CURS 1:

1. Shell interpreteaza comenzile scrise in linia de comanda. O comanda este un program. Prima comanda e input pentru a doua (cu pipe). Cu valgrind verific erorile de memorie.

CURS 2:

1. 0-standard input, 1-standard output, 2-standard error (daca am scrie erorile la output, urmatorul program nu ar putea diferentia datele de erori – e nevoie sa le separam).
2. Pipe redirecteaza iesirea standard.
3. Citire cu string-uri si cu array-uri de caractere:

O imagine care conține hartă

Descriere generată automat

1. < intrare standard; > iesire standard; >> adaug la fisier, nu suprascriu;
2. Daca redirectez o eroare intr-un fisier (teoretic alaturi de output), nu o pune acolo. 2> inseamna ca pune eroarea standard intr-un fisier;

>a.txt 2>&1 inseamna sa redirecteze eroare unde redirecteaza iesirea standard;

/dev/null -> cand vreau sa scap de output si erori redirectez aici, ele sunt strese automat.

1. Valoarea de adevar in shell este codul de exit al programului: 0 este adevarat si orice altceva e fals;
2. Pt. stringuri avem operatori ca si in assembly: -eq, -lt, …
3. -e nume\_dir verifica daca exista fisier
4. ` face atribuire la variabile; intre “ ” se substituie $N (variabilele) cu valoarea lor; la awk pun ‘ ‘
5. Permisiuni: primul camp vine de la ce e (d=director,f=fisier) apoi 9 caractere care descriu permisiuni: read write execute (utilizator, grupul owner, restul lumii); read=2, write=3, executable=4.
6. all args: $@ nr args: $# cu shift trec la urmatorul argument ce e NF atunci??
7. -q il face quiet pe grep, sa nu mai afiseaza ce gaseste, doar sa spuna daca gaseste.

CURS 3:

1. | are prioritate mai mare decat operatorii logici
2. tr ‘[A-Z]’ ‘[a-z]’ transforma literele mari in litere mici
3. grep -v inseamna sa nu le arate; .? inseamna ca . poate lipsi
4. ps -o etime nume\_proces da timpul de cand ruleaza
5. daca pun -x dupa /bin/bash face debug
6. O imagine care conține text

   Descriere generată automat
7. O imagine care conține text

   Descriere generată automat

CURS 4:

1. Un process este orice program in executie
2. & pune un process in background
3. O imagine care conține text

   Descriere generată automat

CURS 5:

1. signal( nume\_semnal, functie\_ce\_sa\_faca) pentru a suprascrie cum reactioneaza procesul la primirea unui semnal
2. signal(SIG\_CHLD,SIG\_IGN) opreste crearea de procese zombie(nu il intereseaza codul de exit)
3. un exec intr-un process inlocuieste tot codul cu cod-ul exec-ului
4. 0 e pentru citire, 1 pt scriere in pipe

CURS 7:

dup2 doar copiaza de udeva altundeva handler-ul. Ca sa faci un undo la dup2, prima data il dublici cu dup si apoi suprascrii cu dup2 aceasta dublicatie

exec nu elimina tabelul asta de handle-uri

O imagine care conține text

Descriere generată automat

1. Grep, awk,… comunica prin pipe prin simplul fapt ca fac printf si scanf si bash-ul le modifica fisierele de scriere si citire
2. Nu inchidem noi pipe-urile pt. exec cand lucram asa ps|grep|awk pentru ca exec oricum sterge tot codul dupa ce se executa, pipe-ul va fi inchis cand se va termina ps/grep/awk. Programul interesant:

O imagine care conține text

Descriere generată automat

1. Ipcmk -M 1024 -p 600

Ipcs ne arata ce ipc-uri sunt partajate

CURS 8:

1. Memorie: context, code, constante, variabile,stiva
2. Stiva: de cate ori chemi o functie, se creaza o intrare in functie, asa stim unde sa ne intoarcem cand se termina functia (variabilele si stiva se dublica ;a fork)
3. Thread-ul: doar ia functia si o ruleaza deodata cu restul ( are nevoie sa aiba propria stiva). Prin acele atribute la care noi acum pasam null se poate atribui un anumit stack pe care il vrem unui thread
4. Scrierea in pipe e mai rapida decat scrierea in consola. Program simplu threads:

O imagine care conține text

Descriere generată automat

1. Variabile conditionale: asteapta pana cand ti se spune, dar nu fa nimic decat daca..

CURS 12:

1. FAZA DE PREZENTARE->HOLD(pana cand am resurse necesare)-> READY->RUN(cand am procesor)->FINISH

->ELIBERARE DE RESURSE

Procesele penduleaza intre run si ready (pentru ca sunt mai multe procese ce ruleaza si vor procesor)

