is the interpreter <-> /bin/bash a.sh
 $1 -> first argument in the command line
 argv -> $@ -> all arguments in the command line
 argc -> $# -> number of arguments in the command line
```

tabs are important in bash, just like in python

$ works like a pointer in bash

B=1 would add a value to a variable, while $B would get the value of the variable

* if you want to use a variable in the command line, use $variable

a.sh

```
#!/bin/bash
echo "Hello World, $1!"

for A in $@; do
    echo "$A"
done
```

LAB 7

for most of the requirements, you have to think that there must be a sheel command that can do it (sed / grep / awk)

you can see grep as a function that provides filtering

```
for A in $@; do  // A is iterator, $@ are all the arguments
    // TABS
    // check if they are files or directories
```

```
    if test -f $A; then // -f is for file -> this is the "shell way of
doing it"
        echo "$A is a file"
    elif [-d $A]; then // -d is for directory -> more like a switch
case in other languages
        echo "$A is a directory"
    else
        echo "$A is not valid"
    fi
done
```

if you want to verify lower branches, you need to use the full path of the file

```
5.6.2.2
for A in $@; do
     if test -f $A; then
         echo "$A is a file"
     elif [-d $A]; then
         echo "$A is a directory"
     elif echo $A | grep -q -E '  ^[0-9]+$'; then
        echo "$A is a number"
     fi
 done

5.5.2.1
 count all lines from all files

 for A in `find . -type f`; do
    // when you want to associate a value do a variable don't use
spaces and "capture" the result with ``
    N=`grep -E '^[0-9]+$' $A | wc -l` // wc -l counts the lines
    S=`expr $S+$N` // expr is a shell command that can do operations
 done

find all files in a directory that have a size greater than a given
value
the 5th field of ls -l is the size of the file
the classical operators do not work for comparing numbers in shell!!
    -gt, -lt, -eq, -ne, -le, -ge
D=$1
S=$2
for A in `find $D -type f`; do
    N=`ls -l $A | awk '{print $5}'`
    if test $N -gt $S; then
        echo $A
    fi
done


WHILE
read from keyboard until the user types "stop"
5.7.1.3
== verifies 2 strings
```

```
  1 #!/bin/bash
  2
  3 while true; do
  4     read X
  5     if [ "$X"=="stop" ]; then // "" are important, since they CAST
the variable to a string
  6         break
  7     fi
  8 done
```

*script thar recieves a directory and checks for any changes into it
change can be at the file level or at the directory level, content or
details

	content	details
file	cat f	ls -l f
directory	ls -l d	ls -l -d d

-> build hash of all files and directories
-> hash is one way, you cannot go back

checking if the directory changed in any way

```
#!/bin/bash
D=$1
if [ -z "$D" ]; then
    echo "ERROR: No directory provided for monitoring" >&2
    exit 1
fi
if [ ! -d "$D" ]; then
    echo "ERROR: Directory $D does not exist" >&2
    exit 1
fi
STATE=""
while true; do
    S=""
    for P in `find $D`; do
        if [ -f $P ]; then
            LS=`ls -l $P | sha1sum`
            CONTENT=`sha1sum $P`
        elif [ -d $P ]; then
            LS=`ls -l -d $P | sha1sum`
            CONTENT=`ls -l $P | sha1sum`
        fi
        S="$S\n$LS $CONTENT"
    done
    if [ -n "$STATE" ] && [ "$S" != "$STATE" ]; then
        echo "Directory state changed"
    fi
    STATE=$S
    sleep 1
done
```

redirecting errors

LAB 9

1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

// create n child processes using fork and print pid of each child
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  for (int i = 0; i < n; i++) {
    int f = fork();
    if (f == 0) {
      printf("Child %d - PID %d - Parent %d\n", i, getpid(),
getppid());
      return 0;
    }
    else if (f < 0) {
      printf("Error creating child %d\n", i);
      return 1;
    }
  }
  return 0;
}
```

2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void f(int n) {
    if (n > 0) {
        if (fork() == 0) {
            printf("Child %d - PID %d - Parent %d\n", n, getpid(),
getppid());
            f(n - 1);
        }
        wait(0);
    }
    exit(0);
}

// create n child processes using fork and print pid of each child
int main(int argc, char *argv[]) {
  int n = atoi(argv[1]);
  //f(n);
  for (int i = 0; i < n; i++) {
      int f = fork();
      if (f == 0) {
          printf("Child %d - PID %d - Parent %d\n", i, getpid(),
getppid());
          wait(0);
      }
      else if (f == -1) {
          printf("Error\n");
```

```
          return 1;
      }
      else
         exit(0);
  }
  return 0;
}
```

LAB 13

```
#include <pthread.h>

pthread_mutex_ - init
               - lock
               - unlock
               - destroy

  | R  W
---------
R | 1  0    -> pthread_rwlock - init
W | 0  0                      - rdlock
                              - wrlock
                              - unlock
                              - destroy
```

SEMAPHORES

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntered..\n");

    //critical section
    sleep(4);

    //signal
    printf("\nJust Exiting...\n");
    sem_post(&mutex);
}


int main()
{
    sem_init(&mutex, 0, 2);
    pthread_t t1,t2, t3;
```

```
    pthread_create(&t1,NULL,thread,NULL);
    sleep(1);
    pthread_create(&t2,NULL,thread,NULL);

    sleep(2);
    pthread_create(&t3,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    pthread_join(t3,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

SEMINAR 2

the return value of a function
-> echo $? to see it
    -> 0 if all went well

any shell program begins with #!/bin/bash
    -> reference to the version of shell i wanna use
        bash - born again shell :)

    # <- comment usually, but at the beginning it's a DIRECTIVE

if an argument doesn't exits, bash considers it an empty string
$10 is not actually the 10th arg, rather the first one, to which we append 10
for it to give me the 10th, ${10}

```
>./1.sh 1 2 3 4 5 6 7 8 9 0
First argument: 1; Second argument 2
The 10th argument: 0
```

$* retruns all arguments as a string
$@ retruns it as an array

when assigning variables, DONT USE WHITE SPACES, because it thinks it's a command, and = is the arg

IFS=$'\n' -> internal field separator

```
if condition; then
fi
```

anything that's not a link, pipe, directory is essentially a regular file
for using grep to check if it's an int, we pipe it to it
    > du -b -> tells you the numb of bytes of a file

1.sh

