



## 4.4 Stările unui proces

Un **proces** poate fi, la un moment dat, într-o dintre **trei stări principale**:

1. **Running (Rulează)** – Procesul este activ și execută instrucțiuni pe procesor.
2. **Ready (Pregătit)** – Procesul este pregătit să ruleze, dar sistemul de operare (OS) nu i-a alocat momentan procesorul.
3. **Blocked (Blocat)** – Procesul a inițiat o operație (de exemplu, I/O - input/output), așteaptă un eveniment extern și nu poate rula până acel eveniment se încheie.

---

### Tranzitii între stări:

- Un proces poate fi:
  - **Planificat (scheduled)**: trece din starea Ready în Running.
  - **Dezactivat (descheduled)**: trece din Running în Ready.
  - **Blocat**: trece din Running în Blocked, de obicei când inițiază o operație I/O.
  - **Deblocat**: când operația I/O se finalizează, trece din Blocked în Ready (și posibil imediat în Running).



---

### Exemple de scenarii:

#### 1. Două procese fără I/O:

- Ambele procese alternează între stările Running și Ready.
- Procesorul comută între ele pentru a le oferi timp de execuție.

#### 2. Un proces folosește I/O:

- Procesul 0 rulează, apoi inițiază o operație I/O și devine blocat.
- În acest timp, Procesul 1 rulează.
- După finalizarea I/O, Procesul 0 devine Ready, apoi revine la Running.
- Astfel, CPU-ul este folosit eficient pe durata așteptării.

---

### Deciziile sistemului de operare:

- Sistemul de operare decide **când și pe cine rulează**, printr-un modul numit **scheduler**.
- Acesta trebuie să optimizeze performanța și utilizarea resurselor, de exemplu:
  - să nu țină CPU-ul inactiv când un proces este blocat;
  - să decidă dacă să schimbe imediat înapoi la un proces deblocat sau nu.



## Fișă de învățare – Scheduling: Introduction

### ◆ Ce este scheduling-ul?

Este politica prin care sistemul de operare decide **ce proces să ruleze și când**, pentru a **maximiza performanța și/sau echitatea**.

---

### ◆ Obiectivul principal:

**Cum dezvoltăm o politică de planificare eficientă (scheduling policy)?**

---

### ◆ 7.1 Workload Assumptions (Presupuneri despre sarcină)

Initial se fac presupuneri simple pentru claritate:

1. Fiecare job are aceeași durată.
2. Toate joburile sosesc simultan.
3. Fiecare job rulează până la final, fără intrerupere.
4. Nu există I/O – doar folosire CPU.
5. Timpul de execuție e cunoscut dinainte.

### ◆ Aceste presupuneri sunt nerealiste, dar utile pentru a introduce conceptele.

---

### ◆ 7.2 Scheduling Metrics (Metrice pentru analiză)

#### ▀ Turnaround time:

$$T_{turnaround} = T_{completion} - T_{arrival}$$

- Pentru început, toate joburile sosesc în același moment  $\rightarrow T_{arrival} = 0$ .
  - Alte metrice: **echitate** (ex: Jain's Fairness Index).
- 

### ◆ 7.3 FIFO (First In, First Out)

- Cel mai simplu algoritm.
- Răspunde la ordine de sosire: primul venit, primul servit.
- Bun pentru joburi de durată egală.

#### ! Problemă:

Dacă un job lung este primul, restul așteaptă → **efectul de convoi** (convoy effect).

---

#### ◆ 7.4 SJF (Shortest Job First)

- Răspunde la problemă prin execuția celor mai **scurte joburi mai întâi**.
- Reduce dramatic timpul mediu de așteptare.

 Ex:

- Joburi A (100s), B (10s), C (10s)
  - FIFO → timp mediu: 110s
  - SJF → timp mediu: 50s

 **SJF este optim** pentru joburi cunoscute și sosire simultană.  
Dar ce facem când joburile sosesc **în momente diferite**?

---

#### ◆ Limitările SJF (când joburile nu sosesc simultan)

Ex:

- A (sosire la t=0, durată 100s)
  - B și C (sosire la t=10, durată 10s)
- ◆ SJF pur => B și C așteaptă până când A termină → **ineficiență**  
→ Timp mediu: **103.33s**
- 

### i Alte concepte:

#### ◎ Preemptive Scheduling:

- Sistemul oprește un job pentru a rula altul (ex: un job mai scurt sosit între timp).
  - Necesită mecanisme de context switching.
- 

## Întrebări tip Quiz – Scheduling: Introduction

### 1. Ce metrică este folosită pentru a evalua performanța algoritmilor de scheduling?

- a) Timpul total de boot
  - b) Timpul de turnaround
  - c) Numărul de procese
- **Răspuns corect: b**

---

**2. Ce presupunere este nerealistă dar utilă pentru începutul analizei?**

- a) Joburile rulează aleator
  - b) Toate joburile folosesc doar I/O
  - c) Fiecare job are timp de execuție cunoscut
- **Răspuns corect:** c
- 

**3. Care este principala problemă a FIFO?**

- a) Necesită prea multă memorie
  - b) Creează efectul de convoi dacă joburile au durate diferite
  - c) Ignoră prioritatea utilizatorilor
- **Răspuns corect:** b
- 

**4. De ce este SJF o alternativă mai bună la FIFO?**

- a) Rulează cele mai scurte joburi primele, reducând timpul de așteptare
  - b) Rulează cele mai recente joburi primele
  - c) Rulează joburile cu cel mai mare ID numeric
- **Răspuns corect:** a
- 

**5. Ce se întâmplă dacă un job lung începe, iar joburi scurte sosesc ulterior în SJF pur?**

- a) Joburile scurte sunt prioritizate imediat
  - b) Joburile scurte așteaptă ca jobul lung să termine
  - c) Scheduler-ul le ignoră
- **Răspuns corect:** b
- 

**6. Ce este un scheduler preemptiv?**

- a) Scheduler care rulează procese fără oprire
  - b) Scheduler care poate întrerupe un proces pentru a rula altul
  - c) Scheduler care rulează doar un singur job
- **Răspuns corect:** b
- 

## **STCF – Shortest Time-to-Completion First (Preemptive SJF)**

 **Ce este STCF?**

STCF este o versiune **preemptivă** a algoritmului Shortest Job First (SJF). Spre deosebire de SJF, care rulează un job până la final, STCF:

- Preia controlul oricând un **job nou** sosește în sistem,
- Recalculează care job are **cel mai puțin timp rămas**,
- Și rulează **acel** job, chiar dacă înseamnă **preemptarea** celui curent.

### Avantaje:

- Optimizează **timpul mediu de finalizare (turnaround time)**.
- Este **provabil optim** pentru turnaround time dacă lungimile joburilor sunt cunoscute.

### Dezavantaje:

- Nu este bun pentru **response time** (timp până la prima execuție).
- 

## Exemplu concret (Figura 7.5):

### Datele problemei:

- Trei joburi: A, B și C.
- Sosiri:
  - A la timpul **0**
  - B și C la timpul **10**
- Duratele:
  - A: 100 unități
  - B: 10 unități
  - C: 20 unități

### STCF în acțiune:

1. **Timp 0 - 10:** Doar A este disponibil, deci rulează A.
  2. **Timp 10:** Sosesc B (10 unități) și C (20 unități).
    - A are 90 unități rămase.
    - STCF alege **B** (cel mai scurt job disponibil).
  3. **Timp 10 - 20:** Rulează **B** (finalizează la 20).
  4. **Timp 20 - 40:** STCF alege **C** (20 unități rămase).
  5. **Timp 40 - 130:** A rămâne singur, continuă de la 90 → finalizează la 130.
- 

## Calculul timpului mediu de turnaround:

Turnaround time = finish time - arrival time

- A:  $130 - 0 = 130$
- B:  $20 - 10 = 10$
- C:  $40 - 10 = 30$

## Media:

$$\frac{130 + 10 + 30}{3} = \frac{170}{3} \approx 56.67$$

 Dacă s-ar fi folosit **SJF (fără preemptie)**, A ar fi rulat complet înainte ca B și C să înceapă — rezultând în **timpuri de aşteptare uriaşe** pentru B și C.

---



## Intuiție:

- STCF **profită** de faptul că unele joburi sunt **scurte** și le rulează înaintea celor lungi, chiar dacă au sosit mai târziu.
- Aceasta scade **timpul mediu total până la finalizare**.
- Este ideal pentru **batch systems** (sisteme de procesare în serie), dar nu pentru interactivitate (cum vedem în secțiunile următoare).

# Rezumat – Scheduling: The Multi-Level Feedback Queue (MLFQ)

## ◆ Context și Problemă

- MLFQ este un algoritm istoric (din 1962, Corbato et al.) pentru planificarea proceselor.
  - Problema cheie:
    - Vrem să optimizăm **turnaround time** (prin rularea joburilor scurte primele), dar nu știm **dinainte durata joburilor** (spre deosebire de SJF/STCF).
    - Vrem să oferim **timp de răspuns scurt** pentru joburile interactive, dar metode simple (ex. Round Robin) reduc turnaround time slab.
  - Întrebarea:  
**Cum să planificăm bine fără a cunoaște duratele proceselor?**
- 

## ◆ Ideea centrală: Învățare din istorie

- MLFQ folosește istoria execuției proceselor pentru a-și ajusta prioritățile.
  - Se bazează pe faptul că procesele au "faze" de comportament predictibil (ex. proces interactiv care face I/O).
  - Atenție: predicțiile pot fi greșite și pot afecta negativ sistemul.
- 

## ◆ Regulile de bază ale MLFQ

1. Există mai multe cozi de priorități (ex. Q0=prioritate joasă, Q8=prioritate mare).
  2. La un moment dat, un job este într-o singură coadă.
  3. Jobul din coada cu prioritate mai mare rulează întotdeauna înaintea joburilor din cozi mai joase.
  4. Dacă două joburi au aceeași prioritate (coadă), se folosește **Round Robin** între ele.
- 

## ◆ Cum variază prioritatea?

- Prioritatea unui job nu este fixă, ci se ajustează în funcție de comportamentul său:
    - Dacă un job **folosește tot timpul alocat** (ex. o feliă de timp), își **săde prioritatea** (mutat într-o coadă inferioară).
    - Dacă jobul **renunță voluntar la CPU** înainte de a-și termina feliile de timp (ex. așteaptă I/O), își **păstrează prioritatea** (și i se resetează alocarea).
  - Joburile intră în sistem la **prioritatea maximă** (sus, coada cea mai înaltă).
-

