# Notite examen - Sisteme de Operare

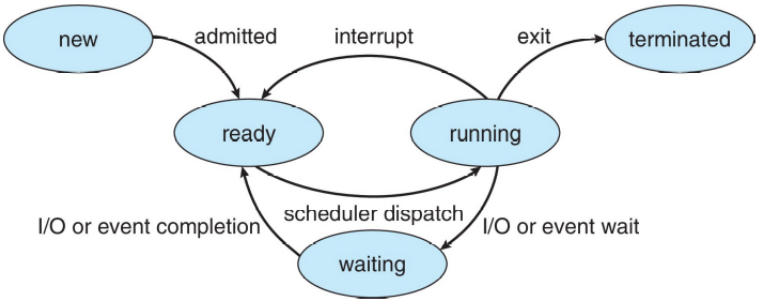
Dinu Florin-Silviu
grupa 231

## Contents

# 1 Processes

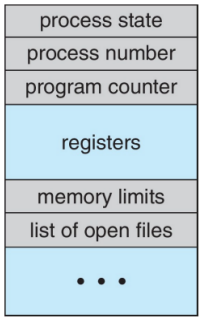
## States

1. New - e creat
2. Running - se executa instructiunile
3. Waiting - asteapta un eveniment
4. Ready - asteapta sa fie asignat unui procesor
5. Terminated - a terminat executia




## Task control block

1. Process state
2. Program counter (location of next instruction)
3. CPU registers (contents of all process-centric registers)
4. CPU scheduling information (priorities, scheduling queue pointers)
5. Memory-management information (memory allocated to the process)
6. Accounting information (CPU used, clock time since start, time limits)
7. I/O status information (I/O devices allocated to the process, list of open files)

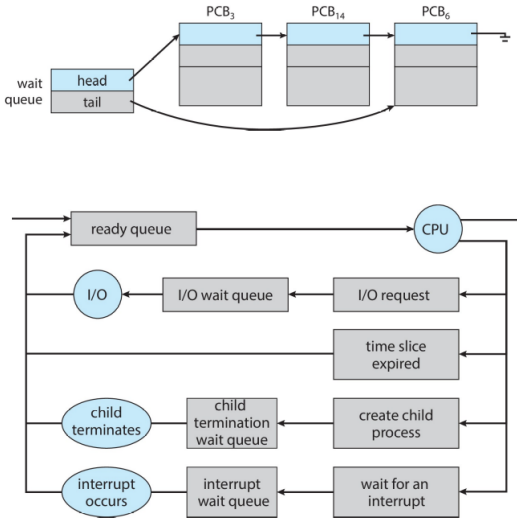



## Process scheduling

**Process scheduler** selects among available processes for next execution on CPU core

**Goal** - maximize CPU use, quickly switch processes onto CPU core

**Scheduling queues** Ready queue and wait queue




## Context switch

Ehen CPU switches to another process, the system must save the state of the old process and load the save state for the new process via a context switch

**Contextul** unui proces este reprezentat in PCB

## Process creation

**Parintii** creaza copii care, la randul lor, pot crea alte procese, formand un arbore de procese

**PID** - process identifier ID

### Partajarea de resurse - optiuni

1. Parintii si copiii partajeaza toate resursele
2. Copiii folosesc o submultime a resurselor parintilor
3. Parintii si copiii nu partajeaza nicio resursa

### Executie - optiuni

1. Parintii si copiii se executa concurent
2. Parintele asteapta sa termine copilul

### Spatiu de adresa

1. Copilul e un duplicat al parintelui
2. Copilul are un program incarcat in el

### Exemple UNIX

1. fork() - syscall pentru crearea de noi procese
2. exec() - syscall dupa fork() ca sa inlocuiasca memory space-ul cu un alt program
3. wait() - parintele asteapta sa termine copilul

## Process termination

**exit()** - procesele executa ca ultima instructiune syscallul exit() care returneaza statusul catre parinte via wait() si resursele sunt dealocate de sistem

