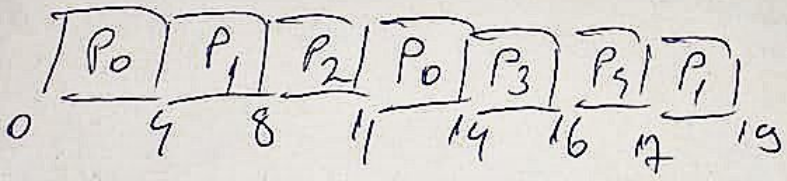
4.5)

Process	t	CPU
P_0	0	3
P_1	1	6
P_2	4	2
P_3	5	2
P_4	7	1

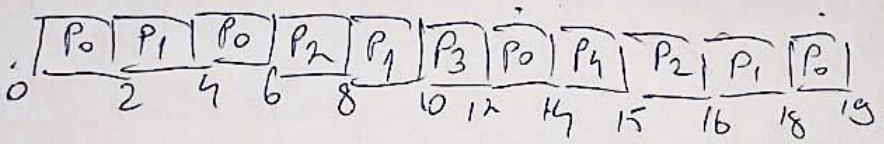
Queue:
P_0
P_1
P_2
P_0
P_3
P_4
P_1



4a)

Process	t	CPU
P_0	0	7 5 3 1 0
P_1	1	6 4 2 0
P_2	4	3 1 0
P_3	5	2 0
P_4	7	4 0

Ready queue:
~~P_0~~ P_0
~~P_1~~
~~P_0~~
~~P_2~~
~~P_1~~
~~P_3~~
~~P_0~~
~~P_4~~
~~P_2~~
P_1



$w(P_0) = (4-2) + (12-6) + (18-14) = 2+6+4 = 12$

$w(P_1) = (2-0) + (8-4) + (16-10) = 2+4+6 = 12$

$w(P_2) = (6-4) + (15-8) = 2+7 = 9$

$w(P_3) = (10-5) = 5$

$w(P_4) = (14-7) = 7$

$avg = (12+12+9+5+7)/5 = 8.8$

1. a) fork();
pthread_t *thr = malloc(sizeof(pthread_t) * 10);
for (int i = 0; i < 10; i++)
pthread_create(&thr[i], NULL, rout);

Procesul părinte va da fork și va crea un proces copil. Acestea vor rula în mod concurent (cu zone de memorie separate). Apoi fiecare proces va crea câte 10 threaduri care vor executa o rutină. Threadurile au acces la aceeași zonă de memorie și rulează în mod paralel. (task parallelism)

b) Dacă fork() eșuează și nu se creează procesul copil, părintele va continua cu restul instrucțiunilor și astfel nu mai avem concurență.

2.
```
p = fork(); //2
if(p == 0) {
    // proc 2
    r = fork(); //4
    if(r != 0) {
        // proc 2
        fork(); //5
    }
} else {
```

```
// proc 1.
q = fork(); //3
if(q == 0) {
    // proc 3
    s = fork(); //6
    if(s != 0) {
        // proc 3
        fork(); //
    }
} }
```

3.a) Lacătul este inițializat cu 0, deci resursa critică este liberă. Procesele împart lacătul și așteaptă ca acesta să fie liber. ~~Deci, rezultă~~ Când lacătul are altă valoare față de 0, înseamnă că un proces a luat lock și manipulează resursa critică, dar celălalt așteaptă ca primul proces să termine și să dea unlock. De aici rezultă excluderea mutuală. Dacă procesele au o rotimă care garantează sfârșit de execuție, atunci rezultă și progres și timp limit de așteptare (Pi preia resursa, dă lock, execută, dă unlock și așteaptă; Pj așteaptă, preia resursa, dă lock, execută și dă unlock).

5) Inițial lock este 0. ~~Deci~~ Să zicem că Pi este primul proces care apelează compare_and_swap. Astfel lock devine 1, iar dacă funcție se returnează 0. (vezi snippet-ul de cod de la pg 210 a cărții de curs). Astfel se iese din while și Pi accesează zona critică.

```
int compare_and_swap (int *value, int expected, int ne
{
    int temp = *value;
    if (*value == expected) *value = new_value;
return temp;}
```

Va fi apelat astfel:

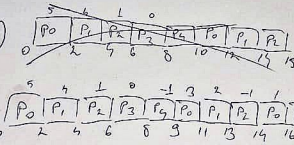
compare_and_swap (&lock, 0, 1);

Lock este 0, deci temp este 0.

Funcția va returna temp, adică 0, deci se iese din while.

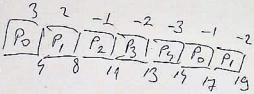
După zona critică, lock-ul se restaurează la 0, și pleacă Pj zona critică.

4a)



?