```
#!/bin/bash
```

```
# write a shell script that takes any amount of arguments and prints
if thaiti':as the name of a
# - regular file
# - directory
# - positive base 10 integer
# - anything else

echo "First argument: $1; Second argument $2"
echo "The 10th argument: ${10}"
echo "Argument count: $#" # argument count
echo "All arguments: $*" # returns it as a string
# echo "All arguments: $@" -> $@ returns it as an array
# echo "All arguments: $#" -> argument count
for a in $@; do
    var=0
    if test -f $a; then
        echo "$a is a regular file"
        var=1
    fi
    if test -d $a; then
        echo "$a is a directory"
        var=1
    fi
    if echo $a | grep -E -q "^[0-9]+$";then
        echo "$a is an integer"
        var=1
    fi
    if test $var -eq 0; then
        echo "$a is something else"
    fi
done
```

2.sh

```
#!/bin/bash
# write a shell script that takes as arguments pairs of arguments
(filename, number)
# for each pair, check if the file size if less than the given num

size=0
while [ $# -eq 2 ]; do
    echo "args: $@"
    file=$1
    num=$2
    if [ -n $file ] && [ -f "$file" ]; then
        fsize=`du -s --block-size=1 $file | awk '[print $1]'`
        if [ $fsize -lt $num ]; then
            echo "File $file has size $fsize which is smaller than
$num"
        else
            echo "File $file has size $fsize which is greater than
$num"
        fi
        size=$(($size + $fsize))
```

```
    fi
    shift 2
done

echo "Total size: $size"
```

SEMINAR3
we've been working as if we're the only ones running on that machine
usually a machine has 1 processor, 8 cores, 16 threads

ps - ef | wc -l -> see how many processes are running
> ./1.sh in & -> runns the process in the background
when two programs are running in the background, they are running in parallel, therefore you connnot predict the order of the output
if those two processes are interracting with the same file or that interfere with one another, you might get a corrupted file
    > ./1.sh in & ./1.sh in & -> runns both the process in the background

moving foreward, we'll assume that running two processes at the same time it's gonna mess our files

```
 #!/bin/bash

 COUNT=0
 F=$1
 while [ $COUNT -lt 100 ]; do
     X=$(cat $F)
     X=$((X+1))
     echo $X > $F
     COUNT=$((COUNT+1))
 done

 execl VS execlp
```

EXECL
for execl, you have to specify the full path of the file you want to execute
```
     ❯ which ls
        /bin/ls
```
the initial argument is the name of the command you want to execute
*remember int main(int argc, char ** argv)
*the list of arguments must be terminated by a NULL pointer
```
    execl("/bin/ls", "ls", NULL);
```

if you want to add more arguments, you just do it like this
```
    execl("/bin/ls", "ls", "-l", NULL);
```
    ^ first step into running another process in a C program

    * exec remplaces the current process with a new process, so it it runs successfully, the current process will be terminated
    * in case exec fails, the current process will continue running
    -> this is called parasitic behaviour

```
    ! you should always check the return value of exec, see if it
failed or not
you can run whatever executable with execl
    execl("./1.sh", "1.sh", "./in", NULL);

EXECLP
execlp is looking for the path for us
    execlp("ls", "ls", "-l", NULL);

EXECV
-> not a variadic function, takes 2 arguments
execv(argv[1], argv + 1); -> ./exe /bin/ls -l
    -> this meets the criteria of the exec family having NULL as the
last argument
    * must have full path

EXECVP
execvp(argv[1], argv + 1); -> same, but full path is not needed

they all work in the same fashion, choosing which one to use is a
matter of preference or situation


CTRL + C -> sends a signal to the process, SIGINT which iterrupts it
* man 7 signal -> see all signals available in linux
by calling the function signal() you can catch the signal and do
something with it
    signal(SIGINT, handler);
    void handler(int sig) {
        printf("Caught signal %d\n", sig);
    }
    -> this will print "Caught signal 2" when you press CTRL + C

or signal(SIGINT, f) -> f is a function that will be called when the
signal is caught
    void f(int sig) {
        printf("Caught signal %d\n", sig);
    }
    signal(SIGINT, f);
becuase we've overwritten the default behaviour of the signal, we
can't interrupt the program with CTRL + C anymore
now you have to SIGKILL the process
 void f(int sig) {
        printf("Caught signal %d\n", sig);
        exit(0);
     }
    signal(SIGINT, f);


FORK()
-> creates a child process
-> everything after the fork() will be executed by both the parent and
the child, essentially creating two processes
everything gets copied, the DS, CS, but also EIP (instruction pointer)
```

```
fork returns a value if it was successful
if (fork() < 0) {
    perror("fork");
    exit(1);
} else if (fork() == 0) {
    printf("Child PID = %d - PPID = %d\n", getpid(), getppid());
} else {
    printf("Parent PID = %d\n", getpid());
}
int i;
wait(&i);
-> this will make the parent wait for the child to finish executing

1.sh
#!/bin/bash

COUNT=0
F=$1
while [ $COUNT -lt 100 ]; do
    X=$(cat $F)
    X=$((X+1))
    echo $X > $F
    COUNT=$((COUNT+1))
done
```

SEMINAR5

up.c

```
/ create a thread for each argumentprintf("thrd %d - arg %s\n", d->id,
d->s);
// each thread converts any lowercase letters in the argument to
uppercase
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

typedef struct {
    int id;
    char *s;
} data;

void* upcase(void *arg) {
    data *d = (data*)arg;
    char *copy = malloc(strlen(d->s) + 1);
    strcpy(copy, d->s);
    for (int i = 0; i < strlen(d->s); i++) {
        if (islower(d->s[i])) {
            copy[i] = toupper(copy[i]);
        }
    }
    printf("thrd %d - arg %s - copy %s\n", d->id, d->s, copy);
    return copy;
}
```

```
int main(int argc, char **argv) {
    int size = argc - 1;
    pthread_t th[size];
    data *args = malloc(size * sizeof(data));
    for (int i = 0; i < size; i++) {
        args[i].id = i;
        args[i].s = argv[i + 1];
        if (pthread_create(&th[i], NULL, upcase, &args[i])) {
            perror("Oh no");
        }
    }

    for (int i = 0; i < size; i++) {
        void *res;
        if (pthread_join(th[i], &res)) {
            perror("Error on join");
        }
        printf("Result from thread %d: %s\n", i, (char*)res);
        free(res);
    }
    free(args);
    printf("END: \n");
    for (int i = 0; i < size; i++) {
        printf("argv[%d] - %s\n", i+1, argv[i+1]);
    }
    return 0;
}
```

SEMINAR 6

```
*((int *) v) -> first casting, then dereference

we can have global variables

int n;
int *arr;

void *f(void *arg) {
    int index = *((int *) arg);
    arr[index] = rand() % 1000;

    // wait until all threads execute this line

    int other = rand() % n;
    while (other == index) {
        other = rand() % n;
    }

    int val = arr[other] / 10;
    arr[other] += val;
    arr[index] -= val;
    return NULL;
}
```

- BARIER

```
    pthread_barrier_t bar;

    pthread_mutex_t mtx;

    void *f(void *arg) {
     int index = *((int *) arg);
     arr[index] = rand() % 1000;

