Problema celor 5 filozofi - generalizare pentru n filozofi

```
#define N 5
#define LEFT ((i+N-1) % N)
#define RIGHT ((i+1) % N)
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N]; // variabila globala; retine starea celor N filozofi
semaphore mutex = 1;
semaphore s[N];
void philosopher (int i)
    { while (TRUE)
        { think();
          take_forks(i);
          eat();
          put_forks(i);
        }
    }
void take_forks (int i)
    { down (mutex);
      state[i] = HUNGRY;
      test(i);
      up (mutex);
      down (s[i]);
    }
```

```
void put_forks(int i)
    { down(mutex);
      state[i] = THINKING;
      test[LEFT];
      test[RIGHT];
      up(mutex);
    }

void test(int i)
    { if ((state[i] == HUNGRY) && (state[LEFT] != EATING) &&
          (state[RIGHT] != EATING))
              up(s[i]);
    } // ridică semaforul pentru procesul i aflat.
```

- Problema cititori - scriitori

  -> apare de obicei când avem aplicații cu baze de date

- Problema frizerului somnoros

  -> frizer <-> tunde / doarme

  -> clienții care intră în prăvălie trebuie să aștepte dacă frizerul tunde.
  -> clienții trebuie să stea obligatoriu pe scaun (sunt n scaune)
  -> dacă un client intră și vede că frizerul tunde, iar toate scaunele sunt ocupate, atunci el pleacă

```
#define CHAIRS 5
typedef int semaphore;
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0; // numărul de clienți care sunt în așteptare.
void barber ()
    { while (TRUE)
        { down (customers);
          down (mutex);
          waiting --;
          up (barbers);
          up (mutex);
          cut_hair ();
        }
    }
void customer ()
    { down(mutex);
      if(waiting < CHAIR)
          { waiting ++;
            up (customers);
            up (mutex);
            down (barbers);
            get_haircut ();
          }
      else up(mutex);
    }
```

Planificare (scheduling)

Politici:

-> FIFO -> cea mai dezastruoasă metodă

-> bazată pe prioritatea procesului.

UNIX -> politică bazată pe priorități -> algoritmul ROUND ROBIN

- începe să execute procesele cu prioritate 0
- când le-a terminat, trece la procesele de prioritate 1.

. . . .

Prioritatea unui proces poate să fie dată de:

- utilizator, la lansare

program: $ myExe -> are o prioritate standard.

lansare: $ nice myExe -> are prioritate mai scăzută decât cea standard

$ nice -nr myExe -> scade prioritatea cu nr.

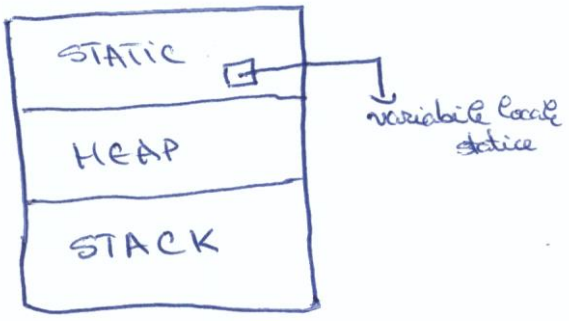
-> preemptivă -> bazată pe preemptări

- se acordă o prioritate în funcție de ultimele execuții (de obicei, ultimele n execuții)
- există riscul „de falsificare" a timpului.

ex: se rulează de 5 ori un program care durează ~2 ore
în una din cele 5 rulări au fost introduse greșit
argumentele => timp = ~ secunde

Securitatea și gestiunea memoriei.

Zona de date a unui proces



- STATIC -> zona este inițializată cu "0"; variabilele globale
- HEAP -> zona cu variabile memorate la compilare, dar sunt folosite la rulare; variabile dinamice.
- STACK -> „stiva"; variabilele locale ale lui „main";

- recapitulare alocare dinamică

```
void * malloc (int dim)
void * calloc ( int nr, int dim)   // este făcut pentru alocare de vectori
                                   // nr = nr. de elemente ; dim = dimensiunea.
                                   // dim este oferit de dorel cu SIZE OF
```

calloc (m, n) <=> malloc (m * n)

calloc -> inițializează zona cu 0.

```
void * realloc (void *p, int dim)   // realocarea unui „obiect" cu noua
                                    // dimensiune
                                    // returnează în mod sigur o zonă de
                                    // memorie liberă -> ori mărește spațiul
                                    // la același adresă, dacă are spațiu liber;
                                    // ori copiază totul la o nouă adresă
                                    // care oferă suficient spațiu și returnează
                                    // noua adresă
```

```
void free (void *p)   // eliberarea memoriei
```