SPM. IV — Subiecte posibile:

Cap. I : taxonomiile (sigur)

Cap. II : ± TOT

1) ~~Planif. ppl.~~, Org. ppl., perf., etc.

2) Hazard → ce sunt, care sunt & cum se rezolvă

3) Planificarea ppl. → probabil (99%) ppl. static

↓

Pb.: se dă diagr. de stări sau tbla. de rezervări ⇒ se cere calc. latenței medii ⇒ tb. să se facă cu rețele de [redacted] coliziune, calcule, etc. (NU ochiometric!!)

Cap. III: 1) diametru, etc. --- ce repr. rețeaua

2) explicarea mec. de routare

3) filosofia de comutare

4) modul de construcție → Benes! Omega! Multistage cube!

5) evaluarea unui polinom prin shuffle-exchg

Cap. 6 : Alg. de calcul ||

Alg. < secvențiali

|| ⇒ bucăți din alg. lucrează în ||

Cum evaluăm un ACII ? Seamănă cu ppl.

Evaluarea se face după timpul de execuție.

↓

crit. capital (cum facut alg. || pt. a minimiza timpul de exe.)

2 soluții de realiz. a alg. ||:

- se ia alg. secvențial & se det. ||-ismul inerent al său (port. de instr. indep. care pot fi exe. în ||, de. ∃ resurse) ⇒ sol. simplă, dar puțin perf.;
- rescrierea totală a alg. ⇒ regândim modul de rez. a pb. aî. să se exe. || → ∄ refețe.

Pb. ce afectează alg. ||:

→ comunicația între firele de execuție || ⇒ scade perf. câștigate prin paralelizare datorită schimbului intermediar de info între firele de exe.;

∃ 2 met. de comunicare: partajarea memoriei (multi/μP)
mesaje (multicalculator)

∃ multe aplicații de calcul ∥ implem. pe sist. multi/μP care comunică tot prin mesaje, ptr că e mai simplu (partajarea mem. pp. pb. de tipul sincronizare, deadlock, blocare, op. atomice, etc.).

Comunicarea prin msje. în sist. multi/μP e eficientă numai dc. sunt op. simple → timpul de comunicare nu trb. să depășească timpul de calcul efectiv ⇒ dc. am calcul matricial nu e eficient să fol. msje.

I. Cazul programare ∥ pe sist. multi/μP (partajare memorie):

De ce avem nevoie? 1. instrum. ptr a crea procesele
2. instrumente de sincronizare ⇒ excludere mutuală

dc ex. trb. să adunăm 2 vectori ⇒ am 2 procese, A ≥ B ⇒ alg. l-am gândit astfel: fiec. proces citește o var. index, preia din vector elem. index, le adună, apoi incrementează index ⇒ ambele procese vor face la un moment dat index = = index++ ⇒ trb. să am un control f. bun al op., ptr că dc. între citirea și scrierea lui index de către A se intercalează și B, în loc ca index la sf. celor 2 op. să crească cu 2, va crește cu 1 ⇒ trb. să am mec. de excludere mutuală (când un μP lucrează să nu poată interveni nimeni).

1. Crearea de procese:

• Ptr. a crea un proces din alt proces putem apela fețile S.O. ⇒ posibil și sub Windows și sub Unix.
↳ are o instr. fork ⇒ dc. într-un proces exe. instr. fork acel proces (atât cod cât și zonă de dte.) se dublează în memorie

fork: return-code = fork();
if (return-code = 0) { cod slave (copil) } → se poate ca crearea codului copil să dea eroare.

else { cod părinte }

Instr. fork e utilă ptr. că: → { cod compact (un sg. executabil ptr. alg. de calcul ||)
→ pot să fac o sg. iniț/alizare atât ptr. codul părinte cât și ptr. copil (copilul copiază iniț. din codul părinte).
⇓
modif. se fac numai în zona de mem. a procesului care a modif.

multithread ≠ multiproces

↓ aceleași sp. program
↓ la duplicarea multifirului de exe., mem. e comună, deci nu se dublică
⇓ e f. simplu să implem. alg. de calcul ||, ptr. că nu mai trb. să implem. mec. de comunicare între fire
⇓ pb.: aplic multithreading sunt optimizate ptr. sist. monoUP, ptr. că de ex. vrem să implem. multithreading pe sist. multiUP apare pb. enormă a sp. comun de memorie a procesului părinte
⇒ Se bat cap în cap pe zona aia de mem., pe resursa aia de calcul (un fel de multitasking)
↳ optimizarea max. e hyperthreading, dar tot ptr. monoUP este

↓ fiecare proces are propriul sp. program
⇓ la implem. trb. să implem. și mec. de comunicare între procese
↓ mec. foarte bine puse la pct. ⇒ nu apar pb. la implem. pe sist. multi/UP
↓ La LAB!!! : urm. 2 teme (485) trb. implementate multiproces !