     // wait until all threads execute this line
     pthread_barrier_wait(&bar);

     int other = rand() % n;
     while (other == index) {
        other = rand() % n;
     }

     // protect the shared variable when accessing it
     pthread_mutex_lock(&mtx);
     int val = arr[other] / 10;
     arr[other] += val;
     arr[index] -= val;
     pthread_mutex_unlock(&mtx);
     return NULL;
    }

int main() {
    srand(getpid());
    n = atoi(argv[1]);
    arr = malloc(n * sizeof(int));
    pthread_barrier_init(&bar, NULL, n);
    pthread_mutex_init(&mtx, NULL);
    pthread_t th[n];
    for (int i = 0; i < n; i++ ) {
        if (0 != pthread_create(&th[i], NULL, f, &i)) {
            perror("Cannot create thread");
            exit(1);
        }
    }

    for (int i = 0; i < n; i++) {
        pthread_join(th[i], NULL);
    }
    pthread_barrier_destroy(&bar);
    pthread_mutex_destroy(&mtx);
    for (int i = 0; i < n; i++) {
        printf("arr[%d] ", arr[i]);
    }
    free(arr);
    return 0;
}
```

```
!!! passing here i, allocated on the stack, always the same address,
so we need to allocate it on the heap
    int index[n];
    for (int i = 0; i < n; i++ ) {
         index[i] = i;
         if (0 != pthread_create(&th[i], NULL, f, &index[i])) {
             perror("Cannot create thread");
             exit(1);
         }
     }

! every time a barrier hits 0, it is reset to the initial value

- CONDITIONAL VARIABLES
pthread_cond_t cond;
 pthread_cond_wait (&cond, &mtx);
    -> unlocks the mutex and waits for a signal
    -> must be called with the mutex locked

    LOCK (mtx)                 ------ Unlocks the mutex
    WHILE (condition) {        |  ---- thread waits for a signal
        WAIT (cond, mtx); -------|
                                |  ---- thread is woken up
                                | ------ unlocks the mutex
    }

    -> if you call a signal with a locked mutex, you must unlock it
before calling the signal


SEMAPHORE
sem_t sem1, sem2;
sem_init(&sem1, 0, n/2); -> the flag in the middle is 0 for local, 1
for global
                        -> the last argument is the initial value of
the semaphore
sem_init(&sem2, 0, 1); -> is just a mutex

sem_post(&sem1); -> increment the semaphore
sem_wait(&sem1); -> decrement the semaphore
```

SEMINAR 7

Conditional variables

- one or more treads can wait for a condition to be true

pb 27 : Write a C program that takes two numbers, N and M, as arguments from the command line. The program creates N "generator" threads that generate random lowercase letters and append them to a string with 128 positions. The program will create an additional "printer" thread that that waits until all the positions of the string are filled, at which point it prints the string and clears it. The N "generator" threads must generate a total of M such strings and the "printer" thread prints each one as soon as it gets to length 128.

shell exam
begin line: ^
end line: $
begin word: \<
end word: \>

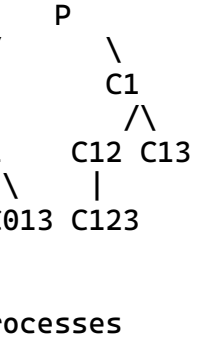
2. write a regex that takes the multiple of 5 + 2 (ex: 2, 7, 12)
    > grep -E "^(.{5})*..$"

3. give the operators for the following shell verification:
    a. string is not empty
        [-n $var]
    b. argument is a regular file
        [-f $var]
    c. is different (numerical)
        [$n1 -ne $n2]

4. how would you restore the value of a file descpriptor overwritten by dup2?
    save the initial value of the file descriptor before calling dup2

5. 5

6. fork returns 0 or smt positive if there's a child



=> 9 processes

7. only A

LECUTRE 3
"in" file
"out" file
* daca o comanda se termina cu codul de retur 0, atunci comanda este True (no error)

in shell, toate lucrurile se rezuma la codul ascii

constructia IF
$@ -> toate argumentele din linia de comanda, unul dupa altul
[-f $A] -> comanda test
#!/bin/bash
for A in $@; do
    if [-f $A]; then
        echo $A is a file
    elif [-d $A]; then

```
        echo $A is a dir
    elif echo $A | grep -Eq "^[0-9]+$" then
        echo $A is a number
    else
        echo We do not know what $A is
    fi
done

[ -z "$F" ] -> nu e vid
[ -r "$F" ] -> "citibil"

md5, sha -> o functie care primeste ca argument un fisier super mare,
care au ca output un sir de 160bits
hashing -> nu s-au gasit doua fisiere care sa aiba acelasi output
-> find > tipareste toate fisierele din acest director
-> [ -n "$STATE" ] lungimea nenula

*numele unui director, vreau sa anunt cand acesta a fost modificat
#!/bin/bash
D=$1 // numele directorului, primul argument
if [ -z "$D" ]; then
    echo "ERROR: No directory provided for monitoring" 1>&2 // iesirea
standard acelei comenzi echo, se va pune pe acealasi fisieri de erori
standard
    exit 1
fi
if [ ! -d "$D" ]; then
    echo "ERROR: Directory $D does not exist" >&2
    exit 1
fi

STATE=""
while true; do
    S=""
    CONTENT=""
    for P in `find $D`; do
        if [ -f $P ]; then
            LS=`ls -l $P | sha1sum`
            CONTNET=`sha1sum $P`
        else
            LS=`ls -ld $P | sha1sum`
            CONTENT=`ls -l $P | sha1sum`
        fi
        S="$S\n$LS $CONTENT"
    done
    if [ -n "$STATE" ] && [ "$S" != "$STATE" ]; then
        echo "Directory state changed"
    fi
    STATE=$S
    sleep 1
done

*anuntarea userilor cu prea multe procese
#!/bin/bash
```

```
maxim = $1
for user in `who | awk '( print$1 )'`; do
    if [ 0`ps -u $user | wc -l` -ge 0$maxim ]; then # 0 daca cumva nu
se da $1
        echo $user
        write $user <<MESAJ
        Aveti prea multe procese
    MESAJ
    fi
done
```

*din directorul curent, sa se determine primul fisier text care contine o linie al carei prim cuvant are cel mult 5 caractere
-> comanda file * - pentru directorul curent
    -> putem decide ca un fisier este text daca comanda file returneaza pentru el "ASCII text"

```
#!/bin/bash
    for x in * ; do
        if [ `file $x | grep -ci "ASCII text"` -eq 0 ] ; then continue
; fi
        cat $x | while read cuv1 t; do
            if [ ! -z $cuv1 ] && [ `expr length $cuv1` -ge 5 ]; then
                echo In $x s-a gasit $cuv1 cu lungimea `expr length
$cuv1`
                break
            fi
        done
        break
    done
```

LECTURE 4

* is anyone gonna mess with my data while i use it -> the new 2AM formula

ex:

```
    load AX, n
    inc AX
    store AX, n

    A | B
    L
        L
    I
        I
    S
        S
```

    this way the data is compromised, since it happened at the same time. your value was not incremented

& at the end of a line runs it in the background

how do you have multiple programs running on the same processor at the same time?
    -> a program is a file in execution
    -> every few seconds, the processor interrups the execution and chooses another program, puts in onto the cpu

bash knows how to create a process, all commands are essentially that :)

```
------------------
       ctx             --> context
------------------
    source code                     fork() starts another process
------------------                          -> it makes a copy of the executable
    constants
------------------
    variables
------------------
      heap


      stack
------------------
```

-> output of 1.c

```
 #include <stdio.h>
 #include <unistd.h>

 int main(int argc, char** argv) {
     printf("before %d %d\n", getpid(), getppid());
     fork();
     printf("after %d %d\n", getpid(), getppid));
     return 0;
 }
```

```
    before 12173
    after 12173
    after 12174

    before 12199 11884
    after 12199 11884
    after 12200 12199
```

getpid() -> get the id of the child program
getppid() -> get the id of the parent program

```
         P
      /  |  \
     C0  C1  C2
  /  |  \
C01 C02 C03
```

-> for a for loop i=0,3

```
    before 12251 11884
    after 12251 11884
    before 12251 11884
    after 12251 11884
    before 12251 11884
    after 12252 12251
    before 12252 12251
    after 12253 12251
    before 12253 12251
    after 12251 11884
    after 12252 12251
    before 12252 12251
    after 12254 1
    after 12255 12252
    before 12255 12252
    after 12253 1
    after 12252 1
    after 12256 1
    after 12257 12252
    after 12255 1
    after 12258 1

* fork() returns
    -> 0 in the child
    -> child PID in the parent

-> output for 2.c
 #include <stdio.h>
 #include <unistd.h>
 #include <stdlib.h>

 int main(int argc, char** argv) {
     int pid;
     pid = fork();
     printf("before\n");
     if (pid == 0) {
         printf("Child specific code pid =%d for fork return=%d.\n",
getpid(), pid);
         exit(0);
     }
     printf("parent specific code child PID=%d.\n", pid);
     wait(0);
     return 0;
 }

    before
    parent specific code child PID=12546.
    before
    Child specific code pid =12546 for fork return=0

wait() does not specify witch child to wait for
    -> it's job is to wait for the children processes to finish
* all children need to die before the parent finishes the job
```

```
-> if google were to be implemented like this, only one process could
be done at a time
while(1) {
    get request
    process request
    send back response
}

-> in this version, getting the request is the only thing that's done
one by one
while(1) {
    get request
        if (fork() == 0) {
            process  request
            send response
            exit(0);
        }
        // wait(0);
}

zobies -> children that don't die, because wait is not called
you cannot really add a wait() here but it's gonna be very
inefficient. since it gets a request, but does nothing with it until
all children processes finish
```

LECTURE 5

```
children agian??

while(1) {
    get req
        if (fork() == 0) {
            process req
            send response
            exit(0);
        }
}

when hitting ctrl c to interrupt the prgram, that's a signal
same goes for ctrl z which places it in the background

with few exceptions, all signals stop the program
* man signal

make a program that refuses to stop
    we'll use 2 function, signal and kill
    kill sends signals, signal stores it
    TO KILL THE PROCESS
        ❯ kill -SIGINT 8436
        ❯ kill -9 8436
    kill is a function that takes the signal and the runtime (?) id

both wait and sleep stop the cpu
    wait on the other hand waits for the child process to finish
```

```
    void f(int signal){
        wait(0);
    }

    signal (SIGCHILD, f);
    while(1) {
        get req
         if (fork() == 0) {
             process req
             send response
             exit(0);
         }
    }
    SIG_IGN -> ignores the signal
    SIG_DEF -> default

Exec*
       | absolute path | name to be searched in path
-------------------------------------------------------
       |                |
array  |    execv       |         execup
       |                |
-------------------------------------------------------
       |                |
list   |    execl       |         execlp
       |                |

export PATH=.:$PATH
^ this doesn't change the path, it prefixes it
export PATH=.
^ this changes the path

execlp("grep", "grep", "-E", "abc", "/etc/passwd")
char *a[] = {"grep", "grep", "-E", "abc", "/etc/passwd", NULL}
execup("grep", a);

*rewrites echo
 #include <stdio.h>
 #include <unistd.h>

 int main(int argc, char** argv) {
     execlp("echo", "echo", "asdf", NULL);
     printf("we echoed\n");
     return 0;
 }

 creating links between files: > ln -s ./a x
 specify the program you want to run twice when using execlp!!


 1  2  3  4  5  6
 ----  ----  ----
  3     7     11
    ----
```

```
     10
      ----------
          21

^ how to add processes fast
-pipe
-fifo
-ipc -> shared memory (SHM)
     -> message queue (MSG)
     -> ?? smt (SEM)


PIPE
    > ALWAYS CLOSE THE PIPE ENDS AS SOON AS YOU DON'T NEED THEM
    > close() doesn't actually close the pipe, but rather the pipe
handlers

LECTURE 7
read
    -> syncronisation process, not your usual read/write
    - extracts the data (doesn't leave it in place)
    - waits when empty (doesn't give an error either)
        - until some data
        - until no writers
    * read doesn't give you what you want, but rather what it can

write
    - adds data (might overwrite)
    - waits when full
        - until some space
        - until no readers

* none of them gives an error
-> Why doesn't read read as many bytes as i ask it to?
    maybe there aren't that many
    how long are you gonna wait for it?


open (FIFO)
    - waits until FIFO opened for complementary operations
    O_NDELAY (fcutl) -> flag that can be passed to open and it does
NOT wait for anything

pipes are ONE DIRECTION!!
pipes and FIFO are communication channels

IPC - inter processor communication (SHM, MSG, SEM)
 -> mechanisms at the processor level that allow communication between
them
 when you create this, try the ip to be unique
 -> ipcmk
 - clean up after yourself :)
    > ipcmk --shmem 1024 --mode 600 (make a shared memory)
    > ipcs (see it)
```