◆ **Exemple importante:**

- **Job lung (CPU-bound):** după fiecare alocare de timp folosită complet, scade prioritatea și în cele din urmă ajunge la coada cea mai joasă, unde rămâne.
  - **Job scurt, interactiv:** păstrează prioritatea ridicată deoarece renunță rapid la CPU (ex. așteaptă I/O).
  - Astfel, MLFQ aproximează SJF fără să știe în prealabil durata joburilor.
- 

◆ **Scopul MLFQ**

- Oferă timp scurt de răspuns pentru procese interactive.
  - Optimizează turnaround time pentru joburi de lungimi variate.
  - Învățare și adaptare din execuția anterioară a proceselor.
- 

## ?

## Întrebări Quiz – MLFQ

**1. Care este scopul principal al MLFQ?**

- a) Să ruleze joburile în ordinea sosirii
  - b) Să optimizeze turnaround time și response time fără a ști durata joburilor
  - c) Să ruleze toate joburile cu prioritate egală
- **Răspuns corect:** b
- 

**2. Cum decide MLFQ ce job rulează în cazul a două joburi cu prioritate egală?**

- a) Jobul sosit primul
  - b) Random
  - c) Round Robin între joburi
- **Răspuns corect:** c
- 

**3. Ce se întâmplă cu prioritatea unui job care folosește tot timpul alocat pentru execuție?**

- a) I se mărește prioritatea
  - b) I se reduce prioritatea (mutat într-o coadă inferioară)
  - c) Rămâne la fel
- **Răspuns corect:** b
- 

**4. Dacă un job renunță voluntar la CPU înainte să-și consume feliile de timp, ce se întâmplă cu prioritatea sa?**

- a) Se reduce prioritatea

- b) Rămâne aceeași, iar alocarea se resetează
  - c) Jobul este eliminat din cozi
- **Răspuns corect:** b
- 

#### **5. Cum tratează MLFQ joburile noi care intră în sistem?**

- a) Le dă prioritate joasă
  - b) Le dă prioritate maximă
  - c) Le plasează aleatoriu în cozi
- **Răspuns corect:** b
- 

#### **6. Care este avantajul major al MLFQ față de SJF?**

- a) Nu necesită cunoașterea duratei joburilor în avans
  - b) Rulează joburile în mod FIFO
  - c) Nu folosește priorități
- **Răspuns corect:** a

### **Continuarea despre Multi-Level Feedback Queue (MLFQ) – Probleme și îmbunătățiri**

#### **Situată prezentată:**

Figura 8.3 (dreapta) arată un exemplu cu un job interactiv B (gri) care folosește CPU doar 1 ms înainte să facă I/O, concurând cu un job lung A (negru). MLFQ ține jobul B la prioritatea maximă pentru că el eliberează rapid CPU-ul, ceea ce face ca joburile interactive să ruleze rapid.

---

### **Problemele MLFQ de bază**

#### **1. Starvation (înfometarea):**

Dacă sunt prea multe joburi interactive, ele pot consuma tot CPU-ul, iar joburile lungi nu primesc timp deloc (starvează).

#### **2. Gaming-ul scheduler-ului (exploatarea sistemului):**

Un utilizator ar putea „păcăli” scheduler-ul: înainte să consume tot timpul alocat, face o operație I/O, eliberând CPU-ul și astfel rămâne în coada cu prioritate mare, obținând mai mult timp CPU decât ar trebui.

### 3. Schimbarea comportamentului jobului:

Un job poate începe CPU-bound și apoi devine interactiv. Scheduler-ul inițial nu îl tratează corect pe parcursul schimbării.

---

## Încercarea #2: Boost de Prioritate

### Ideea:

La intervale regulate S, toate joburile sunt mutate în coada de prioritate maximă.

- **Avantaje:**

- Previne starvation, deoarece joburile lungi vor ajunge periodic în coada superioară și vor primi timp CPU.
- Dacă un job devine interactiv, va fi tratat corect după boost.

- **Problema:**

Alegerea corectă a intervalului S este dificilă („voodoo constant”). Dacă S e prea mare, joburile lungi pot starva. Dacă e prea mic, joburile interactive pot primi prea puțin timp.

---

## Încercarea #3: Contabilizare mai bună (anti-gaming)

- Problema: Joburile pot elibera CPU-ul înainte să își consume toată cota și rămân la aceeași prioritate (gaming).
  - Soluția:
    - Se păstrează evidența cât timp a folosit jobul în total la fiecare nivel, indiferent de câte ori a eliberat CPU-ul.
    - Odată consumată cota, jobul este coborât în coada cu prioritate mai mică.
    - Astfel, indiferent de I/O, jobul va coborî treptat și nu poate monopoliza CPU-ul.
- 

## Ajustarea parametrilor MLFQ

- **Număr de cozi:** De obicei mai multe, cu tempi de execuție (quantum) diferiți.
- **Quantum pe coadă:** Cozile de prioritate înaltă au quantum mici (ex: 10 ms) pentru a deservi rapid joburile interactive. Cozile joase au quantum mari (ex: 100+ ms) pentru joburile CPU-bound.
- **Boost periodic:** Pentru a evita starvation și a ține cont de schimbarea comportamentului joburilor.

Exemplu: Solaris folosește 60 de cozi, quantum crescător, boost la ~1s.

---

## Reguli finale MLFQ (sumar)

1. Prioritatea mai mare rulează înaintea celei mai mici.
  2. Dacă prioritățile sunt egale, rulează round-robin cu quantum specific cozii.
  3. Joburile noi intră în coada de prioritate maximă.
  4. După ce un job consumă cota de timp la un nivel, este coborât o coadă.
  5. După intervalul S, toate joburile sunt mutate în coada de prioritate maximă (boost).
- 

## **Importanța MLFQ**

- Schedulerul observă comportamentul joburilor (feedback).
- Oferă performanță apropiată de SJF pentru joburile scurte și echitată pentru cele lungi.
- Folosit în multe sisteme reale: BSD, Solaris, Windows.

### **1. Ce avantaj are MLFQ față de alți algoritmi de planificare, cum ar fi SJF/STCF?**

#### **Răspuns:**

MLFQ nu presupune cunoașterea prealabilă a duratei procesului (burst time), ceea ce îl face mult mai practic în sisteme reale. În plus, MLFQ poate diferenția joburile interactive de cele CPU-bound și le tratează diferit, oferind un răspuns rapid pentru cele interactive fără să blocheze complet procesele lungi.

---

### **2. Care este problema de „starvation” în MLFQ și de ce apare aceasta?**

#### **Răspuns:**

Starvation apare când procesele cu prioritate mică (de exemplu, CPU-bound care ajung în cozi inferioare) nu mai primesc acces la CPU deoarece cozi cu prioritate superioară sunt mereu ocupate de joburi interactive. Astfel, procesele de prioritate joasă pot rămâne indefinitely în aşteptare.

---

### **3. Cum poate un utilizator „găsi o portiță” (game the scheduler) în algoritmul MLFQ descris inițial?**

#### **Răspuns:**

Un job interactiv ar putea rula până aproape de sfârșitul quantum-ului și apoi să cedeze voluntar CPU pentru a evita coborârea în coada inferioară. Astfel, procesul „șicanează” scheduler-ul pentru a rămâne mereu cu prioritate mare.

---

#### **4. Ce soluție propune „priority boost” pentru a combate starvation-ul și schimbarea comportamentului joburilor?**

**Răspuns:**

Priority boost urcă toate joburile în coada superioară la intervale regulate (perioade S). Astfel, toate procesele au periodic o șansă egală de a fi procesate, evitând ca unele să rămână blocate permanent în cozi inferioare.

---

#### **5. Ce este „Ousterhout’s Law” și cum se aplică în contextul configurării parametrilor MLFQ?**

**Răspuns:**

Ousterhout’s Law spune că interactivitatea depinde în principal de doi factori: mărimea quantum-ului (time slice) și frecvența cu care joburile CPU-bound sunt readuse în cozi superioare (boost rate). Ajustarea acestor parametri influențează direct performanța percepătă de utilizator.

---

#### **6. Cum rezolvă „better accounting” problema jocului cu scheduler-ul și care este regula nouă (Rule 4) propusă?**

**Răspuns:**

„Better accounting” propune să se măsoare timpul activ de CPU efectiv consumat și să se țină cont de cât de des procesul este blocat voluntar (de ex., așteptând I/O). Regula 4 spune că procesul este considerat interactiv dacă a fost blocat înainte de a termina quantum-ul și dacă a folosit mai puțin CPU decât quantum-ul.

---

#### **7. Cum afectează mărimea timpului alocat (time slice) și numărul de cozi funcționarea MLFQ?**

**Răspuns:**

Un time slice prea mic înseamnă frecvențe comutări de context, ceea ce poate crește overhead-ul. Un time slice prea mare înseamnă răspuns mai lent pentru procesele interactive. Numărul de cozi influențează granularitatea diferențierii priorităților — mai multe cozi permit o ajustare mai fină, dar complică managementul scheduler-ului.

---

#### **8. Care este diferența dintre cum tratează Solaris MLFQ și FreeBSD MLFQ prioritizarea proceselor?**

**Răspuns:**

Solaris folosește o politică în care prioritatea se calculează pe baza consumului recent de CPU și boost-uri periodice, fără a separa explicit cozi. FreeBSD, în schimb, folosește mai multe cozi cu politici și reguli diferite de ajustare a priorităților, adaptându-se mai granular la comportamentul proceselor.

---

**9. De ce este importantă securitatea în politica de planificare și cum poate fi afectată aceasta în contextul MLFQ?****Răspuns:**

Politiciile de planificare pot fi exploatare de programe malicioase care încearcă să monopolizeze CPU sau să evite să cedeze procesorul. De exemplu, jocul cu scheduler-ul poate fi folosit pentru a obține prioritate nejustificată. De aceea, politica trebuie să fie robustă și să prevină astfel de abuzuri.

---

**10. Cum pot utilizatorii sau administratorii să ofere „advice” sistemului de operare pentru planificare și alte resurse?****Răspuns:**

Prin API-uri speciale sau setări de parametri (de ex., nice values în Unix), utilizatorii și administratorii pot oferi sugestii despre priorități, comportament așteptat al proceselor sau politică preferată pentru alocarea resurselor.

---

**11. Cum ar trebui să se seteze valoarea perioadei S (intervalul de priority boost) pentru un echilibru bun între interactivitate și progresul joburilor CPU-bound?****Răspuns:**

Valoarea S trebuie să fie suficient de mică pentru a preveni starvation și a menține interactivitatea, dar suficient de mare pentru ca joburile CPU-bound să progreseze semnificativ. Ajustarea depinde de tipul de sarcini și de caracteristicile sistemului.

