

# Curs 7 (So)

Întrebări : "≠" dintre proces și thread

↓  
au memorii comună  
∈ unui proces

ex: - suma unui array mare → favorab. thread (dimens. mare)

- resize pt. mai multe imagini → proces

→ nu ne trebuie același conținut de memorie pt. fiecare fol.

→ crearea proceselor în batchuri

multithread vs multiprocess

in memoria unui proces → o poză

in cazul fol. thread. → sunt încărcate în memorie la o poză.



Un thread / proces să facă cât mai multă muncă

INDIVIDUALĂ

ASC

• 1 procesor → mai multe unități de calcul cu propriile ALU

↳ multi-core

in comun: L3 cache

FPU  
(floating.)  
cache  
regist.

logical processors (core-uri virtuale → thread) → eficient in cazul op. diferite

→ hyper-threading → resurse proprii = doar registrii

→ nu au ALU dublată  
avantaj doar dacă resursa este disponibilă

dezavantaj: concurență pt. resurse.



→ tehnă multithread → adăugată în 2004 & 2005 de Intel  
↓  
hyperthreading doar pe arhitectură x86

- Thread Context Block (TCB)
- Process Context Block (PCB)

## Procesarea paralelă

### Amdahl's Law

$$\text{speedup} \leq \frac{1}{s + \frac{(1-s)}{N}}$$

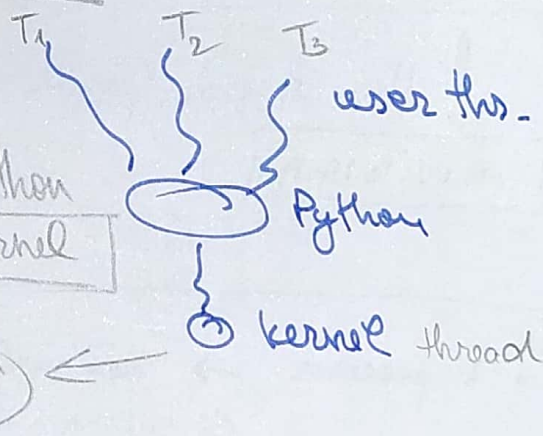
$s$  → serial portion  
 $N$  → processing cores

### Alocarea threadurilor

User threads , kernel threads.

- **Many to one** : python

(→) are există multithreading în python  
→ thr. utiliz. sunt mapate în kernel  
într-un singur thread.



alte ex: Solaris Green threads  
GNU Portable threads

mechanism: Python CIL

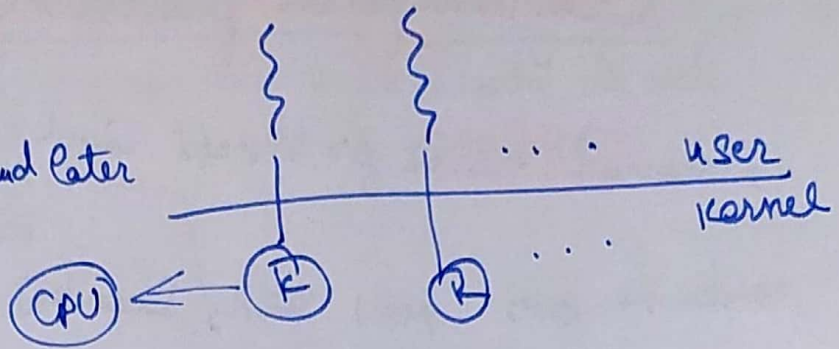
"global interpreter lock"

- are acces la un singur thread de kernel
- numpy e scris în C, C++ și assembly; operează în python

## • One - to - one

ex: Windows  
Linux, Solaris 9 and later

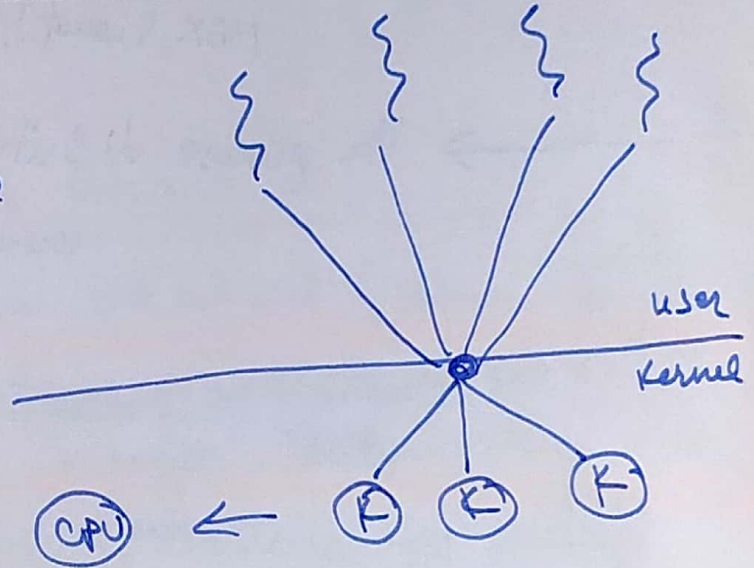
- în situații speciale  
paralelizare 100%