```
    > ipcrm -M [key] (remove it)

how does it work?
    -> create it
    -> where you open it, you get a handler

shmget has no syncronization, it doesn't wait!
by the time b is ran, you have no more shared memory either

h.h
 #include <stdio.h>
 #include <sys/ipc.h>
 #include <sys/shm.h>
 #include <stdlib.h>
 #include <time.h>

 struct absp {
     int a;
     int b;
     int s;
     int p;
 };


a.c
 #include "h.h"

 int main(int argc, char ** argv) {
     int shmid;
     struct absp* x;

     shmid = shmget(1234, sizeof(struct absp), IPC_CREAT|0600); //
create a shared memory at ip 1234 and give it permition 0600
     x = shmat(shmid, 0, 0); // pointer to that memory, shmat just
connects it
     srand(time(NULL)); // current time in seconds since 1970 :)
     while (1) {
         x->a = rand() % 100;
         x->b = rand() % 100;
         if (x->p == x->s)
             break;
     }

     shmdt(x);
     shmctl(shmid, IPC_RMID, NULL);
     return 0;
 }


b.c
  #include "h.h"

  int main(int argc, char ** argv) {
      int shmid;
```

```
      struct absp* x;

      shmid = shmget(1234, 0, 0);
      x = shmat(shmid, 0, 0); // pointer to that memory
      srand(time(NULL)); // current time in seconds since 1970 :)
      while (1) {
          x->p = x->a * x->b;
          x->s = x->a + x->b;
          if (x->p == x->s)
              break;
      }

      shmdt(x);
      return 0;
  }
```

threads are mechanism that take a function and runs them WITH the current process, in parallel

- -> unlike fork(), you don't clone the whole process
- when creating a thread, you create a new stack

```
    ctx
---------------
    source
---------------
    const
---------------
    variables
---------------
    heap

    stack
```

THREAD

```
 #include <stdio.h>
 #include <pthread.h>

 void* f(void* a) {
     printf("f\n");
     return NULL;
 }

 int main(int argc, char ** argv) {
     // threads will have a handler, but there's no child-parent
relation here
     pthread_t t;
     pthread_create(&t, NULL, f, NULL);
     // now we have 2 threads, main and f that work in parallel
     printf("main\n");
     pthread_join(t, NULL);
     return 0;
 }
```

one argument is enough since you can pass a struct with all the arguments you need

LECTURE 8

-race condition
n -> critical
    variable
    resource
n++ -> critical region

mutex (mutual exclution)
rwlock (read-write lock)
contional variable
barriers
semaphores

! surround critical region with mutex lock and unlock

LECTURE 10

prb 1: you have a bunch of kids trying to go into a bathroom stall
trylock() -> either locks or returns a value
    -> problem with kids and bathroom stalls
    for (int i = 0; i < 3; i++) {
        lock(&m[i]);
        1 go
        unlock(&m[i]);
    }
    -> this will make everyone wait for the first stall, not distribute them

prb 2: elevator -> more kilos than allowed, the elevator keeps locking

prb 3: trampoline -> only a specific number of kids can jump at a time

prb 4: forest -> bees and bears
    -> bees make honey, bears eat honey
    -> ranger kicks in when there's no honey

forest.c

```
#include <stdio.h>
#include <pthread.h>
#include <seamaphore.h>
#define BEES 1
#define BEARS 5

int honey = 100;
pthread_mutex_t m;
pthread_cond_t c;
sem_t s;

void* bear(void* a) {
    while (1) {
        pthread_mutex_lock(&m);
        if (honey >= 10) {
            printf("-");
            honey -= 10;
```

```
        }
        else {
            printf("!");
            pthread_cond_signal(&c);
        }
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

void* bee(void* a) {
    while (1) {
        pthread_mutex_lock(&m);
        printf("+");
        honey++;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

void* ranger(void* a) {
    while (1) {
        pthread_mutex_lock(&m);
        while (honey >= 10) {
            pthread_cond_wait(&c, &m); // wait -> unlock | waits for
signal | lock
        }
        printf("H");
        honey += 100;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i;
    pthread_t bees[BEES], bears[BEARS], rangert;
    pthread_mutex_init(&m, NULL);
    pthread_cond_init(&c, NULL);
    sem_init(&s, 0, 0);
    for (i = 0; i < BEES; i++) {
        pthread_create(&bees[i], NULL, bee, NULL);
    }
    for (i = 0; i < BEARS; i++) {
        pthread_create(&bears[i], NULL, bear, NULL);
    }
    pthread_create(&rangert, NULL, rangert, NULL);
    for (i = 0; i < BEES; i++) {
        pthread_join(bees[i], NULL);
    }
    for (i = 0; i < BEARS; i++) {
        pthread_join(bears[i], NULL);
    }
    pthread_join(rangert, NULL);
```

doesn't go straight to producing honey, but rechecks

```
    pthread_mutex_destroy(&m);
    pthread_cond_destroy(&c);
    sem_destroy(&s);
    return 0;
}
```

LECTURE 11

Operating Systems Structure

KERNAL

Tech book keeping

```
          interrupts
         /  (talk to the hardware)
hardware - process manager          -> memory management  -> phisical
I/O  -> File System  -> Job planning & resource allocation
         \  (chosses processes)
          CPU dispathcer
           (?)
```

Book keeping

```
    Compiler + Assemblying + Linkers + Loaders
    Interpreters + Macro processors + Editor + Debuggers     -> IDEs
    Utils + Libraries + DataBases + Other
```

Processes

---

states

```
        HOLD
         |
  ----> READY <---
 |      | |       |
WAIT <- RUN  -> SWAP
         |
        FINISH
```

Scheduling