Mutam intr-o stare de wait: il dam jos de pe processor ca sa lasam celelalte procese sa isi faca treaba pana cand primim datele de care avem nevoie (pana cand le citeste hard-ul sau vin in pipe/consola). Cand le primim se muta iar in starea de ready. Asta se intampla pt ca discul e foarte lent si nu are sens sa tinem procesorul ocupat degeaba. (RUN->WAIT->READY)

RUN->SWAP daca nu este destul spatiu pt a rula un program, mutam unele pagini dintr-un process in swap. Dezavantaj, acolo se misca foarte lent. (RUN->SWAP->READY)

Planificarea proceselor:

-primul venit primul servit

-shorter job first (clientul trb sa dea o estimare a duratei programului): risc de starvation pentru task-uri mari

-prioritati (robot Marte)

-deadline scheduling (task – durata - termen)

-round Robin: se aloca fiecarui process cate o cuanta de timp processor (prioritatile ar primi mai multe cuantumuri)

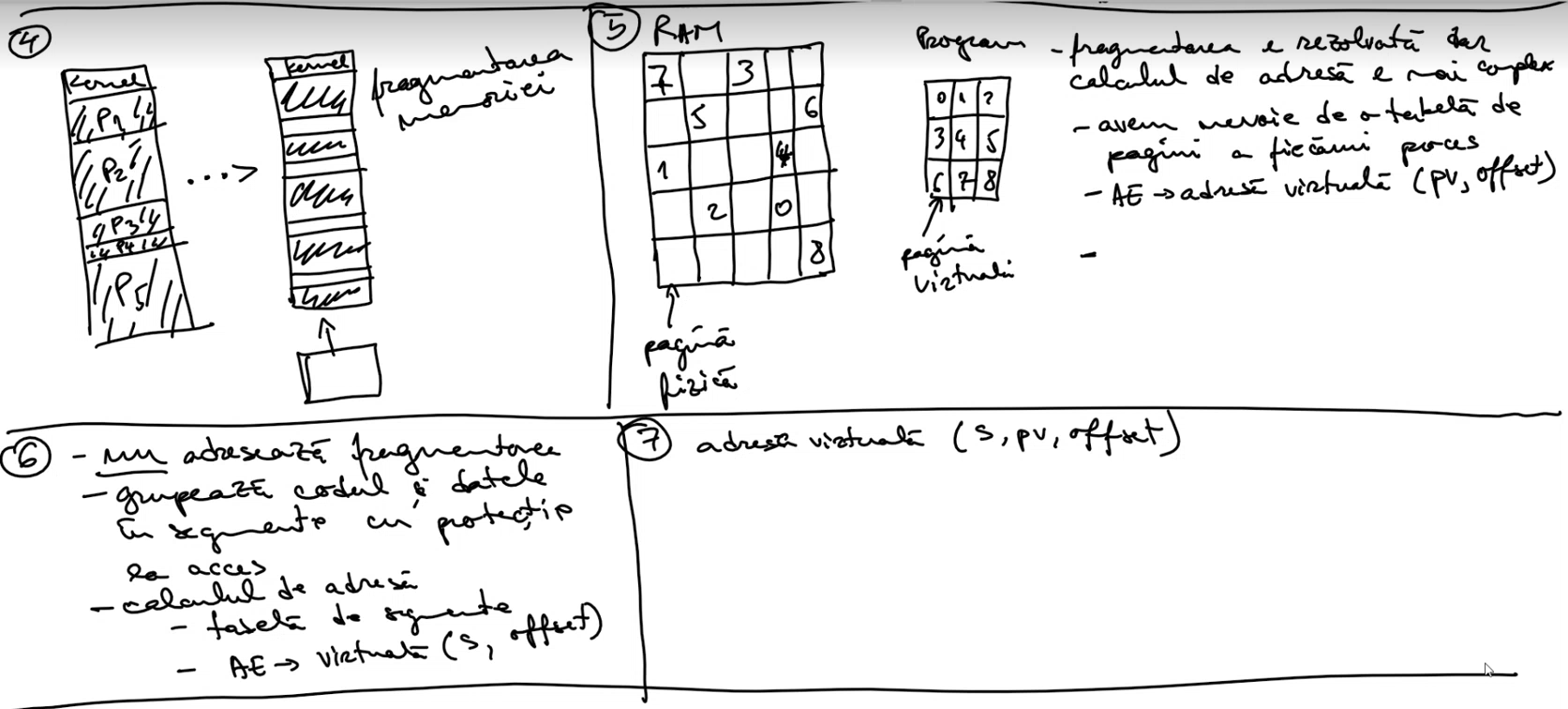
-gestiunea memoriei:

O imagine care conține text

Descriere generată automat

O imagine care conține text

Descriere generată automat



POLITICI DE INCARCARE: ce si cand incarcam in RAM la pornirea unui process? (pagini, dar cand?)