5)



c) $avg = (w(P_0) + w(P_1) + w(P_2) + w(P_3) + w(P_4))/5 = (12+12+11+6+8)/5$

$= (49)/5$

$= 9.8$

$w(P_0) = 2+2+2+1+2+1+2 = 12$

$w(P_1) = 2+2+2+1+2+1+2 = 12$

$w(P_2) = 2+2+2+1+2+2 = 11$

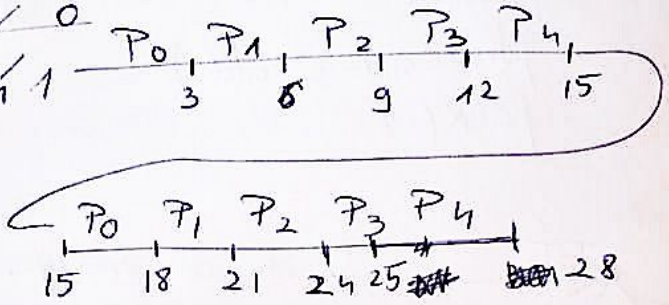
$w(P_3) = 2+2+2 = 6$

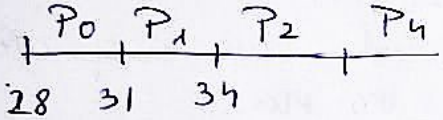
$w(P_4) = 2+2+2+2 = 8$

Alg Robby

②

Proces	CPU
P_0	9 6 3 RR $q = 3$
P_1	13 10 7
P_2	8 5 2
P_3	4 1 0
P_4	7 4 1





Framewri

Alg. LRU

	1	2	3	4	5	6	1	2	3	4	5
3 frame-wri	1	2	3	4	5	6	1	2	3	4	5
	4	5	3	4	5	6	1	2	3	4	5
	4	5	3	4	5	6	1	2	3	4	5

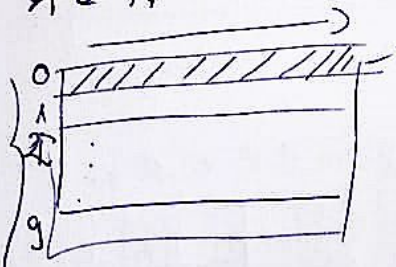
	1	2	3	1	4	2	5	3	6	4	4	5	4	6
	1	2	3	1	4	2	5	3	6	4	4	5	4	6
	1	1	1		1	1	5	5	5	4	4			
		2	3		4	4	4	3	3	3	5			
			2		3	3	2	2	6	6	6			

Subiect 2:

Problema filozofilor

```
do {
    wait( chopstick[i] )
    wait( chopstick[(i+1) % n] )
    /* — */
    signal( chopstick[i] )
    signal( chopstick[(i+1) % n] )
```

$A \subset \mathbb{N}^{10 \times 10}$



→ se află continuu în memorie nu împărțită

Pagina ține 10 întregi ⇒ ⇒ 10 pagini

$P_0, P_1, \dots, P_9, P_{10}$ = A, P_1

```
P1: for (i=0; i<10; i++)
        for (j=0; j<10; j++)
            A[i][j]=0;
```

b)

c) cum arată diagrama P_0 în mem.

$\underbrace{A[0][0] \dots A[0][9]\ (i=0)}_{P_0}\ \underbrace{A[1][0] \dots A[1][9]}_{P_1}$

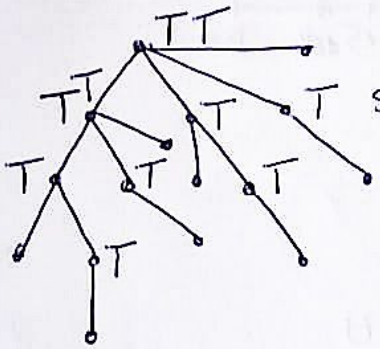
EXAMEN SO

5,6 probleme — 50p minim 25p 2 ore

Subiect 1:

```
for (i=0, i<2, i++)
    fork()
    pthread_create()
    fork()
```

a) câte sunt create?
b) Desenați arborescența

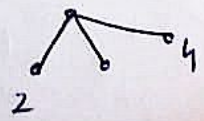


La nr. de pași în care s-au făcut ptr. fork se copiază și thread-urile?
↓ nu este un răsp. corect.
ori da ori nu

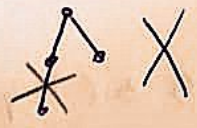
fork
fork
pthr
fork fork
pthr
fork

4 fork => 2^4 procese

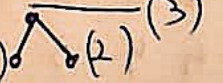
fork plec de la 1 proces → 2 procese
fork 2 p → 4 pr.




```
fork()
fork()
```



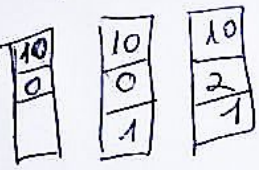
```
pid = fork()(1)
if(pid != 0) {
    pid = fork()(2)
        if(pid)
            fork()(3)
```



pagini:
A 0 1 2 3 4 5 6 7 8 9

100

prog1



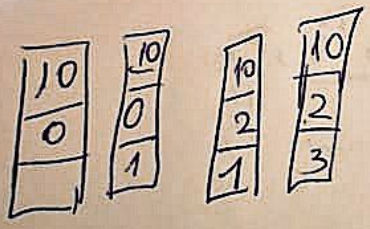
10 ori

10,0, 10,0 .... ,10,0 10,1 ....

```
for(j=0; j<10; j++)
    for (i=0; i<10; i++)
        A[i][j]=0;
```

=> 10,0, 10,1, 10,2, 10,3 10,9

A[0][0], A[1][0], A[9][0]



0 0 0 0 0 0 0 0 0 0

de 10 ori pagina

0

1 1 1 ... 1 1 2 2 .. 2 ... 9 9

10 ori A[2][0]