```
    - FCFS (FIFO order)
    - SJF (Small Jobs First)
    - Priorities
    - Deadline scheduling
```

DeadLock

LECTURE 12

Conditions that make dead lock possible

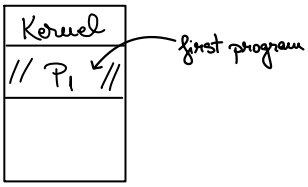
1. mutual exclusion
2. lock & wait
3. non-preemption
4. circular wait

ALWAYS LOCK THE RESOURCES IN THE SAME ORDER
↑ to prevent deadlock
+ open fifo's in the same order
+ make sure you don't have cycles
+ sem capacity is less than all threads (sumo elevator problem)
+ barrier is never completed ⇒ no. of threads is not a multiple of the barrier init

Memory management

- real → single tasking OS (A)
  → multitasking OS
  - fixed partition * absolute (B)
    * relocatable (C)
  - variable partitions (D)
- virtual → paged (E)
  → segmented (F)
  → paged - segmented (G)

A. Single tasking OS



- the compiler hard codes physical memory addresses in the executable

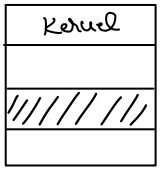
Address calculation:

a.c → a.o → a.exe → process
(after compile we get an object file)
b.c → b.o
compile
link
exec.
here the variable gets an assigned "spot" in the memory

* there's a tight connection between the .exe and the machine, since the compiler knows exactly how big the kernel is

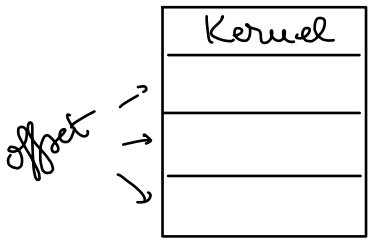
* stopped working for us once we were running more programs

B.



→ split into memory partitions
→ compile into certain partition

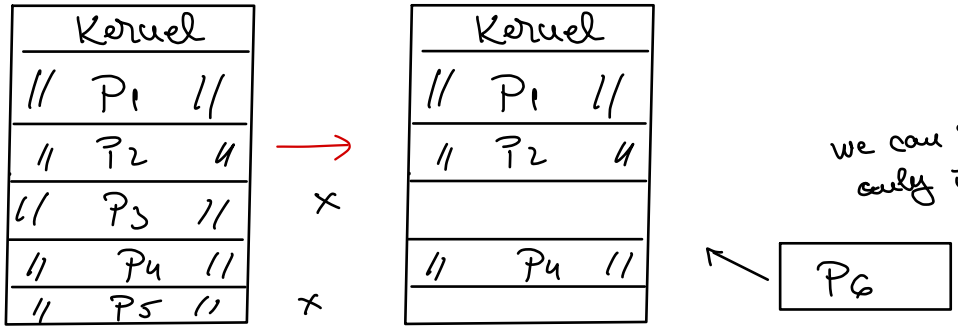
C. (change compiler and improve address computation)



→ no hardcoding of physical addresses
→ every partition has an offset
→ relative addresses (relative to the beginning of the partition)
→ the program can run in any partition (that is free)

* Binary address - offset
Partition address - partition start + BA

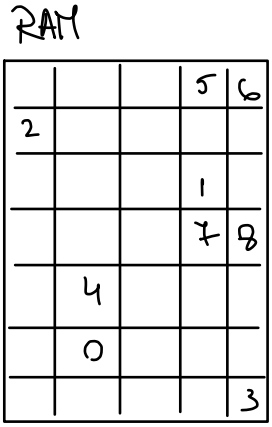
D. partitions are not predefined, we define them based on the size of the program that has to start (each new process starts after the last one)



we can run P6 only if we have the continuous memory necessary

We have the necessary memory but it is SPLIT ⇒ fragmentation

E.



real pages (fixed size of a page)

Prog

0	1	2
3	4	5
6	7	8

virtual pages

→ when the process stops, the real pages are freed
→ fragmentation solved with more fragmentation
→ more complex address calculation

needs a search

→ we need a table for each virtual page position in the real pages ⇒ slow
BUT the processor help with hardware

— virtual address = virtual page + offset
⇒ we fixed fragmentation

F. segments do not solve fragmentation, segments only group sections to be protected

they do not have a fixed size | provides memory access protection
→ not meant to organize your code + address calculation similar to page

G. TOP but waste a bit more memory than just paged

every segment split into pages
→ address : segment table + page table → virtual address (s, vp, offset)

Loading policies:

→ when should pages be loaded?

- load all of them at process start
  (slower start and wasted RAM) loading pages that might never be used
- load when needed (a location from it is referred)
  (even ~~slower start~~ faster start + no unused pages loaded and slower execution)
- load the requested page and a few neighbouring pages
  (chances are they will be needed)

Neighbouring principle (if it requests a page, it is likely to soon request its neighbouring pages)
↳ what we use today

Unloading policies:

RAM

	0		
3			1
	2		

Prog:

0	1
2	3

LECTURE 13

Unloading policices / replacement policies

-> RAM memory

RAM	Program
- - - -	0 1
- 0 - -	2 3
3 - - 1	
- - - -	
- 2 - -	

* choose something unlikely to be needed soon

-> What if the memory is full?
- you splill some data to disk

-> How to decide what to spill?
? FIFO - little to no prediction (what if the first one is a c library?) or the printing to screen

- NRU (Not Recently Used) - use 2 bits to track the usage of each page (that are periodically set to 0)
  1 - 00 - not used recently, not modified, not read, not written
  3 - 01 - not used recently, modified, not read, written
  2 - 10 - used recently, not modified, read, not written
  4 - 11 - used recently, modified, read, written
  -> you use the file from where you read more often then the one you write to * Choose a victim from the smallest class

LRU - last recently used
- N pages of RAM
- NxN bits matrix