1. incarcam toate paginile de la bun inceput (dezavantaje: pornire lenta, memorie ocupata de pagini care nu se vor folosi; avantaje: odata incarcat, merge repede)
2. incarcam fiecare pagina cand devine necesar (dezavantaje: rulare mai lenta, avantaje: pornire rapida, paginile nefolosite nu ajung in memorie)
3. principiul vecinatatii: daca un process refera o pagina, e probabil sa refere curand paginile invecinate -> daca ni se cere o pagina aducem in memorie vecinatatea

POLITICI DE INLOCUIRE(cum alegem o pagina sa fie mutata in swap cand e plina memoria?)

1. FIFO – nu tare bine

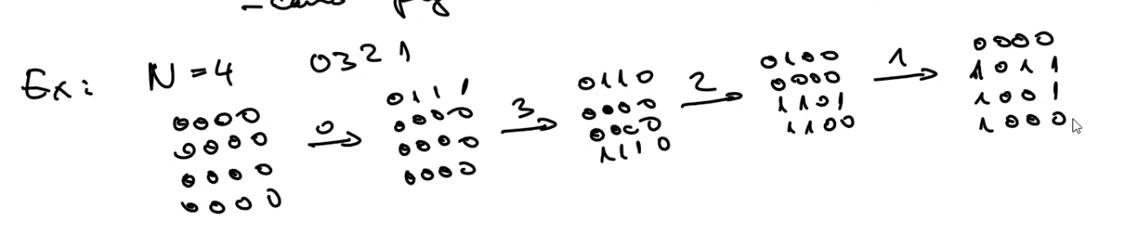
2. NRU-not recently used: fiecare pagina fizica are 2 biti care periodic sunt resetati la 0 0;

O imagine care conține text

Descriere generată automat

Asa cum sunt as scoate. Alegem o victima din cea mai mica clasa nevida.

1. LRU-least recently used: considerand ca avem n pagini in RAM, intretinem o matricem de nxn biti astfel: cand pagina k este accesata, populam cu 1 linia k si apoi cu 0 coloana k

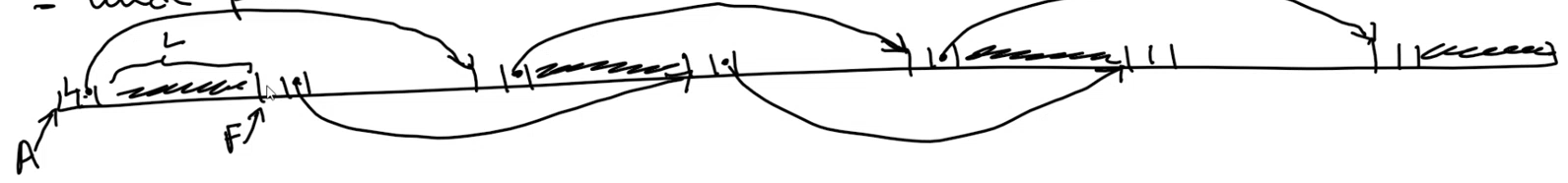


Alegem ca victima pagina cu cea mai mica suma a liniei

POLITICI DE PLASARE:

(cum tinem contabilitatea spatiilor allocate/libere in heap? Unde plasam o cerere de alocare(malloc)?)

cu 2 liste inlantuite pentru spatial alocat si cel free, cautam prin aceste liste unde sa plasez malloc-uri (ne confruntam iar cu fragmentare)



Cum putem evita fragmentarea?

-first fit: plasam in prima poz libera destul de mare(paraleli cu fragmentarea-dar e rapida)

-best fit: alegem cel mai mic spatiu suficient de mare(mai lenta, fragmentare cu spatii foarte mici)

-worst fit: alocam in cel mai mare spatiu disponibil(e mai ok fragmentarea)

-buddy fit: alocam cea mai apropiata putere a lui 2; intretinem liste ale spatiilor goale dupa dimensiuni

O imagine care conține text

Descriere generată automat

CACHE-URI:

O imagine care conține text

Descriere generată automat

SISTEME DE FISIERE:

O imagine care conține text

Descriere generată automat

38. CE FACI CAND AI UN DEADLOCK?

Iesire din deadlock-> detectare de deadlock->prevenire

Iesire:

-oprim procesul

-alege un prces(thread) ca “victima” si opreste-l

-daca am avea posibilitatea de a salva uns avepoint am putea cere unui process sa revina la starea anterioara fara sa-l oprim (risc de live-lock: incerci si ajungi tot acolo)

Detectare:

-un ciclu in graful de alocare a resurselor e un deadlock

Prevenire:

Ce il face posibil?

-mutual exclusion (lock/unlock) O PASTRAM

-hold(lock) and wait O PASTRAM

-non-preemtion (un nou lock nu anuleaza cel anterior) O PASTRAM

-asteptare circulara: o dazactivam prin **blocarea resurselor in aceeasi ordine**