Un proces poate conține m. multe fire de execuție (threads).

2. Excluderea mutuală:

Dacă avem posib. partajării procesel, tot trb. să rez. pb. excluderii mutuale.

∃ m. multe clase:

→ cuplul LOCK/UNLOCK:

ex: lock
{ while (L==1) NOP;
L=1; }

unlock
{ L=0; }

⇒ L e folosit pe post de semafor ⇒ cât timp L==0, pot să fac lock; când L=

Lock (L)
index = index + 1;
Unlock (L)

⇒ prin intermediul lui L am acces la var. index

→ s.n. secțiune critică

Lock/Unlock rez. pb. de citire/modificare, nelăsând alt proces să intervină.

Pb.! fiecărei sect. critice trb. să i se aloc. un semafor ⇒ dc. sect. critică e una foarte mare, e ok, dar dc. avem multe sect. critice simple (1 sg. instr.) e greoi să îi alocăm fiecărei sect. câte un semafor.

→ cuplul TEST & SET: op. atomică ⇒ nu poate fi întreruptă
⇒ cât timp μP execută TAS (pp. citire și scriere), zona resp. de mem. e blocată (hw.)

Instr. TAS: → Motorola: Xmem
→ Intel: Xchg
} schimbă 2 locații de mem. între ele.
↳ #Lock: blochează magistrala până se termină exec. instr. TAS

→ cuplul WAIT/SIGNAL: Un fel de P și V de la semafoare (& SMP!!!)

Rolul ≡ WAIT & SIGNAL e de a realiza semaforizarea, dar nu la nivel binar, ci permit unui anumit nr. de procese să intre în secțiunea critică.

ex.: ≡ Wait

```
{while(S<=0) NOP;
 S=S-1; }
```

Signal

```
{ S=S+1; }
```

⇒ Semaforul în momentul în care se preiau acele semnale, și semaforul.

→ cuplul FETCH & ADD: e altă var. de TAS, dar poate fi utilizată în cadrul adunării unor vectori, lucrând direct cu elem vectorului respectiv.

→ Barrier: îmi spune când câte procese au ajuns într-un pct. → le ține pe loc pe unele până ajung toate în pctul respectiv, apoi pot să treacă la pctul următor;

Deadlock: poate apărea (la excluderea mutuală) între diversele mecanisme

ex: la A: Lock(L1)
t1 B: Lock(L2)
t2 A: Lock(L2)
t4 B: Lock(L1)

⇓

blocare ciclică (definitivă & permanentă) a 2 procese

Astfel de situații apar datorită programării proaste. ∄ instrumente de evitare/rezolvare a acestor situații ⇒ la noi să gândim codul așa încât să nu se ajungă în deadlock.

II. Cazul programare || în sist. multicomputer (comunicare prin msje.):

Singurul lucru care ne-a prevăzut e alocarea instrum. care să permită alocarea bufferelor și comunicarea între ele (msje).

↓

protocol de rutare, protocol de comunicație, etc.

//

1 pp.: gestionarea & crearea procesel

lock: procesul nou creat s.n. copil sau slave, pt. că instr. lock returnează părintelui ID-ul procesului nou creat ⇒ procesul părinte poate gestiona procesul copil (procesul copil nu deține info. despre procesul care l-a creat).

Pb. mare la multiprocesi: ne-a să avem grijă să ∄ accese concurente nesemaforizate, iar semafoare ne-a să ne asigurăm că ∄ deadlock.

2 ne-a să realizez un mec. eficient & fiabil de comunicare (prin msje.) între µP.

//

Comunic. prin msje. la multi/µP e eficientă doar dc. am volum mic de mesaje ce se transmit (cant. mică de info. ce se transmit)

↓

pt. că timpul de creare & gestionare a msj. e mult mai mare decât timpul necesar accesului la o locație partajată de memorie

↓

mereu ne-a comparat timpul de comunicație (µP stă degeaba) cu timpul de execuție efectiv.

∄ alg. paralelizabili 100% ⇒ timp secv. + timp || + timp de comunicare.

Dc. câștigul pe care îl avem în partea de || nu justifică timpul de comunicare, nu are rost să paralelizăm (dublăm resursele de calcul, dar dc. timpul de comunicare e > decât timpul de calcul, atunci aceste resurse vor sta degeaba mai mult decât vor lucra ⇒ nu se justifică costurile).