---

**12. Ce se întâmplă în cazul în care un job își schimbă comportamentul de la CPU-bound la interactiv? Cum gestionează MLFQ această situație?****Răspuns:**

Mecanismul de „better accounting” detectează schimbarea comportamentului prin modul în care jobul folosește CPU și când se blochează. Astfel, jobul poate fi promovat în coada superioară dacă devine interactiv, oferindu-i prioritate mai mare.

---

### **13. Care sunt avantajele și dezavantajele folosirii unui număr mare de cozi în MLFQ?**

**Răspuns:**

Avantaje: permite o ajustare fină a priorităților și poate diferenția mai bine tipurile de joburi.  
Dezavantaje: crește complexitatea scheduler-ului, timpul de administrare, și poate introduce overhead la comutarea între cozi.

---

### **14. De ce este util să se permită joburilor să fie readuse în coada superioară periodic?**

**Răspuns:**

Pentru a preveni starvation-ul joburilor de prioritate joasă și pentru a permite joburilor să își revină dacă și-au schimbat comportamentul (de ex., devin mai interactive). Astfel se asigură o alocare mai echitabilă a resurselor.

## Capitolul 13 – Abstractizarea: Spațiile de adrese

### 💡 Ideea principală

Sistemul de operare oferă fiecărui program iluzia că are propria sa memorie – numim asta **spațiu de adrese**.

În realitate, toate programele folosesc aceeași memorie fizică, dar cu ajutorul hardware-ului și al OS-ului, această memorie este „virtualizată”.

### 📋 13.1 Sistemele de operare timpurii

- La început, era simplu: un singur program rula în memorie.
- OS-ul era doar o bibliotecă de funcții, pusă la începutul memoriei fizice.
- Utilizatorii nu aveau așteptări mari – deci viața dezvoltatorilor era mai ușoară.

### ⌚ 13.2 Multiprogramare și partajare în timp (Time Sharing)

- **Multiprogramare**: mai multe programe sunt în memorie, OS-ul le comută când unul așteaptă I/O.
- **Time sharing**: comutare rapidă între procese → iluzia că toate rulează simultan.
- Inițial, salvarea și restaurarea întregii memorii era lentă.
- Soluție: păstrarea mai multor procese simultan în RAM → eficiență crescută.
- A apărut nevoie de **protectie** între procese (să nu poată accesa memoria altora).

### 📦 13.3 Spațiul de adrese

- OS-ul oferă fiecărui proces **un spațiu de adrese propriu** – adică o vedere virtuală asupra memoriei.
- Componente importante ale spațiului de adrese:
  - **Codul programului** – unde sunt instrucțiunile.
  - **Stack-ul** – pentru variabile locale, apeluri de funcții, etc. (crește în jos).
  - **Heap-ul** – pentru alocări dinamice (crește în sus).
- Deși programul „vede” adrese de la 0 în sus, în realitate el este încărcat în altă parte în RAM.
- Aceste adrese sunt **virtuale** – OS-ul le mapează pe cele reale (fizice).

## ⌚ 13.4 Obiectivele virtualizării memoriei

1. **Transparentă** – programul nu trebuie să știe că memoria e virtualizată.
2. **Eficiență** – performanță bună (viteză și spațiu).
3. **Protecție** – procesele să nu interfereze între ele.

💡 **Izolarea** este esențială pentru fiabilitate: dacă un proces greșește, să nu afecteze pe altele.

## ✓ 13.5 Concluzie

- Virtualizarea memoriei este una dintre cele mai importante funcții ale unui OS modern.
- OS-ul și hardware-ul colaborează pentru a traduce adresele virtuale în adrese fizice.
- Această traducere permite rularea mai multor programe, în siguranță și eficient.

## 🔍 Exemplu C: Toate adresele sunt virtuale

```
c

#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("code @ %p\n", main);
    printf("heap @ %p\n", malloc(100));
    int x = 3;
    printf("stack @ %p\n", &x);
    return 0;
}
```

Copy Edit

- Toate valorile afișate sunt adrese **virtuale**. Doar OS-ul știe unde sunt ele în RAM fizic.

## 📘 Fișă de învățare – Address Spaces

### Ce este un address space?

- Este spațiul virtual de memorie pe care OS îl oferă fiecărui proces, astfel încât acesta să credă că are toată memoria pentru el singur.

## ◆ 13.1 Early Systems

- Nu există protecție: un singur program rula la un moment dat.
  - Codul era încărcat direct în memorie fizică.
- 

#### ◆ **13.2 Multiprogramming și Time Sharing**

- Se pot rula mai multe programe simultan.
  - OS face comutări rapide între ele.
  - Necesită protecție între procese.
- 

#### ◆ **13.3 Goals of Virtual Memory**

- Iluzia unui spațiu continuu de memorie.
  - Protecția între procese.
  - Izolarea și relocarea codului.
- 

#### ◆ **13.4 Mechanism: Address Translation**

- Folosește **MMU (Memory Management Unit)** pentru a converti adrese virtuale în adrese fizice.
  - Traducerea se face prin:
    - Segmentare (simplu, dar limitat)
    - Paginare (cel mai des folosit)
- 

#### ◆ **13.5 Performance: TLB (Translation Lookaside Buffer)**

- Cache special pentru adrese traduse.
  - Reduce numărul de accesări lente în tabelele de pagini.
- 

### Quiz – Address Spaces

#### **1. Ce este un address space și de ce este important?**

- a) O locație fizică din RAM
  - b) Spațiul virtual de memorie alocat fiecărui proces
  - c) Un fișier de configurare al sistemului de operare
- Răspuns corect: b
-

**2. Ce rol are MMU (Memory Management Unit)?**

- a) Rulează aplicații din kernel
  - b) Traduce adresele fizice în adrese virtuale
  - c) Traduce adresele virtuale în adrese fizice
- Răspuns corect: c
- 

**3. Ce este un TLB (Translation Lookaside Buffer)?**

- a) O memorie RAM suplimentară
  - b) O unitate care gestionează fișiere
  - c) Un cache care stochează traduceri recente de adrese
- Răspuns corect: c
- 

**4. Ce tehnică de adresare oferă cea mai bună flexibilitate și performanță?**

- a) Segmentare
  - b) Paginare
  - c) Multiplexare
- Răspuns corect: b
- 

**5. De ce este importantă protecția memoriei?**

- a) Pentru a preveni încălzirea procesorului
  - b) Pentru a proteja datele între procese
  - c) Pentru a reduce dimensiunea RAM-ului
- Răspuns corect: b

## Rezumat 7.3 – Basic Strategies (Strategii de bază)

Această secțiune prezintă patru strategii simple de gestionare a spațiului liber în alocarea memoriei:

- **Best Fit:** caută cel mai mic bloc liber care să se potrivească cererii, minimizând risipa de spațiu, dar necesită o căutare exhaustivă, ceea ce poate fi lent.
- **Worst Fit:** alege cel mai mare bloc liber, încercând să lase blocuri mari disponibile pentru alocări viitoare, însă poate duce la fragmentare excesivă și costuri mari de căutare.
- **First Fit:** alocă primul bloc suficient găsit, fiind rapidă și evitând căutările complete, dar poate cauza fragmente mici în partea de început a listei.
- **Next Fit:** continuă căutarea de unde s-a oprit ultima dată în listă, pentru a distribui alocările și evita fragmentarea locală, având performanțe similare cu First Fit.

Fiecare strategie are avantaje și dezavantaje, iar alegerea optimă depinde de contextul și tiparul de utilizare.

---

## Rezumat 7.4 – Other Approaches (Alte abordări)

Pe lângă strategiile de bază, există metode mai avansate și specializate:

- **Liste segregate:** păstrează liste separate pentru blocuri de dimensiuni populare, reducând fragmentarea și accelerând alocările pentru aceste dimensiuni.
- **Slab Allocator:** o variantă avansată de liste segregate folosită în kernelul Solaris, care păstrează obiectele libere pre-initializate pentru a evita costurile repetate de inițializare și distrugere.
- **Buddy Allocation:** împarte memoria în blocuri cu dimensiuni puteri ale lui 2 și permite coalescență rapidă a blocurilor alăturate („buddy”) când sunt eliberate, facilitând gestionarea spațiului liber și reducând fragmentarea externă.
- **Structuri complexe de date:** pentru scalabilitate, alocatoarele folosesc arbori echilibrați sau alte structuri pentru a accelera căutările și alocările.
- De asemenea, alocatoarele moderne sunt proiectate să funcționeze eficient în medii multiprocesor și multi-thread.

Acste tehnici avansate permit o gestionare mai eficientă și adaptată a memoriei în sisteme complexe.

## Întrebări despre „Basic Strategies” și alte metode de gestionare a memoriei

### 1. Care este obiectivul ideal al unui allocator de memorie?

Răspuns: Să fie rapid și să minimizeze fragmentarea.

### 2. Cum funcționează strategia „Best Fit”? Care este avantajul și dezavantajul ei?

Răspuns: Caută cel mai mic bloc liber care să fie suficient de mare pentru cerere.

Avantaj: reduce spațiul irosit. Dezavantaj: cost mare de căutare.

### 3. Cum funcționează strategia „Worst Fit” și de ce nu este recomandată în general?

Răspuns: Alege cel mai mare bloc liber pentru alocare. Nu e recomandată pentru că duce la fragmentare mare și are cost ridicat.

### 4. Ce face strategia „First Fit” și care este avantajul ei principal?

Răspuns: Alocă primul bloc suficient de mare găsit în lista liberă. Avantaj: rapiditate, căutare parțială.

### 5. Cum funcționează „Next Fit” și ce încearcă să evite față de „First Fit”?

Răspuns: Continuă căutarea de unde s-a oprit ultima dată, pentru a dispersa alocările și a evita fragmentele mici la începutul listei.

### 6. În exemplul cu blocurile de dimensiuni 10, 30 și 20, ce bloc ar alege „Best Fit” pentru o cerere de 15?

Răspuns: Blocul de 20 (cel mai mic care poate satisface cererea).

### 7. Ce este o listă segregată (segregated list) și care sunt beneficiile ei?

Răspuns: Liste separate pentru blocuri de dimensiuni populare. Beneficii: mai puțină fragmentare și alocări rapide pentru dimensiuni frecvente.

### 8. Cum funcționează slab allocator-ul lui Jeff Bonwick și ce avantaj aduce?

Răspuns: Menține obiecte libere într-o stare pre-initializată, evitând costurile de initializare și distrugere repetitive, oferind astfel performanță mai bună.