## • Many - to - many

ex: Windows: ThreadFiber package  
Solaris prior to 9

→ th. sunt alocate th. de  
kernel disponibile



lab.

Pthreads

POSIX standard API

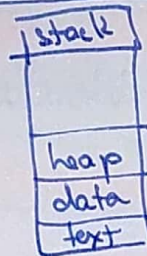
\* pthread\_create  $\Leftrightarrow$  fork  
- join  $\Leftrightarrow$  wait  
- exit(0)  $\Leftrightarrow$  exit()

UNĂ  
PRACTICĂ

- pornim mai multe thread. (le stocăm într-un array)  
→ așteptăm într-un ciclu repetitiv să se termine toate.

\* windows e puțin \* CreateThread  
WaitForSingleObject

și în java alt sistem



int sum

void \* runner, (void \* param)  
(funcția apelată de  
th.)

(sp. comun threadurilor)



# Sincronizarea proceselor

→ resursă hb. să fie blocată când e utiliz. → locked.

counter++  $\Leftrightarrow$  MOV RCX, [counter] ([0xdeadbeef])  
INC RCX  
MOV [counter], RCX

→ la procese diferite → RCX dif. → nu este probl. de concurență  
core-uri dif.  
↳ în paralel.

→ dar ~~se~~ scrierea în counter (aceasi poră de memorie) este problematică

!!!!  
→ este necesară fiec. proc. să facă munca individuală (vezi pg. 1)

→ de operația este identică → coreurile sunt diferite.  
→ alte unit. ALU

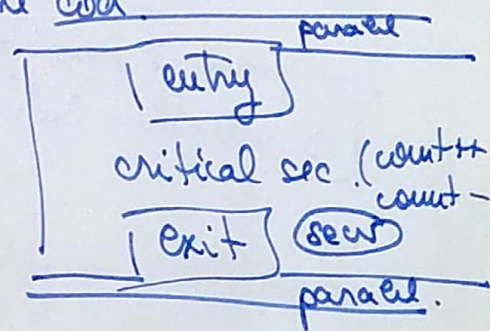
## Critical Section Problem

→ fiecare proces are o secțiune critică de cod.  
cu un entry și un exit

while (turn == j) (j) → banca de j  
→ C.S.C.

turn = j

→ remainder sec.





# Synchronization Hardware

locking pe resursă → op. din acea bucată de cod devin ATOMICE

acquire lock  
c.s.  
release lock  
remainder.s.

→ funcție atomică de test-and-set.

ca un switch :

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

ATOMICĂ

→ ret. vechea val.

→ set. noua val. ca fiind TRUE

→ cât timp test and set nu s-a terminat → resursa e lockată

⇒ while ( test\_and\_set (&lock))  
→ do nothing

/CS

lock = false

/remainder section

\* ca un semafor la thread \*

sau mutex

Mutex Locks ⇔ mutual exclusion.

→ var. binară

↓

1 resursă

mtx\_lock

mtx\_unlock

Semaphore

lock

unlock

wait(S)

signal(S)

mai multe resurse

când e solicit. semaforul scade ==

când un thr. termină în res → sem ++

nr. neg. → nu mai avem res. disp.



→ cât timp. așteaptă th. nu consumă  
 buzzer, var, instr. de același tip.  
 → Am 5 ALU :  $S = 5$   
                   ↓  
               resursă

wait(S)  
 signal(S)