- when a page is accessed, its line is filled with 1, it's column with 0
- the victim is the page with the minimum line total

	page		page		page 2	
0 0 0 0	->	0 0 0 0	->	0 0 0 0	->	0 0 0 0
0 0 0 0		1 0 1 1		1 0 1 0		1 0 0 0
0 0 0 0	1	0 0 0 0	3	0 0 0 0		1 1 0 1
0 0 0 0		0 0 0 0		1 1 1 0		1 1 0 0

Allocation policies / Placement policies
---
where do we malloc?

|__|__|_____ L _____|

allocating a block of memory

*next = NULL *next = NULL
|__|__|_____ L1 _____|__|__|_____ L2 _____|
^Allocated list ^Free list

- Where do you place a malloc request?
  -> First Fit (choose the first place big enough)
    - fast
    - but you make no attempt to minimize fragmentation
  -> Best Fit (choose the ~~tightest fitting space~~)
    - slower
    - very small slices of free memory are created and left behind => fine grained fragmentation
  -> Worst Fit (allocate the largest one available)
    - slow

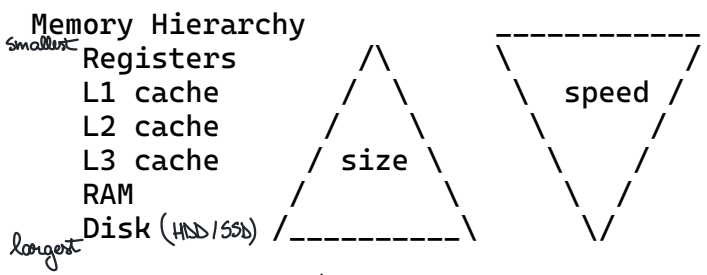
Buddy fit →
- leaves the largest possible chunks of free memory

$2^n = 2^0 + 2^0 + 2^1 + 2^2 + \ldots + 2^{(n-1)}$

↳ always allocate a multiple of 2 / keep lists for each power 2 of free spaces / allocate the closest / split the remaining list across the other list

What's the point of the cache?
- to reduce the time it takes to access data
- you keep stuff that you use often in the cache, close to you

Memory Hierarchy (smallest → largest)



Cache → small + fast, size C pages

RAM → large + slow, size R pages

Direct swapping / Direct cache

cache location = page number % cache size

cache size = 16

0 8 17 1 33 49 2

0 8 1 1 1 1 2

* you cancel the cache's usefulness, since there's collision

Associative swapping

place the page in the first empty slot

* organize cache in groups of pages, choose a group with %, search for page / free spot within the group

Set Associative swapping

C=8

k → k%4 and search

0 | 0 8 2 . . .

1 | 17 1 33 49 . .

File Systems

- data is stored in blocks
- file - a sequence of blocks

I-Node

| inode data | 0 | 1 | . . . | 9 | 10 | 11 | 12 |

*pdf mara

Directory

Name	I-Node
a.txt	17
b.c	23
dir/	108

* poza links? ask mara

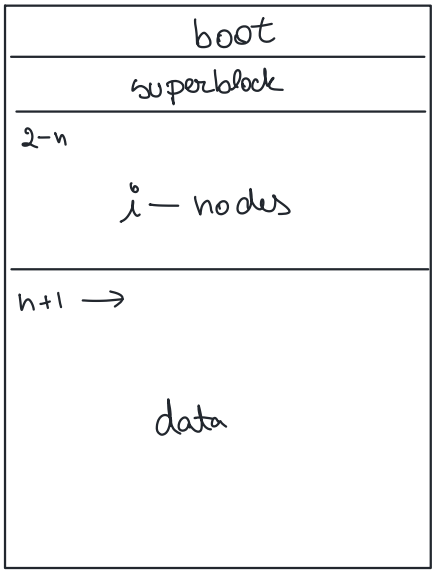
Remounting Volumes

/: | A | B |

B: | X | Y |

stack

/: | X | Y |



* each file has an i-node as its entry
* a directory is just a file that contains pointers to other files

* how large can a file be?

if: address size A

block size N*A

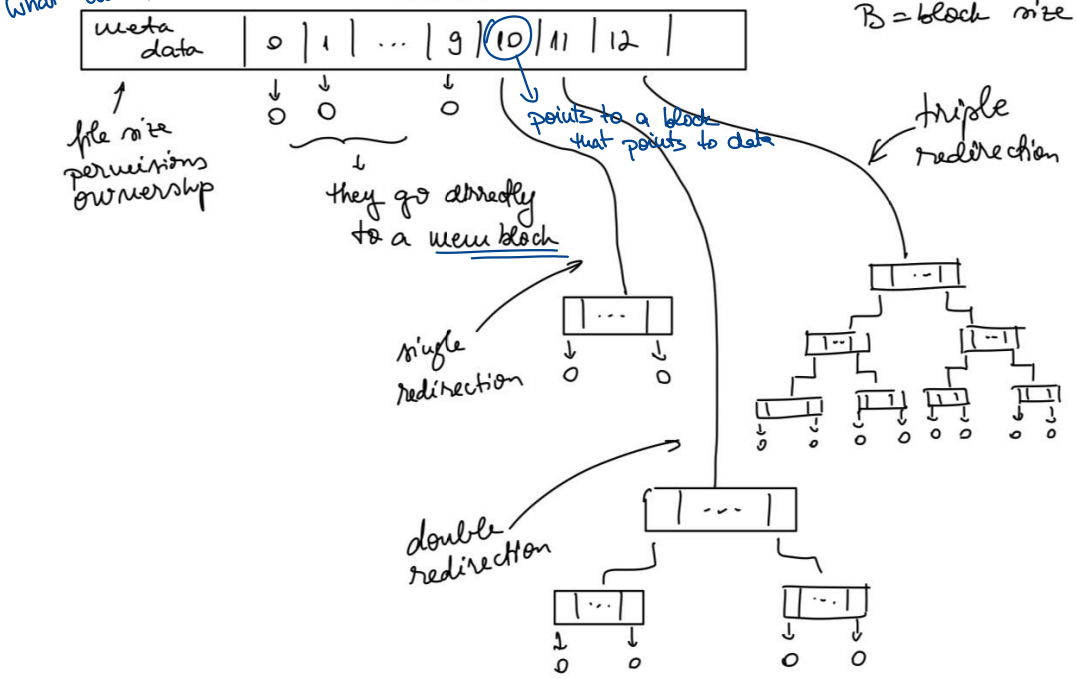
⇒ $10*N*A + N^3*A + N^4*A$

UNIX FS: → data is stored in blocks // interfață între sistemul de fișiere și locația efectivă
(file system) file = collection of blocks

I-node → keeps track of where the data is

what an i-node looks like ← addresses →

A = address size
B = block size



* se poate face legătură hard dacă face parte din același partiție

* diferența dintre un link simbolic și un link hard?

o legătură simbolică nu e o copie a fișierului

linkul e ca un "pointer" spre fișier

dacă ștergem fișierul original, un link soft nu mai are spre ce să pointeze
DAR un link hard e okay, pentru că e copie a inode-ului

un link soft pointează spre inode-ul fișierului

ln -s [fișier] → soft link

* dacă modificăm link-ul hard se modifică și fișierul de pe disk

* ce conține un fișier de tip director în Linux?

Fişierele directori. Un fişier director se deosebeşte de un fişier obişnuit numai prin informaţia conţinută în el. Un director conţine lista de nume şi adrese pentru fişierele subordonate lui. Uzual, fiecare utilizator are un director propriu care punctează la fişierele lui obişnuite, sau la alţi subdirectori definiţi de el.

Fişierele speciale. In această categorie putem include, pentru moment, ultimele 6 tipuri de fişiere. In particular, Unix priveşte fiecare dispozitiv de I/O ca şi un fişier de tip special. Din punct de vedere al utilizatorului, nu există nici o deosebire între lucrul cu un fişier disc normal şi lucrul cu un fişier special.

Fiecare director are două intrări cu nume speciale şi anume:
"." (punct) denumeşte generic (punctează spre) însuşi directorul respectiv;
".." (două puncte succesive), denumeşte generic (punctează spre) directorul părinte.

Fiecare sistem de fişiere conţine un director principal numit `root` sau `/`.

* deadlock

↳ un proces sau thread asteapta ca o resursa sa fie eliberata de un proces care la randul lui asteapta