### 9. Ce este alocarea „buddy” și cum face ea simplă coalescarea blocurilor libere?

Răspuns: Împarte memoria în blocuri de dimensiune putere a lui 2. La eliberare, verifică „buddy”-ul (blocul „frate” care diferă într-un singur bit în adresă) pentru a face coalescere recursivă.

### 10. Care este dezavantajul potențial al alocării buddy?

Răspuns: Fragmentarea internă, pentru că alocă doar blocuri cu dimensiuni putere a lui 2, uneori mult mai mari decât cererea.

### 11. De ce devin căutările în liste lente în cazul unor alocatoare simple și ce structuri de date se folosesc pentru a îmbunătăți performanța?

Răspuns: Pentru că lista crește și căutarea este liniară. Se folosesc arbori echilibrați, arbori splay sau arbori parțial ordonați pentru a accelera căutarea.

### 12. Cum influențează sistemele multiprocesor și aplicațiile multi-thread alocarea memoriei?

Răspuns: Necesită alocatoare care funcționează eficient concurrent, evitând blocajele și coliziunile, pentru a menține performanță.

### 13. Ce exemplu real de allocator modern poți studia pentru a înțelege cum funcționează alocatoarele în practică?

Răspuns: Alocatorul glibc (GNU C Library).

## 18.1 Paging: Introducere și exemplu simplu

- Sistemele de operare gestionează memoria virtuală folosind două abordări: segmentare (segmente de dimensiuni variabile) sau paginare (bucăți de dimensiuni fixe).
- Segmentarea poate duce la fragmentare, ceea ce complică alocarea memoriei.
- Paginarea împarte spațiul de adrese virtuale în pagini de dimensiuni fixe (ex.: 16 bytes) și memoria fizică în cadre de pagină de aceeași dimensiune.
- Exemplul arată un spațiu de adrese virtuale de 64 bytes împărțit în 4 pagini de câte 16 bytes fiecare și o memorie fizică de 128 bytes împărțită în 8 cadre.
- Fiecare pagină virtuală este mapată într-un cadru fizic diferit, iar sistemul de operare ține o **tabelă de pagini** pentru fiecare proces, care stochează aceste traduceri (ex: pagina virtuală 0 → cadrul 3).
- La accesarea unei adrese virtuale, aceasta se împarte în număr de pagină virtuală (VPN) și offset (adică poziția în pagină). VPN este folosit pentru a găsi cadrul fizic corespunzător.
- Exemplul detaliază traducerea adresei virtuale 21 într-o adresă fizică.

## 18.2 Unde sunt stocate tabelele de pagini?

- Tabelele de pagini pot fi foarte mari: pentru un spațiu de adrese de 32 biți și pagini de 4KB, sunt 1 milion de intrări, ceea ce înseamnă 4MB per tabelă de pagini.
- Pentru 100 de procese, ar fi nevoie de 400MB doar pentru tabele de pagini.
- Tabelele de pagini sunt stocate în memoria fizică gestionată de OS, nu în hardware specializat din MMU.
- În sisteme moderne, tabelele de pagini pot fi virtualizate și chiar swap-uite pe disc, dar asta e mai avansat.

## 18.3 Ce conține o intrare din tabela de pagini?

- Tabela de pagini este o structură (de obicei un array) care mapează VPN la PFN (număr cadrului fizic).
- Fiecare intrare (PTE - Page Table Entry) conține:
  - Bit de validitate (valid/present) care arată dacă traducerea este validă.
  - Biți de protecție (citire, scriere, execuție).
  - Bitul prezent (indică dacă pagina e în memorie sau pe disc).
  - Bit dirty (modificată sau nu).
  - Bit accesat (folosit pentru politici de înlocuire a paginilor).
- Exemplul de PTE x86 conține aceste biți și câțiva pentru gestionarea cache-ului.
- Dacă bitul prezent e 0, accesul generează o excepție, iar OS decide ce face (swap înapoi sau termină procesul).

---

## Întrebări și răspunsuri

### 1. Ce este paginarea și de ce este folosită în locul segmentării?

Paginarea împarte memoria în bucăți de dimensiuni fixe (pagini), evitând fragmentarea

spațiului de adresă care apare la segmentare cu dimensiuni variabile. Aceasta simplifică gestionarea spațiului și alocarea memoriei.

## 2. Cum este reprezentată o adresă virtuală în sistemul cu paginare?

O adresă virtuală este împărțită în două părți: numărul paginii virtuale (VPN) și offsetul în cadrul acelei pagini. VPN este folosit pentru a găsi cadrul fizic corespunzător.

## 3. Ce este tabela de pagini și ce rol are?

Tabela de pagini este o structură de date care stochează traducerile dintre paginile virtuale și cadrele fizice. Fiecare proces are propria sa tabelă de pagini.

## 4. Unde sunt stocate tabelele de pagini?

Tabelele de pagini sunt stocate în memoria fizică, gestionată de sistemul de operare, nu în hardware-ul MMU.

## 5. De ce tabelele de pagini pot fi foarte mari?

Pentru că fiecare pagină virtuală necesită o intrare, iar spațiile de adrese pot fi foarte mari (ex:  $2^{20}$  pagini pentru 32-bit cu pagini de 4KB), rezultând tabele de milioane de intrări.

## 6. Ce informații conține o intrare din tabela de pagini?

Un PTE conține bitul valid/present, biți de protecție (citire, scriere, execuție), bitul dirty, bitul de acces, și numărul cadrului fizic.

## 7. Ce se întâmplă dacă un proces încearcă să acceseze o pagină cu bitul de prezent setat pe 0?

Se generează o excepție (trap) către OS, care poate decide dacă să încarce pagina din disc sau să opreasă procesul (în caz de acces ilegal).

## 18.4 Paging: Also Too Slow

- Traducerea adresei virtuale în adresă fizică se face accesând **tabela de pagini**, care este în memorie.
- Astfel, pentru fiecare acces la memorie (citire sau scriere), sunt două accesări în memorie:
  1. Acces la tabela de pagini pentru a obține intrarea corespunzătoare (PTE).
  2. Acces la adresa fizică calculată pentru date.
- Exemplu:
  - Adresa virtuală se separă în VPN (numărul paginii virtuale) și offset.
  - VPN este folosit pentru a calcula adresa în memoria fizică unde se găsește intrarea în tabela de pagini:
$$\text{PTEAddr} = \text{PTBR} + (\text{VPN} * \text{sizeof(PTE)})$$
(unde PTBR = registrul ce indică începutul tabelei de pagini în memorie)
  - Se citește PTE din memorie.
  - Se extrage PFN (numărul cadrului fizic).
  - Se formează adresa fizică finală concatenând PFN cu offsetul.
- Această dublă accesare face ca procesul să fie semnificativ mai lent (de 2x sau mai mult).
- Este o problemă importantă de performanță: paginarea necesită o metodă de a accelera această traducere (ce urmează să vedem în capitolele următoare).

---

## 18.5 A Memory Trace

- Un exemplu simplu de program C: inițializarea unui array de 1000 de elemente cu 0.
  - Se arată instrucțiunile în assembly care efectuează această operație (în buclă).
  - Se presupune o memorie virtuală de 64 KB, pagini de 1 KB.
  - Codul și array-ul se află pe pagini virtuale diferite (VPN-uri distincte).
  - Sunt date mappări virtual → fizic pentru pagina codului și pentru paginile array-ului.
  - Pentru fiecare acces la memorie (instrucțiune sau date), se fac două accesări: una la tabela de pagini și una la memoria fizică.
  - În primele 5 iterații ale buclei, se pot observa 10 accesări la memorie pe iterație (instrucțiuni + date + accesări la tabela de pagini).
  - Acest exemplu arată clar cât de multe accesări suplimentare implică paginarea și complexitatea gestionării memoriei.
- 

## 18.6 Summary

- Paging oferă o metodă elegantă de virtualizare a memoriei, evitând fragmentarea externă și permitând utilizarea eficientă a spațiului virtual.
  - Totuși, fără optimizări, paginarea produce două probleme majore:
    1. Necesitatea multor accesări suplimentare la tabela de pagini, care încetinesc accesul la memorie.
    2. Consumul mare de memorie pentru stocarea tabelelor de pagini.
  - Următoarele capituloare vor prezenta soluții pentru aceste probleme (ex: TLB-uri).
- 

## Explicații suplimentare / ce înseamnă pentru sistem

- Registrul PTBR (Page Table Base Register) este un registru special ce ține adresa de start a tabelei de pagini pentru procesul curent.
- Fiecare acces la memorie trebuie să facă această traducere virtual→fizic, deci sistemul face două accesări efective la memorie.
- Pentru a evita încetinirea, hardware-ul folosește cache-uri speciale pentru intrările din tabela de pagini (TLB - Translation Lookaside Buffer), care nu sunt încă discutate aici, dar care sunt esențiale.
- Exemplul cu inițializarea array-ului arată cum codul și datele pot fi pe pagini virtuale diferite, necesitând mappări multiple și accesări repetitive la tabela de pagini.