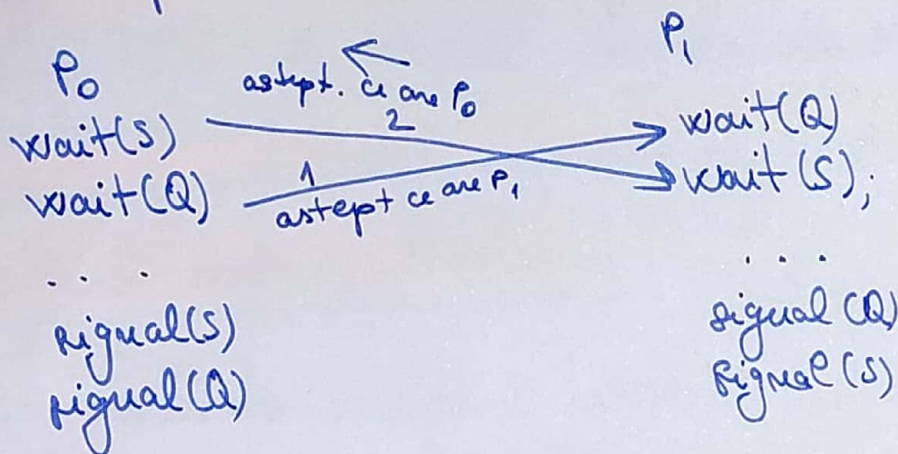
1 RESURSA = MUTEX (ex.: o sumă totală  
 tonă de măr)

> RESURSE = SEMAPHORE

(ex.: mai multe fișiere  
 de log)

Deadlock

2 procese așteaptă <sup>fișiere</sup> indefinit ~~așteaptă~~ după o resursă  
 eliberată după terminarea așteptării celui alt proces.



Starvation - Indefinite blocking \* (curse)

→ Priority Inversion : priority-inheritance protocol

astepțarea infinită a eliberării unor resurse

lock => bottleneck de performanță

computerphile

(lock suma array)



# Ans 8 (So)

## Deadlock and Starvation:

pr. context switch: stocarea datelor necesare unui proc. curent + comutarea către un exec. unui alt proces salvat

→ condițiile Goldilocks

thread context = toată informația de care un thread are nevoie pentru a-și relua activitatea de unde a rămas.

→ setul de regist. CPU + stackul

thr. switch mai ușor decât proc. c. switch ⇒

⇒ nu trebuie să faci switch al spațiului de ad. din memorie.  
↳ adr + cache + ...

Probleme la job. multor thr.

→ au avere sufic. de mare

→ multe thr. pt. serv. → multe op. secvențiale pt. proces. pr.

→ te costă să pornești / oprești threaduri ⇒ context switch.

→ se det. empiric

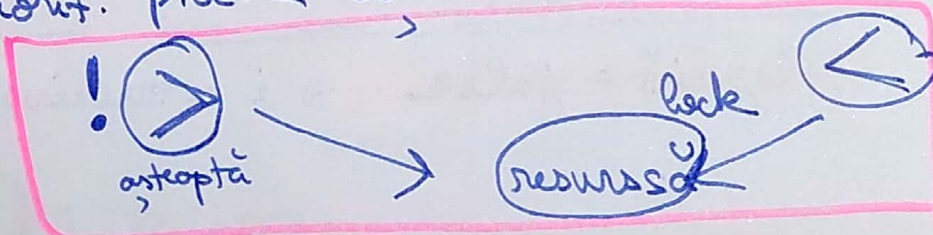
## Starvation

: un proces cu priorit. < are lockul pe o res. ~~cu prior. mare~~. care este solicit. de un proc. cu prior. mare.

un proc. are dreptă / după un altul cu priorit. mică → o resursă a unui proces

→ proces care nu primește cicluri pe CPU

→ priorit. proc. < crește



= indefinite blocking

ex.: Mars  
pathfinder  
↳ sit. de Starvation



## Problema de sincronizare:

→ consumer / producer

- accesul simultan la counter var. globală

= race condition

(sol<sup>o</sup>) sincronizare = acces sequential la counter.

## Proprietățile soluției

- excludere mutuală: un singur proc. în sec. critică
- progres: → secțiunea critică nu rămâne goală dacă este solicitată  
→ să aibă loc progres :))
- asteptare finită:  $\exists$  o lim. de asteptare pt. fiecare proc.

## Soluția lui Petersen

înt. turnu  $\Rightarrow$  al cui e rândul :)) (am scris din greșală așa)  
boolean flag [2]  $\Rightarrow$  cine așteaptă să intre în SC.

• proc. i:

do { l. vrea să intre în sec. crit. }

flag[i] = TRUE

turn = j;  $\rightarrow$  este rândul lui j

while (flag[j] && turn == j);  $\rightarrow$  barieră

$\rightarrow$  i așteaptă să termine j  $\leftarrow$

SEQUENTIAL  
SC

flag[i] = false  $\rightarrow$  i a terminat în SC

{ while (True);

ACCES ALTERNATIV!



① operatie atomica  $\Rightarrow$  nu poate fi intrerupta.