```
fifo1 = open("f1", O_RDONLY);
fifo2 = open("f2", O_WRONLY);
```

```
fifo2 = open("f2", O_RDONLY);
fifo1 = open("f1", O_WRONLY);
```

Procesul 1 asteapta dupa 2 si invers

sau

fara mutex ul locked la iesire din thread

* signal

↳ handler + cum se inregistreaza

↳ semnalele care nu pot fi ignorate

signal (SIGINT, handler);

↳ asociaza interupt-ul cu functia handler.

Producer / consumer problem

threads communicating through pipe
i want the producer to wait if buffer is full
& consumer to wait if buffer is empty

2 semaphores: full(0) empty(N)

P	C
wait (empty)	wait (full)
lock (m)	lock (m)
produce	consume
unlock (m)	unlock (m)
post (full)	post (empty)

---

to see the difference between 2 directories you can use { diff / cmp }

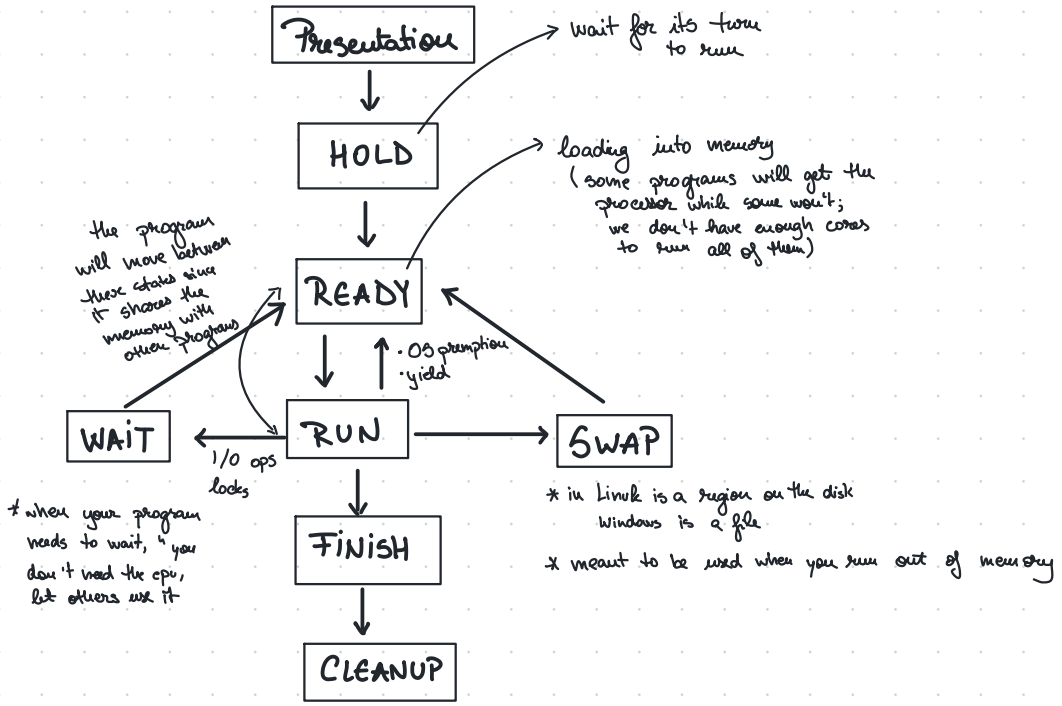
diff goes line by line and cmp goes byte by byte

```
find "$1" -type f | while read F; do
    find "$1" -type f | while read G; do
        if [ "$F" != "$G" ] && cmp -s "$F" "$G"; then
            echo "$F = $G"
        fi
    done
done
```

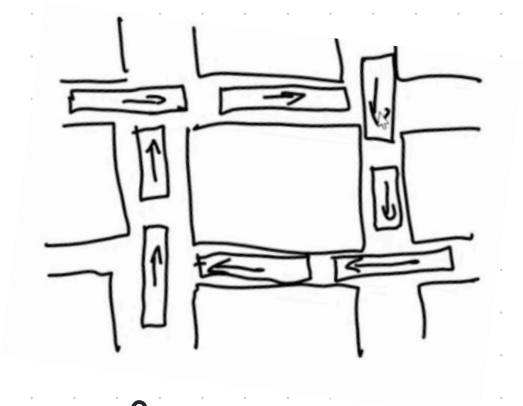
Lecture 12

Process Management

→ process states



Deadlock

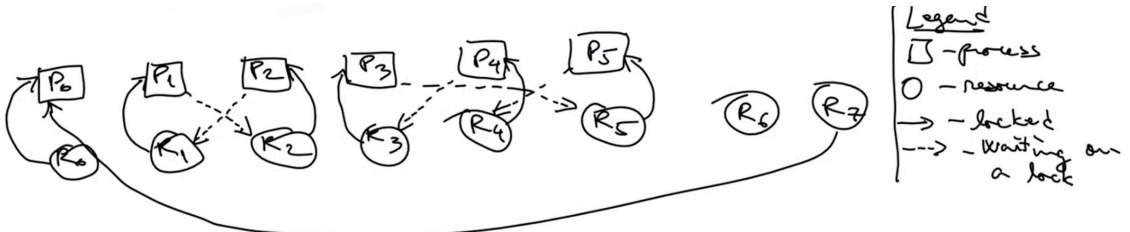


Only solution is to choose a victim to back up

like:	A	B
	lock(x)	lock(y)
	lock(y)	lock(x)

the invisibility creates the deadlock

* getting out of a deadlock?
  - stop one of the processes / threads
  - maybe (if you have the tech) <u>rollback</u> to a saved safe state
* detect deadlock



deadlock 1 | deadlock 2

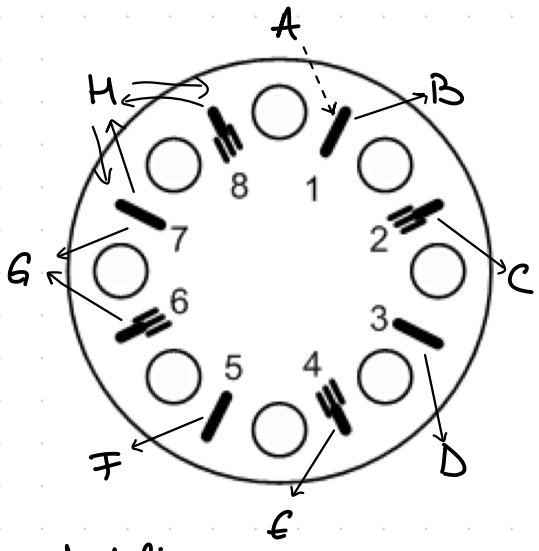
! cycles
if you detect a cycle, that's a deadlock

* preventing a deadlock
what makes it possible?

1) Mutual exclusion (mutexes are how to stay)
2) Hold (lock) and wait (not useable)
3) Non-preemption (someone else won't be able to steal the lock)
4) Circular wait (break by locking in the SAME order)

this is how you prevent a deadlock

Philosopher's problem



to eat you need both sticks
if everyone picks the one on the right no one eats
⇒ deadlock

! impose to choose the same order of sticks all the time
ex: pick up the stick with the lower number

Process scheduling

- FCFS – first come first serve
- SJF – shortest job first
  → client will have to provide an estimate of the execution duration
  * causes starvation, the guy with more time will never get to the cpu
- use priorities * mars robot story when priorities fail
- deadline scheduling
- Round Robin – assign small quantas of time to each process

Memory management

1) Allocation policies
2) Replacement policies
3) Placement policies

- Real allocation
  - single tasking systems
  - multi tasking systems
    - fixed partitions
      - absolute
      - relocatable
    - dynamic partitions