### 1. De ce poate paging-ul să încetinească accesul la memorie?

*Paging-ul poate încetini accesul deoarece pentru fiecare acces la memorie trebuie să se facă mai întâi o accesare la tabela de pagini pentru a traduce adresa virtuală în fizică, apoi încă o accesare pentru datele efective. Astfel, numărul total de accesări la memorie se dublează.*

---

## **2. Ce este registrul PTBR și ce rol are în traducerea adreselor?**

*PTBR (Page Table Base Register) este un registru special care ține adresa de început a tabelei de pagini pentru procesul curent. Hardware-ul folosește această adresă pentru a găsi intrarea potrivită în tabela de pagini când traduce o adresă virtuală.*

---

## **3. Cum se calculează adresa intrării în tabela de pagini (PTEAddr) pentru o adresă virtuală?**

*Se extrage VPN (Virtual Page Number) din adresa virtuală, apoi se calculează:*

$$\text{PTEAddr} = \text{PTBR} + (\text{VPN} * \text{sizeof(PTE)})$$

*unde PTBR este adresa de bază a tabelei, iar sizeof(PTE) este mărimea unei intrări în tabelă.*

---

## **4. Ce se întâmplă după ce s-a găsit intrarea corespunzătoare în tabela de pagini?**

*Se extrage PFN (Page Frame Number) din intrarea găsită. Adresa fizică se formează concatenând PFN (mutat la stânga cu SHIFT) cu offsetul din adresa virtuală:*

$$\text{PhysAddr} = (\text{PFN} \ll \text{SHIFT}) \mid \text{offset}.$$

## **5. În exemplul dat, ce presupuneri s-au făcut despre dimensiunea spațiului virtual și mărimea paginii?**

*Spațiul virtual este de 64KB, iar mărimea paginii este de 1KB.*

---

## **6. Ce tip de probleme cauzează paginarea fără optimizări suplimentare?**

\*Problemele sunt:

- Încetinirea accesului la memorie din cauza accesărilor suplimentare la tabela de pagini;
- Consum excesiv de memorie pentru stocarea tabelelor de pagini.\*

## **7. Cum se traduce o adresă virtuală 21 în exemplul de la început?**

*Adresa 21 (bin 010101) este mascată cu VPN\_MASK 0x30 (bin 110000), rezultând 010000, care după shift devine 01 (VPN=1). Aceasta înseamnă că pagina virtuală este pagina 1.*

---

## **8. De ce există un risc ca paginarea să fie lentă pentru fiecare acces la memorie?**

*Pentru că fiecare acces trebuie să consulte tabela de pagini în memorie, iar accesarea memoriei pentru tabela de pagini adaugă latență suplimentară față de accesul direct la date.*

---

**9. Ce face instrucțiunea `movl $0x0, (%edi,%eax,4)` în exemplul cu array-ul?**

*Scrie valoarea 0 în adresa calculată ca `%edi + %eax*4`, unde `%edi` este baza array-ului și `%eax` este indexul elementului.*

---

**10. În cadrul exemplului, câte accesări la memorie apar pentru fiecare iterație a buclei?**

*10 accesări: 4 pentru instrucțiuni, 1 pentru scrierea în array, și 5 pentru accesările tabelei de pagini care traduc adresele.*

---

**11. Ce sunt PFN și VPN?**

*VPN (Virtual Page Number) este indexul paginii în spațiul virtual, iar PFN (Page Frame Number) este indexul cadrului fizic în memoria fizică.*

---

**12. Ce soluții urmează să fie prezentate pentru problema performanței în paginare?**

*Soluții precum folosirea unui cache hardware pentru tabelele de pagini, numit TLB (Translation Lookaside Buffer), care reduce numărul de accesări la memorie necesare pentru traducere.*

## 39.14 Hard Links

- **Ce este un hard link?**

Un hard link este o nouă referință (nume) pentru același fișier din sistemul de fișiere. Practic, creezi o altă intrare în director care arată către același inode (metadatele fișierului).

- **Cum se creează?**

Prin apelul de sistem `link(olddpath, newpath)` sau comanda shell `ln oldfile newfile`.

- **Ce se întâmplă când creezi un hard link?**

- Nu se copiază datele fișierului.
- Se face o nouă legătură către același inode.
- Astfel, fișierul are două (sau mai multe) nume ce indică același conținut.

- **Cum verifici?**

Folosești `ls -i` pentru a vedea inode-ul fișierului. Dacă două fișiere au același inode, sunt hard links.

- **De ce se numește unlink?**

Pentru că atunci când ștergi un fișier cu `rm` (care apelează `unlink()`), de fapt elimini o legătură (nume) către inode, nu datele în sine. Inode-ul și datele sunt șterse numai când **ultimul hard link dispare** (adică referința în inode scade la 0).

- **Exemplu:**

```
bash
CopyEdit
echo hello > file
ln file file2
rm file
cat file2 # încă va afișa "hello" pentru că file2 mai are legătura
          către inode
```

- **Link count:**

Poți vedea numărul de legături cu `stat file`, câmpul „Links” arată câte nume (hard links) există către inode.

---

## 39.15 Symbolic Links (Soft Links)

- **Ce sunt symbolic links?**

Sunt un alt tip de link, care nu fac legătură directă la inode, ci conțin pur și simplu calea (path-ul) către alt fișier.

- **Cum se creează?**

Cu `ln -s target linkname`.

- **Diferență față de hard link:**

- Symbolic link-ul este un fișier special, cu tipul „symlink”.
- Păstrează textul calea către fișierul său.
- Poate face legătură către directoare sau fișiere de pe alte sisteme de fișiere (hard link nu poate).
- Poate deveni „dangling” (referință ruptă) dacă fișierul său dispare.

- **Cum verifici?**  
ls -l arată tipul linkului (primele caractere: „l” pentru link), și către ce țintește.  
stat file2 arată tipul „symbolic link”.
- **Exemplu:**

```
bash
CopyEdit
echo hello > file
ln -s file file2
cat file2 # afișează "hello"
rm file
cat file2 # eroare: fișierul țintă nu mai există, linkul e rupt
```

- **Dimensiunea unui symlink** este egală cu lungimea căii pe care o stochează (ex: 4 bytes pentru "file").
- 

## Recapitulare rapidă:

Caracteristică	Hard Link	Symbolic Link
Leagă direct la inode	Da	Nu, leagă la un path
Poate lega directoare	Nu (de obicei)	Da
Poate lega între sisteme de fișiere	Nu	Da
Link „rupt” posibil	Nu (referință e pe inode)	Da (dacă fișierul țintă dispare)
Tip fișier	Regular	Link simbolic (special)

## Întrebări și răspunsuri despre Hard Links și Symbolic Links

1. **Ce este un hard link într-un sistem de fișiere?**  
Un hard link este o referință suplimentară (un alt nume) către același fișier pe disc, care indică același inode și, prin urmare, același conținut.
2. **Cum poți crea un hard link folosind comanda din linia de comandă?**  
Folosind comanda:  
  
bash  
CopyEdit  
ln fișier\_original fișier\_link
3. **Care este diferența principală între un hard link și un fișier copiat?**  
Hard link-ul nu creează o copie a datelor, ci doar un alt nume care indică același inode și aceleași date, în timp ce copia creează un nou fișier independent.

**4. Ce înseamnă inode și cum este folosit în contextul hard link-urilor?**

Inode este o structură de date care stocă metadatele fișierului (ex: locația datelor, mărimea, permisiunile). Hard link-urile sunt legături care indică același inode.

**5. Ce se întâmplă cu fișierul și datele sale atunci când ștergi un nume (hard link) al fișierului?**

Se elimină legătura (link-ul) respectivă, iar sistemul scade numărul de referințe (link count) din inode. Datele rămân accesibile atât timp cât există cel puțin un hard link.

**6. Cum poți vedea câte hard link-uri există pentru un fișier?**

Folosind comanda:

```
bash
CopyEdit
stat fișier
```

sau:

```
bash
CopyEdit
ls -l
```

în coloana cu numărul de legături (Links).

**7. Ce se întâmplă dacă ștergi toate hard link-urile unui fișier?**

Când link count ajunge la zero, sistemul de fișiere eliberează inode-ul și spațiul pe disc ocupat de fișier, ștergând efectiv datele.

**8. Care sunt limitările unui hard link? (exemplu: ce nu poți face cu hard link-uri?)**

- Nu poți crea hard link către directoare (pentru a evita cicluri în arborele de directoare).
- Nu poți crea hard link între diferite partiții (inode-urile sunt unice doar în cadrul unui singur sistem de fișiere).

**9. Ce este un symbolic link (soft link)?**

Este un fișier special care conține calea către alt fișier sau director, funcționând ca un pointer către acel obiect.

**10. Cum creezi un symbolic link din linia de comandă?**

Folosind comanda:

```
bash
CopyEdit
ln -s fișier_original fișier_link
```

**11. Cum diferă symbolic link-ul de hard link în ceea ce privește modul de stocare și referință?**

Symbolic link-ul este un fișier separat care conține un path către fișierul original, pe când hard link-ul este un alt nume care indică același inode.

**12. De ce poate un symbolic link să devină „dangling” sau rupt?**

Pentru că link-ul conține o cale către un fișier care poate să nu mai existe sau să fie mutat, iar symbolic link-ul nu actualizează automat această cale.

**13. Cum poți identifica dacă un fișier este un symbolic link?**

Folosind comanda `ls -l`, unde un symbolic link este marcat cu un `l` în prima coloană și arată spre fișierul țintă (ex: `file2 -> file`).

**14. Poți crea un symbolic link către un director? Dar un hard link?**

Poți crea symbolic link către un director fără probleme. Hard link către un director nu este permis (sau este foarte restricționat).

**15. Ce simbol este afișat în ls -l pentru un symbolic link? Dar pentru un fișier obișnuit?**

- Symbolic link: l
- Fișier obișnuit: -

**16. Cum poți verifica în terminal numărul de legături (hard links) ale unui fișier?**

Folosind comanda:

```
bash  
CopyEdit  
stat fișier
```

și uitându-te la câmpul „Links”.

**17. Care este dimensiunea unui symbolic link comparativ cu un fișier normal? De ce?**

Dimensiunea symbolic link-ului este egală cu lungimea căii stocate în el, adesea foarte mică (ex: 4 bytes pentru „file”), pe când un fișier normal are dimensiunea conținutului său.

**18. Dacă faci ln file1 file2 și apoi modifici conținutul prin file2, ce se întâmplă când citești conținutul lui file1?**

Modificarea prin file2 afectează și file1, pentru că ambele sunt referințe la același inode și același conținut.

**19. Poți face hard link între fișiere situate pe partitii diferite? Dar symbolic link?**

Nu poți face hard link între partitii diferite (deoarece inode-urile nu sunt globale), dar poți face symbolic link către un fișier de pe altă partitură.

**20. Ce se întâmplă dacă stergi fișierul original, dar ai un symbolic link către el?**

Symbolic link-ul devine „dangling” și, când încerci să accesesezi link-ul, vei primi eroare de tip „No such file or directory”.

## **Rezumat: Implementarea unui sistem simplu de fișiere (vsfs)**

În acest capitol este prezentat vsfs (Very Simple File System), un sistem de fișiere simplificat, bazat pe structuri și concepte din sistemele UNIX. Scopul este de a introduce structurile de pe disc, metodele de acces și politicile uzuale în sistemele de fișiere.

### **1. Conceptul și mentalitatea unui sistem de fișiere:**

- Un sistem de fișiere are două aspecte principale:
  - a) Structurile de date de pe disc care organizează datele și metadata.
  - b) Metodele de acces care leagă apelurile proceselor (open, read, write) de aceste structuri.
- Înțelegerea acestor două aspecte ajută la formarea unui model mental clar al funcționării sistemului.

