READ & WRITE

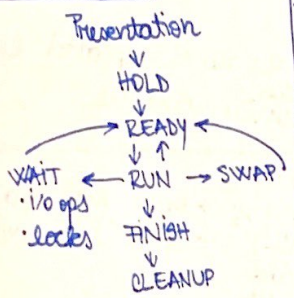
nb = nb. of bytes we must transfer
n = left space
enough space ⇒ returns nb
not enough space ⇒ transfer n bytes & returns n

DEADLOCK

- preventing a deadlock: figure out what makes it possible and deny one of that condition.
- what makes it possible: 1. Mutual exclusion → stay
  2. Hold (lock) and wait
  3. Non-preemption
  4. Circular wait → break by *

* locking resources always in the same order (impose a rule) (pay attention to how you lock)

PROCESS STATES:



PROCESS SCHEDULING:

FCFS = first come first served
SJF = shortest job first
→ client has to estimate the execution duration
Round Robin = assign quantas of time to each process in a round-robin fashion (start → first, end → last

ALLOCATION: ① Single tasking

② Multitasking – fixed partition – absolute
③ Multitasking – fixed partition – relocatable
BA = Binary Address → offset
PA = Physical Address → partition start + BA
④ Multitasking – dynamic partition
fragmentation ⇒ we don't have enough contiguous memory more complex
⑤ Paged allocation ⇒ address calculation
– we need a "page table" stating the physical address of each page
– BA → virtual address (virtual page, offset)
⑥ Segmented allocation
– fragmentation
– meant to provide memory access protection
– address calculation similar to paged
⑦ Paged – segmented: segments split into pages
– segment table & page table
– virtual address: segment, page, offset

LOADING POLICIES: What and when should we load into memory when a program starts?

① load all (pages) from start: • disadv: slow start, loading pages that might not be used
• adv: once loaded → is fast
② load every page when needed: • disadv: slower execution
• adv: fast start & no unused pages
③ Neighboring principle: if a process requests a page, it is likely to soon request its neighboring pages.

REPLACEMENT POLICIES: When memory is full, which pages should we kick to swap?

① NRU = Not Recently Used
→ every page has 2 bits that are periodically reset to 00
0: 00 → no recent read or write
1: 01 → recent write
2: 10 → recent read
3: 11 → recent read & write
→ for choosing a victim we take a page from the smallest class available ② FIFO
③ LRU = Least Recently Used
→ considering the system has N physical pages, maintain N×N matrix of bits as follows: whenever page k is accessed fill row k with 1 and then column k with 0.
ex: N = 3

000		000
000	1 →	101
000		000

& choose as victim the page with minimum row sum

PLACEMENT POLICIES:

① First Fit → choose the first spaces large enough (fast)
② Best Fit → choose the tightest fitting space (slow, fine grained fragmentation)
③ Worst Fit → allocate from the largest chung available
④ Buddy Fit → keeps lists for each power of 2, of free spaces
→ allocate the closest power of 2 to the request
→ split the remaining spaces across the other lists

$2^n = 1+1+2+2^2+\dots+2^{n-2}+2^{n-1}$

How do we track the occupied / free memory in the heap?
• 2 linked lists: one for free spaces & one for occupied.

⇒ fragmentation

i-NODE:

– 10 direct pointers to data block
– 11 → simple indirectation: N blocks of data
– 12 → double indirectation: N^2
– 13 → triple indirectation: N^3
– address size = A
– block size = N*A = B
N – nb of data blocks
– largest file that can be stored:
$10 * N*A + N*N*A + N^2*N*A + N^3*N*A$

CACHE: small & fast size C pag

RAM: large & slow size R pag.
R >> C

① Direct cache: place RAM pg. k in cache at k mod c
problem: cache collisions ⇒ * trashing
② Set cache: place in first free location
③ Set-associative: x sets of y pages, choose the set with k mod x & search for the free spot

EXEC: the exec system calls re-use the current process to run the other program. Essentially they wipe out the current process code, and replace it with the code of the new program. If we create a child and run the exec in it, we can keep our process.

POPEN: run a program or any Shell command from C code and get back its standard output or send data to its standard input. Use at final: ~~fd~~ pclose(..);

0 - handle reading from console
1 - handle writing to the console
DUP & DUP2: int x = dup(1) → makes copy of handle at index 1
dup2(oldfd, newfd) → overwrites newfd with handler at oldfd
undo dup2: use dup

? = 0 sau 0 dată
+ = 1 sau mai multe ori
* = 0 sau mai multe ori
^ = început de linie
$ = sfârșit de linie
\< = început de cuvânt
\> = sfârșit de cuvânt

[] - grupăm caractere
() - grupăm expresii
. - orice caracter

GREP: -E -i -c -v
(-i: ignore case; -c: line count; -v: invert match)

SED: -E "s/ce/cu ce/g"
(s: substitute; g: global)
"y/ce/cu ce" → înlocuiește pe rând poziția

AWK: NR - nb. of current line
NF - nb. of fields
$0 - entire input line, $1 - field 1
awk '...' → între apostrof
-F: => : field separator
$NF ~ /ABC$/ => NF matches ABC (line ending with ABC)
{print $1} => prints first field

$# => nb of arguments
$@ => all arguments grouped
$0 => the command
$1 => first argument
$? => exit code of last command

SHELL:
FIND: -name "*.txt" → caută fișierul cu numele dat
-type f => fișier | -type d => directory

WC: -m => character count; -l => new lines;
-L => max line count; -w => word counts

SORT: -u => removes duplicates; -r => reverse;
-n => numeric sort;

UNIQ: -c => at beginning of line prints nb of repeated lines;
-d => only repeated; -u => only unique

FIFO: mkfifo(name, 0600);
fd_read = open(name, O_RDONLY); → open for read
O_WRONLY → open for write
close(fd_read); → close the file
unlink(name); → delete the fifo
read(fd_read, &where, sizeof(where))
write(fd_write, &what, sizeof(what))

PIPE: int p[2]; pipe(p);
p[0] → for reading; p[1] → for writing

TEST: ① string: -n → length nonzero
-z → length zero
② int: -eq, ...