1) `boolean testandset (boolean *target)`  
`{`  
    `boolean rv = *target;`  
    `*target = TRUE;`  
`}`  
    `return rv;`

`do { while (testandset(&lock))`

`cs`

`lock = FALSE;` scuti lacatul

`} while (TRUE);`

$\Rightarrow$  inchizi usa dupa tine  $\Rightarrow$  si o deschizi la final.

SOL.

HARDWARE

ATOMIC

$\downarrow$

operatii assembly  
lock ....

⊗ asteptare  
'finita'

$\downarrow$

NU E  
ALTERNATIV

2)

`do {`

`key = TRUE`

`while (key == TRUE)  $\Rightarrow$  resursa ocupata`

`swap(&lock, &key);`

`cs`

`lock = FALSE`

`} while (TRUE);`

⊗ asteptare finita  
 $\downarrow$   
NU E ALTERNATIV

busy waiting (spin lock)

$\rightarrow$  tot core-ul  
este ocupat.

`do { wait(mutex)`

`// cs`

`signal(mutex)`

`// ...`

`} while (true);`

$\rightarrow$  consuma

$\downarrow$

verifica mereu  
daca  $> 0$ .

\* nu se asteapta pasiv,  
fara sa calculeze ceva.



# Problema Bufferului limitat

## → Bounded - Buffer Problem

— avem  $n$  poziții în buffer.

— semafoare:

→ mutex = 1

→ empty =  $n$

→ full = 0

producator {

<sup>full count</sup>  
wait(empty) → mai e loc?

NA ⇒ wait(mutex) → e resursa goală (neocupată)?

DA → adaugă obiect

<sup>empty count</sup>  
signal(mutex) → res. e goală

signal(full) → am adăugat ceva

<sup>empty count</sup>

→ exclusiune mutuală

accesul la buffer este  
secvențial

## SCRIERI / citiri simultane

sem: — mutex pt. count — readers = 0

— wrt → accesul la scriere.

do

wait(wrt)

// scrie

signal(wrt)

lock  
pe  
scriere

while(true);

lock {  
++ readcount = 1  
wait wrt  
block scriere  
citire tota

lock {  
-- readcount = 0  
signal wrt  
= anunțat  
poate scrie 4.



## Curs 9 (So)

### Problema filosofilor

semajor: chopstick [5] ? ~~stă în afara~~ ~~filosofii sunt~~ NUMERCE un sem per 1

→ lock pe ambale betisoare pt. un filosof.

→ operații ATOMICE  
= nu poate fi întreruptă.

? Monitors ⇒ o construcție cu rol de semajor.  
mai sofisticată

→ ~~variabile de condiție~~ x, y

pickup  
putdown

test

stări → thinking, hungry, eating

~~x.wait()~~

~~x.signal()~~

testează starea  
când termină  
de mâncat  
și când se apucă

→ o implen. a unui sem. în care un progr. poate  
face numai puține erori.

## Sistemul de fisiere

File - System Interface

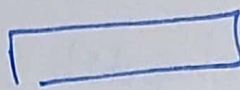
→ meta datele unui fisier (extensie, dimensiune, etc)

→ software care rearanjează inf. de hd. a.i. să  
fie cât mai grupată → DEFRAGMENTER

fisier = ca un vector în hd.



— un ssd = hardware perisabil  
are un nr. finit de citiri / scrieri  
→ nu se face defragment

→ adresa logică continuă  f

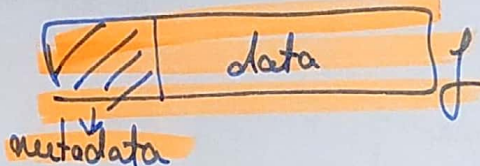
fișiere binare ⇒ exe; pdf; imagini

fișiere text, dar care nu se numesc text : csv, json, html  
↓  
posibile  
adica  
nu sunt .txt

→ hex editor (vizualiz. fiș. bin)

nu ai caractere, ci codul hexazecimal al caracterului

Atribute fișier :



- nume
- identificator (unic) = nr.
- type
- location
- size

→ calea către fișier + nume fișier <sup>define</sup> numele fișierului  
complet unic al

Linux: restricție  
— nume fișier + extensie ⇒ < 255 caractere.

File operations.

open, truncate, write  
read...  
close  
....

→ reposition within file = seek

→ schimbă poziția în cache par. fișierului



Tabele în care toate fișierele = evidența fiș. deschise.  
și fiecare proces → fiș. pe care le-a deschis.

ex: nu poți muta un fișier deschis lock pe el? ~~la~~ altă locație  
în

→ scrierea nu este caracter cu caracter, ci în blocuri  
se acumulează datele într-un buffer ~~chunks~~

→ la sfârșitul scrierii car. cu car. → se dă flush după write.