### **2. Organizarea pe disc a vsfs:**

- Discul este împărțit în blocuri de 4 KB.
- Blocurile sunt grupate astfel:
  - Blocuri pentru datele utilizatorului (data region) — majoritatea spațiului, aici se stochează fișierele efective.
  - Blocuri pentru inoduri (inode table) — stochează metadata fișierelor, cum ar fi dimensiunea, drepturile de acces, și locațiile blocurilor de date.
  - Blocuri pentru bitmap-uri — țin evidența blocurilor și inodurilor libere sau ocupate (inode bitmap și data bitmap).
  - Un bloc pentru superblock — conține informații despre structura și parametrii sistemului de fișiere (număr de inoduri, blocuri etc.).

### **3. Inodul (inode):**

- Inodul este structura principală ce conține metadata unui fișier: dimensiune, permișioni, proprietar, timp de acces/modificare, număr de legături, și adresele blocurilor de date.
- Fiecare inod are un identificator unic (i-number) ce indică poziția sa în tabelul de inoduri.

### **4. Referință către blocurile de date:**

- Inodul conține pointeri direcți către blocurile de date (de exemplu, 12 pointeri direcți).
- Pentru fișiere mari, se folosesc pointeri indirecți care indică blocuri ce conțin la rândul lor pointeri către blocurile de date (pointer indirect simplu, dublu și triplu indirect).
- Această structură ierarhică (multi-level index) permite gestionarea fișierelor foarte mari (până la câteva gigabytes sau chiar mai mult).

### **5. Alternative la pointeri: extente**

- Unele sisteme folosesc extente, care sunt segmente contigüe de blocuri definite printr-un pointer și o lungime, pentru o alocare mai compactă și mai eficientă.

### **6. Observații despre utilizarea fișierelor:**

- Majoritatea fișierelor sunt mici (de obicei ~2 KB).
- Fișierele mari ocupă însă majoritatea spațiului de stocare.
- Sistemul este optimizat pentru acces rapid la fișiere mici, dar poate gestiona și fișiere mari cu ajutorul indexării multiple.

## Întrebări și răspunsuri

### 1. Ce este vsfs?

R: vsfs (Very Simple File System) este o implementare simplificată a unui sistem de fișiere UNIX, folosită pentru a introduce structurile de date, metodele de acces și politicile de bază ale sistemelor de fișiere.

### 2. Care sunt cele două aspecte principale de înțeles despre un sistem de fișiere?

R: (1) Structurile de date de pe disc utilizate pentru organizarea datelor și metadatelor, și (2) metodele de acces care mapează apelurile proceselor (open, read, write) pe aceste structuri.

### 3. Cum este organizat discul în vsfs?

R: Discul este împărțit în blocuri de dimensiune fixă (4 KB), fiecare bloc având o adresă. În exemplul din carte, discul are 64 de blocuri.

### 4. Ce regiuni sunt rezervate pe disc pentru diferite scopuri în vsfs?

R: Sunt rezervate regiuni pentru:

- Superblock (un bloc)
- Bitmap-uri pentru inoduri și blocuri de date (un bloc pentru fiecare)
- Tabelul de inoduri (5 blocuri)
- Regiunea de date (restul blocurilor)

### 5. Ce este un inode și ce informații conține?

R: Un inode este o structură de metadate care stochează informații despre un fișier, cum ar fi tipul, dimensiunea, proprietarul, permisiunile, timpii de acces/modificare și indicii către blocurile de date ale fișierului.

### 6. Cum se organizează blocurile de date ale unui fișier în inode?

R: Inode-ul conține un număr fix de pointeri direcți către blocuri, un pointer indirect care indică un bloc ce conține alți pointeri, un pointer dublu indirect și un pointer triplu indirect pentru a putea susține fișiere foarte mari.

### 7. Ce este un bitmap în contextul sistemului de fișiere?

R: Un bitmap este o structură simplă în care fiecare bit indică dacă un bloc de date sau un inode este liber (0) sau ocupat (1).

### 8. De ce se folosește o organizare în nivele (direct, indirect, dublu indirect) pentru blocurile unui fișier?

R: Pentru a permite stocarea fișierelor foarte mari. Structura optimizată este de tip arbore dezechilibrat, optimizată pentru faptul că majoritatea fișierelor sunt mici și au nevoie de puține pointeri direcți.

**9. Ce conține superblock-ul?**

R: Superblock-ul conține informații despre structura sistemului de fișiere, cum ar fi numărul total de inoduri și blocuri, începutul tabelului de inoduri, și un magic number pentru identificarea tipului de sistem de fișiere.

**10. Care este dimensiunea unui inode și câte inode-uri încape într-un bloc de 4 KB?**

R: Un inode are aproximativ 256 bytes, astfel încât într-un bloc de 4 KB încape 16 inoduri.

## Rezumat: Condition Variables (Variabile de condiție)

### Context:

- În programarea concurrentă, un thread poate dori să aștepte ca o anumită condiție să devină adevărată înainte să continue execuția (ex.: un părinte vrea să aștepte ca un thread copil să termine).
  - Soluția simplă de a aștepta prin „spin” (while loop care verifică o variabilă) consumă inutil CPU și este ineficientă.
- 

### Ce este o variabilă de condiție?

- O variabilă de condiție este o coadă explicită unde thread-urile pot „adormi” (aștepta) până când o condiție devine adevărată.
  - Când un alt thread schimbă starea respectivă (ex.: finalizează o operație), el poate „semnală” (signal) variabila de condiție pentru a trezi unul sau mai multe thread-uri care așteaptă.
- 

### Funcții principale:

- `pthread_cond_wait(cond, mutex)` — thread-ul apelează această funcție pentru a se pune în așteptare pe condiție.
    - Aceasta eliberează mutexul și adoarnează thread-ul atomar.
    - Când thread-ul este trezit prin signal, reacționează la signal.
  - `pthread_cond_signal(cond)` — trezește un thread care așteaptă pe condiția respectivă.
- 

### De ce trebuie mutexul?

- Mutexul protejează accesul la starea partajată (ex.: o variabilă `done` care indică dacă copilul a terminat).
  - Mutexul este deținut când se face `wait` și când se face `signal`.
  - `wait` eliberează mutexul *doar* când adormează thread-ul, și îl recucerește când trezește thread-ul, evitând condițiile de cursă.
- 

### Exemplu simplu – join folosind variabilă de condiție:

- Variabila globală `done = 0` indică dacă copilul a terminat.
- Thread-ul părinte intră în așteptare cu `pthread_cond_wait` dacă `done == 0`.

- Thread-ul copil, când termină, setează `done = 1`, face `pthread_cond_signal` șiiese.
  - Dacă copilul termină înainte ca părintele să intre în aşteptare, părintele verifică starea și nu mai aşteaptă (nu se blochează).
- 

### Probleme frecvente:

- **Lipsa variabilei de stare (`done`)**: dacă copilul face `signal` înainte ca părintele să aștepte, părintele se poate bloca pe vecie. De aceea variabila de stare este crucială.
  - **Lipsa mutexului**: verificarea stării și apelul `wait` trebuie să fie protejate cu mutex; altfel, pot apărea condiții de cursă care duc la blocaje.
- 

### Reguli bune:

- Țineți mutexul în timp ce faceți `wait` sau `signal`.
- Folosiți un ciclu `while` când verificați condiția (pentru a evita probleme cauzate de „spuriușe” treziri).
- Variabila de stare este esențială pentru corectitudine.

## Întrebări și Răspunsuri - Condition Variables

### 1. Ce este o variabilă de condiție?

R: O variabilă de condiție este un mecanism de sincronizare care permite unui thread să se pună în aşteptare (să adoarmă) până când o anumită condiție devine adevărată, fiind apoi trezit de un alt thread care semnalează schimbarea condiției.

---

### 2. De ce este ineficientă metoda de aşteptare prin „spin” (while loop)?

R: Pentru că consumă inutil CPU, deoarece thread-ul verifică continuu condiția fără să cedeze procesorul, ceea ce duce la irosirea resurselor.

---

### 3. Care sunt cele două operații principale ale unei variabile de condiție?

R: `wait()` — pune thread-ul la somn până când condiția este satisfăcută, și `signal()` — trezește unul sau mai multe thread-uri care aşteaptă.

---

### 4. De ce este nevoie ca `wait()` să primească un mutex și să-l dețină?

R: Pentru a proteja condiția verificată împotriva condițiilor de cursă. `wait()` eliberează

mutexul atomar când adoarme și îl recucerește când este trezit, evitând astfel probleme de sincronizare.

---

**5. Ce poate merge prost dacă nu folosim o variabilă de stare (ex. `done`) împreună cu variabila de condiție?**

**R:** Dacă thread-ul care semnalează (de ex., copilul) trimite semnalul înainte ca thread-ul care așteaptă (părintele) să intre în așteptare, atunci părintele poate rămâne blocat la nesfârșit, pentru că semnalul s-a pierdut.

---

**6. De ce se recomandă folosirea unui ciclu `while` în jurul apelului `wait()`?**

**R:** Pentru că semnalul poate fi „spurios” (false wakeup), iar condiția să ar putea să nu fie satisfăcută în realitate; în plus, alt thread poate modifica condiția între timp. Astfel, se verifică mereu condiția reală.

---

**7. De ce este important să deții mutexul când faci `signal()`?**

**R:** Pentru a preveni condițiile de cursă subtile între verificarea condiției și semnalarea acesteia, menținând sincronizarea corectă și evitând pierderea semnalului.

---

**8. Ce face `pthread_cond_wait()` cu mutexul?**

**R:** Eliberează mutexul când pune thread-ul la somn și îl recucerește imediat după ce thread-ul este trezit.

---

**9. Ce se întâmplă dacă semnalezi pe o variabilă de condiție când nu există niciun thread în așteptare?**

**R:** Semnalul se pierde (nu este memorat), iar thread-ul care va aștepta ulterior pe această condiție va rămâne blocat.

---

**10. Care este diferența între o condiție și o variabilă de condiție?**

**R:** Condiția este o stare sau un fapt logic care trebuie să fie adevărat pentru ca thread-ul să continue, în timp ce variabila de condiție este mecanismul de sincronizare care permite așteptarea și semnalarea schimbării acelei condiții.

## **Problema Producer/Consumer (Buffer Limitat)**

Este o problemă clasică de sincronizare între mai multe fire de execuție (threads):

- **Producer** (producătorii) creează date și le pun într-un buffer.
- **Consumer** (consumatorii) iau datele din buffer și le procesează.

Exemplu: un server web multi-threaded, unde un thread pune cererile HTTP într-o coadă (bufferul), iar alte threaduri consumă cererile pentru procesare.

---

## **Problema sincronizării**

Bufferul este o resursă partajată, deci accesul trebuie sincronizat ca să evităm:

- **Race conditions** (condiții de cursă),
  - Acces simultan incorect.
- 

## **Implementare simplă și problemele ei**

Inițial, bufferul are o singură poziție (slot):

- `put()` introduce o valoare în buffer (doar dacă bufferul e gol),
  - `get()` ia o valoare din buffer (doar dacă bufferul este plin).
- 

## **Implementare inițială fără sincronizare corectă**

- Folosind doar un mutex pentru protecția bufferului NU e suficient.
  - Apare problema că un producer poate scrie într-un buffer deja plin, sau un consumer poate citi dintr-un buffer gol.
- 

## **Folosirea variabilelor condiționale (Condition Variables)**

Se introduce:

- un mutex pentru protecție,
  - o variabilă condițională `cond` pentru a aștepta și semnala schimbarea stării bufferului.
- 

## **Prima versiune defectuoasă (cu un singur CV și if)**

- Producerul așteaptă ca bufferul să fie gol (count == 0).
- Consumerul așteaptă ca bufferul să fie plin (count == 1).
- Înainte de `wait()`, folosesc o **instrucțiune if** pentru a verifica condiția.

### Probleme:

1. Dacă există mai mulți consumatori, un thread poate fi trezit dar să constate că bufferul a fost deja consumat de alt thread care s-a trezit înaintea lui — astfel thread-ul se trezește inutil și codul poate eşua (assertion).
  2. Folosirea lui `if` în loc de `while` nu verifică recondiția după trezire — cauza principală a problemei de mai sus.
- 

### Fix: înlocuirea `if` cu `while`

- După ce un thread se trezește din `wait()`, **reverifică condiția**.
- Dacă condiția nu mai e îndeplinită, intră iar la așteptare.

Această tehnică protejează contra "spurious wakeups" (treziri false) și a concurenței între mai mulți consumatori/producători.

---

### A doua problemă: folosirea unei singure variabile condiționale pentru ambele tipuri de threads

- Toți threads așteaptă pe aceeași condiție.
  - Un consumator poate trezi un alt consumator, ceea ce e inutil și duce la blocaje (deadlock).
  - Exemple în text arată cum toți threads pot ajunge adormiți și niciunul să nu mai lucreze.
- 

### Soluția corectă: două variabile condiționale separate

- O variabilă condițională `empty` — pentru așteptarea producătorilor (bufferul gol).
- O variabilă condițională `fill` — pentru așteptarea consumatorilor (bufferul plin).

Astfel:

- Producătorii **așteaptă pe empty** (adică bufferul să aibă spațiu),
  - Consumatorii **așteaptă pe fill** (adică bufferul să conțină date),
  - Producătorii semnalează consumatorii pe `fill` după ce pun date,
  - Consumatorii semnalează producătorii pe `empty` după ce consumă date.
-

## Extindere: Buffer cu mai multe sloturi (coadă circulară)

- Bufferul are mai multe poziții (buffer[MAX]).
  - Două indici: fill\_ptr (unde pune producerul) și use\_ptr (de unde ia consumerul).
  - count ține numărul de elemente din buffer.
- 

## Codul corect pentru producător și consumator

- Folosesc while pentru condiții,
  - Folosesc două variabile condiționale (empty și fill),
  - Sincronizare corectă cu mutex.
- 

## Sfaturi importante

- Folosește întotdeauna **while în loc de if** pentru condiții în jurul pthread\_cond\_wait.
  - Folosește două variabile condiționale separate pentru producători și consumatori.
  - Aceste reguli elimină blocajele și condițiile de cursă.
  - **Întrebări și Răspunsuri - Producer/Consumer și Condition Variables**
  - **1. Care este problema principală pe care o rezolvă variabilele de condiție în programarea concurrentă?**  
R: Ele permit unui thread să se pună în aşteptare (sleep) până când o anumită condiție devine adevărată, evitând astfel aşteptarea activă (spin-wait) care irosește CPU.
  - **2. De ce este ineficientă soluția în care părintele așteaptă terminarea copilului printr-un spin (while done==0);?**  
R: Pentru că părintele consumă CPU în mod activ, verificând continuu condiția, ceea ce este o risipă de resurse.
  - **3. Cum funcționează pthread\_cond\_wait() în relație cu mutexul primit ca argument?**  
R: Când un thread apelează pthread\_cond\_wait(), mutexul este eliberat atomar și thread-ul este pus la somn. Când thread-ul este trezit printr-un signal(), mutexul este recucerit înainte ca pthread\_cond\_wait() să revină.
  - **4. De ce trebuie întotdeauna folosită o buclă while în jurul apelului la pthread\_cond\_wait()?**  
R: Pentru a verifica din nou condiția după trezire, deoarece pot exista semnalări spurioase (false wakeups) sau alte threaduri pot modifica condiția între timp.
-

- **5. De ce este importantă existența unei variabile de stare (ex: `done`) în sincronizarea prin variabile de condiție?**  
**R:** Pentru a reține starea condiției pe care thread-urile o așteaptă, astfel încât thread-ul care așteaptă să nu rămână blocat dacă semnalul a fost trimis înainte de a intra în așteptare.
- 
- **6. Ce poate merge prost dacă un thread semnalează fără a detine mutexul?**  
**R:** Pot apărea condiții de cursă în care un thread care așteaptă nu este trezit corespunzător, ducând la blocaje sau pierderi de semnal.
- 
- **7. În problema Producer/Consumer, ce rol joacă variabilele condiționale `empty` și `full`?**  
**R:** `empty` este folosită de producători pentru a aștepta spațiu liber în buffer, iar `full` este folosită de consumatori pentru a aștepta date disponibile în buffer.
- 
- **8. Ce problemă apare dacă folosim o singură variabilă condițională pentru ambele operații de așteptare?**  
**R:** Un thread poate trezi un alt thread de același tip, ceea ce poate duce la blocaje deoarece niciun thread de tip opus nu este trezit.
- 
- **9. Care este scopul folosirii unui buffer circular în implementarea Producer/Consumer?**  
**R:** Să permită producătorilor și consumatorilor să folosească un buffer de mai multe poziții, evitând suprascrierea datelor și consumarea în exces, crescând astfel eficiența.
- 
- **10. Ce se întâmplă dacă folosim un `if` în loc de `while` în jurul `pthread_cond_wait()`?**  
**R:** Thread-ul s-ar putea trezi fals și să continue execuția fără ca condiția să fie îndeplinită, ducând la comportamente incorecte și erori de sincronizare.

## • **Rezumat - Covering Conditions și Summary**

- **Problema prezentată:**  
Într-un exemplu simplu de alocator de memorie concurrent, mai multe threaduri pot aștepta eliberarea unei cantități de memorie pentru a-și îndeplini cererea. Când un thread eliberează memorie și trimit un semnal (`signal()`), nu este clar care thread care așteaptă ar trebui să fie trezit. Dacă threadul trezit cere mai multă memorie decât a fost eliberată, acesta va trebui să rămână în așteptare, iar celelalte threaduri rămân blocate, ceea ce duce la blocaj.
- **Soluția:**  
În loc să folosim `pthread_cond_signal()` care trezește un singur thread, folosim `pthread_cond_broadcast()` care trezește toate threadurile care așteaptă. Astfel, toate threadurile care pot fi satisfăcute de memoria eliberată se pot trezi, verifică condiția și continua dacă pot.
- **Dezavantaj:**  
`broadcast()` poate trezi mai multe threaduri inutil, care vor verifica condiția și se vor

întoarce imediat în aşteptare, ceea ce poate cauza un impact negativ asupra performanței.

- **Concluzie generală:**

Variabilele de condiție permit sincronizarea fină a threadurilor, permitându-le să aștepte eficient anumite condiții. Problema „covering conditions” este o situație particulară unde `broadcast()` este o soluție simplă și corectă, chiar dacă nu optimă. Dacă programul tău funcționează doar cu `broadcast()`, probabil ai o problemă în design care trebuie corectată.

---

- **Întrebări și Răspunsuri**

- **1. Ce problemă apare în exemplul alocatorului de memorie când un thread eliberează memorie și folosește `pthread_cond_signal()`?**

**R:** Semnalul poate trezi un thread care cere mai multă memorie decât este disponibilă, iar threadurile care ar putea continua să rămân blocate, ducând la blocaj.

---

- **2. Cum se rezolvă problema de trezire incorectă a threadurilor în exemplul dat?**  
**R:** Înlocuind `pthread_cond_signal()` cu `pthread_cond_broadcast()`, astfel toate threadurile sunt trezite și pot verifica condiția.

---

- **3. Care este principalul dezavantaj al folosirii `pthread_cond_broadcast()`?**

**R:** Poate trezi inutil mai multe threaduri care nu pot continua încă, ceea ce duce la un cost de performanță din cauza re-întreruperii repetitive.

---

- **4. Ce înseamnă termenul „covering condition” folosit de Lampson și Redell?**

**R:** O condiție conservativă care acoperă toate cazurile posibile ce necesită trezirea threadurilor, garantând corectitudinea dar posibil la costul unor treziri inutile.

---

- **5. Ce recomandare se face dacă un program funcționează doar când se folosește `broadcast()` în loc de `signal()`?**

**R:** Probabil există o eroare de sincronizare care trebuie corectată, deoarece ideal semnalul ar trebui să fie suficient.

---

- **6. Care este rolul variabilelor de condiție în sincronizarea threadurilor?**

**R:** Ele permit threadurilor să se pună în aşteptare atunci când o condiție nu este îndeplinită și să fie trezite când condiția devine adevărată, evitând utilizarea inutilă a CPU-ului.

---

- **7. De ce trebuie threadurile să verifice condiția într-o buclă după ce sunt trezite?**

**R:** Pentru că pot exista treziri spurioase (false wakeups) sau starea poate fi schimbată de alt thread înainte ca cel curent să acționeze.

# Semaphores (Semafore)

Semaforele sunt primitive de sincronizare introduse de Edsger Dijkstra, care pot înlocui atât **locks** (încuietori), cât și **condition variables** (variabile de condiție). Ele sunt folosite pentru a controla accesul concurent la resurse și pentru a sincroniza evenimente între thread-uri.

---

## 1. Ce este un semafor?

- Un semafor este un obiect cu o valoare întreagă.
  - Are două operații principale:
    - `sem_wait()` (sau `P()` în terminologia lui Dijkstra) — scade valoarea semaforului, așteptând dacă aceasta devine negativă.
    - `sem_post()` (sau `V()`) — crește valoarea semaforului și trezește un thread care așteaptă, dacă există.
- 

## 2. Inițializarea semaforului

```
C
CopyEdit
sem_t s;
sem_init(&s, 0, 1);
```

- Al doilea argument 0 indică faptul că semaforul este partajat între thread-uri din același proces.
  - Al treilea argument este valoarea inițială a semaforului (ex: 1).
- 

## 3. Semafor ca lock (semafor binar)

- Pentru a folosi semaforul ca un lock, îl inițializăm cu valoarea 1.
- Folosim `sem_wait()` înainte de secțiunea critică (blocare).
- Folosim `sem_post()` după secțiunea critică (deblocare).

Exemplu:

```
C
CopyEdit
sem_t m;
sem_init(&m, 0, 1); // X=1 pentru lock
sem_wait(&m);      // intră în secțiunea critică
```

```
// secțiune critică  
sem_post(&m);           // eliberează lock-ul
```

---

## 4. Comportament multithread

- Dacă un thread a apelat deja `sem_wait()` și nu a apelat încă `sem_post()`, valoarea semaforului este 0.
  - Al doilea thread care încearcă să intre în secțiunea critică cu `sem_wait()` va decremente semaforul la -1 și va aștepta (va fi blocat).
  - Când primul thread face `sem_post()`, valoarea crește la 0 și al doilea thread este trezit.
- 

## 5. Semafor pentru ordonarea execuției (ordering)

- Folosit când un thread vrea să aștepte un eveniment (de exemplu, terminarea unui alt thread).
- Se initializează semaforul cu 0 (adică nu există resurse disponibile inițial).
- Thread-ul care trebuie să aștepte face `sem_wait()` și se blochează dacă semaforul este 0.
- Thread-ul care semnalizează face `sem_post()` când evenimentul are loc.

Exemplu simplu de părinte care așteaptă copilul:

```
c  
CopyEdit  
sem_t s;  
sem_init(&s, 0, 0); // inițial 0  
  
void *child(void *arg) {  
    printf("child\n");  
    sem_post(&s); // semnalizează că a terminat  
    return NULL;  
}  
  
int main() {  
    printf("parent: begin\n");  
    pthread_t c;  
    pthread_create(&c, NULL, child, NULL);  
    sem_wait(&s); // așteaptă copilul  
    printf("parent: end\n");  
    return 0;  
}
```

---

## 6. Problema Producer/Consumer (Buffer limitat)

- Se folosesc două semafoare: `empty` și `full`
  - `empty` indică câte spații libere sunt în buffer (inițial MAX).
  - `full` indică câte elemente pline sunt în buffer (inițial 0).

- Producer:
  - Așteaptă ca `empty` să fie  $> 0$  (`sem_wait(&empty)`)
  - Introduce un element (`put()`)
  - Semnalizează că un slot `full` este acum ocupat (`sem_post(&full)`)
- Consumer:
  - Așteaptă ca `full` să fie  $> 0$  (`sem_wait(&full)`)
  - Ia un element (`get()`)
  - Semnalizează că un slot `empty` a fost eliberat (`sem_post(&empty)`)

Exemplu schelet cod:

```
c
CopyEdit
sem_t empty, full;
sem_init(&empty, 0, MAX); // numărul maxim de spații goale
sem_init(&full, 0, 0);   // bufferul este gol la început

void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        sem_wait(&empty);
        put(i);
        sem_post(&full);
    }
}

void *consumer(void *arg) {
    int tmp = 0;
    while (tmp != -1) {
        sem_wait(&full);
        tmp = get();
        sem_post(&empty);
        printf("%d\n", tmp);
    }
}
```

---

## 7. Regula generală de inițializare a semafoarelor

- Valoarea inițială a semaforului trebuie să reprezinte numărul de resurse disponibile imediat după inițializare.
- Pentru un lock binar: inițial 1 (resursa liberă).
- Pentru sincronizare (ordonare): inițial 0 (resursa nu există încă).
- **Întrebări și răspunsuri despre semafoare**
- **1. Ce este un semafor?**  
**R:** Un semafor este un obiect cu o valoare întreagă care poate fi modificată prin două operații principale: `sem_wait()` (scade valoarea, așteaptă dacă valoarea devine negativă) și `sem_post()` (crește valoarea și trezește un thread așteptând dacă există).
- **2. Ce este un semafor binar?**  
**R:** Un semafor binar este un semafor al cărui scop este să fie folosit ca un lock cu doar două stări: 0 (blocață) și 1 (deblocață). Inițializarea sa se face cu valoarea 1.

- **3. Care este valoarea inițială a unui semafor când îl folosim ca lock (semafor binar)?**  
R: Valoarea inițială trebuie să fie 1, pentru că lock-ul este liber la început.
- **4. Care este valoarea inițială a unui semafor când îl folosim pentru sincronizare/ordonare între thread-uri (de exemplu, un thread așteaptă ca altul să termine)?**  
R: Valoarea inițială trebuie să fie 0, deoarece inițial nu este nimic "dat" înainte ca evenimentul să se întâpte.
- **5. Ce se întâmplă dacă un thread apelează `sem_wait()` pe un semafor al căruia valoare este 0?**  
R: Valoarea semaforului scade la -1 și thread-ul se blochează, așteptând un `sem_post()` care să-l trezească.
- **6. Cum folosim un semafor pentru a aștepta ca un thread copil să termine înainte ca thread-ul părinte să continue?**  
R: Thread-ul părinte apelează `sem_wait()` pe un semafor inițializat cu 0, iar thread-ul copil apelează `sem_post()` când a terminat. Astfel, părintele se blochează până când copilul semnalează terminarea.
- **7. Ce reprezintă valoarea negativă a semaforului?**  
R: Valoarea negativă reprezintă numărul de thread-uri care sunt blocate și așteaptă semnalul (un `sem_post()`).
- **8. Cum se poate folosi un semafor în problema producer-consumer?**  
R: Se folosesc două semafoare: `empty` care indică câte sloturi libere sunt în buffer și este inițializat cu mărimea bufferului, și `full` care indică câte sloturi sunt ocupate și este inițializat cu 0. Producerul apelează `sem_wait(&empty)` înainte să adauge un element și `sem_post(&full)` după, iar consumerul face invers.
- **9. Ce înseamnă că `sem_wait()` și `sem_post()` sunt operații atomice?**  
R: Înseamnă că acestea modifică valoarea semaforului și gestionează blocarea/trezirea thread-urilor într-un mod indivizibil, fără ca alte thread-uri să interfereze în timpul execuției acestor operații.
- **10. Care este scopul folosirii unui mutex suplimentar în problema producer-consumer împreună cu semafoarele?**  
R: Mutex-ul protejează accesul la buffer pentru a preveni condiții de cursă în timp ce se adaugă sau se extrage un element, deoarece semafoarele controlează doar numărul de sloturi libere/ocupate, nu și accesul concurrent.

## Sinteză rapidă

### 1. Reader-Writer Locks (RWLocks)

- Se folosește când vrei să permiti mai mulți cititori simultan, dar doar un singur scriitor exclusiv.
- Cititorii cresc un contor `readers` protejat de un mutex (`lock`).
- Primul cititor blochează semaforul `writelock` pentru a împiedica scriitorii.
- Ultimul cititor care termină eliberează `writelock`.
- Scriitorii așteaptă semaforul `writelock` pentru acces exclusiv.

- Problema: pot apărea situații de **starvation** (cititorii pot înfometea scriitorii).
  - Soluții mai sofisticate există pentru a preveni asta, cum ar fi blocarea cititorilor noi dacă un scriitor aşteaptă.
- 

## 2. Dining Philosophers (Problema filosofilor la masă)

- 5 filozofi care alternează între a gândi și a mâncă.
  - Pentru a mâncă, fiecare are nevoie de 2 furculițe (stânga și dreapta).
  - Dacă fiecare filozof ia mai întâi furculița stângă, poate apărea deadlock (toți aşteaptă furculița dreaptă, dar nimeni nu o eliberează).
  - Soluția simplă: un filozof (de exemplu, filozoful 4) ia furculițele în ordine inversă (dreapta, apoi stânga), evitând ciclul de aşteptare.
- 

## 3. Thread Throttling (Limitarea numărului de thread-uri)

- Problema: dacă multe thread-uri accesează o resursă intensivă (ex. memorie) simultan, poate apărea thrashing.
  - Soluția: folosește un semafor cu valoarea inițială egală cu numărul maxim permis de thread-uri simultane.
  - Fiecare thread face `sem_wait()` înainte de regiunea critică și `sem_post()` după, limitând numărul de thread-uri în regiune.
- 

## 4. Implementarea Semaforelor (Semaphores)

- Se pot implementa semafoare folosind mutex și variabile condiționale.
  - Structura conține:
    - un contor (`value`),
    - un mutex,
    - o variabilă condițională.
  - `wait` face blocare pe mutex, apoi aşteaptă condiția dacă `value <= 0`, scade `value`.
  - `post` crește `value` și semnalează condiția.
  - Notă: această implementare nu menține valoarea semaforului negativă, ci blochează la zero, ca în implementarea Linux.
- 

## Întrebări și răspunsuri

### 1. Ce avantaje oferă un reader-writer lock față de un mutex simplu?

R: Permite acces concurent multiplilor cititori când nu există scriitori, crescând paralelismul față de un mutex care permite doar acces exclusiv.

---

---

**2. Cum previne primul cititor accesul scriitorilor într-un reader-writer lock simplu?**

R: Primul cititor face `sem_wait()` pe semaforul `writelock`, blocând scriitorii până când toți cititorii termină.

---

**3. Care este problema principală cu implementarea simplă a reader-writer locks?**

R: Posibilitatea ca scriitorii să fie înfometăți (starvation) dacă cititorii continuă să vină.

---

**4. Cum se evită deadlock-ul în problema filosofilor la masă?**

R: Modificând ordinea în care cel puțin un filosof ia furculițele (de exemplu, ultimul filosof ia mai întâi furculița dreaptă), se rupe ciclul de așteptare.

---

**5. Ce este thread throttling și cum se realizează?**

R: Este limitarea numărului de thread-uri ce pot executa simultan o secțiune critică, realizată cu un semafor care limitează accesul.

---

**6. Ce elemente sunt necesare pentru a implementa un semafor folosind doar mutex și condiții?**

R: Un mutex pentru protecție, o variabilă condițională pentru a bloca/trezi thread-uri, și un contor pentru valoarea semaforului.

---

**7. Care este diferența între un semafor tradițional și un “Zemaphore” din implementarea dată?**

R: Zemaphore nu permite valoarea negativă a semaforului; când valoarea e 0, thread-urile așteaptă pe condiție, nu scad mai jos de zero.