**abort()** - parintele poate termina oricand copilul. Unele din motive sunt: copilul a depasit resursele alocate, ceea ce i s-a cerut copilului nu mai este necesar, parintele a apelat exit() si sistemul nu mai lasa copilul sa continue in acest caz (terminarea in cascada)

**Zombie** nu mai are parinte care sa astepte (nu s-a invocat wait())

**Orfan** parintele s-a terminat fara sa invoce wait()

## Interprocess Communication

**Procesele** pot fi independente sau sa coopereze

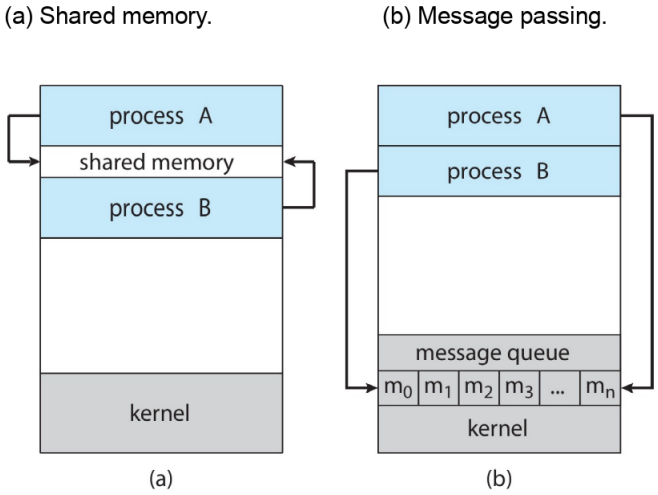
**Cooperarea** inseamna ca procesul poate fi afectate sau afecta alte procese, inclusiv datele partajate

### Motive pentru cooperare

1. Partajarea de informatie
2. Viteza de calcul mai mare
3. Modularitate
4. Convenienta

### IPC - 2 modele

1. memorie partajata
2. pasare de mesaje




## Shared memory

**Producer-consumer** : producatorul produce informatie care e consumata de consumator

**2 variante**

1. unbounded-buffer - nu are limite practice asupra marimii bufferului
   (a) producatorul nu asteapta
   (b) consumatorul asteapta daca nu e buffer de consumat
2. bounded-buffer - toate bufferele au marime fixa
   (a) producatorul trebuie sa astepte daca toate bufferele sunt full
   (b) consumatorul asteapta daca nu e buffer pe care sa-l consume

**Umplerea TUTUROR bufferelor** se face cu un counter care e 0, e incrementat de producator cu fiecare buffer si decrementat de consumator dupa ce il consuma

### Race condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

```
S0: producer execute register1 = counter          {register1 = 5}
S1: producer execute register1 = register1 + 1    {register1 = 6}
S2: consumer execute register2 = counter          {register2 = 5}
S3: consumer execute register2 = register2 - 1    {register2 = 4}
S4: producer execute counter = register1          {counter = 6 }
S5: consumer execute counter = register2          {counter = 4}
```

## Message passing

**2 operatii** : send(message), receive(message)

### Communication link

1. Physical
   (a) Shared memory
   (b) Hardware bus
   (c) Network
2. Logical
   (a) Direct sau indirect
   (b) Sincron sau asincron
   (c) Buffering automat sau explicit

### Comunicarea directa

**Se denumesc explicit** send(P, msg) sau receive(Q, msg)

**Avantaje**

1. Linkurile sunt stabilite automat
2. Linkurile sunt asociate cu exact o pereche de procese care comunica
3. Intre 2 procese este exact 1 link
4. Linkul poate fi unidirectional, dar de obicei, e bidirectional

### Comunicarea indirecta

**Mesajele vin din mailboxes (ports)** Fiecare mailbox are ID unic, procesele pot comunica doar daca partajeaza un mailbox

**Proprietatile linkului de comunicare**

1. Linkurile sunt stabilite doar daca e un mailbox comun
2. Un link poate fi asociat cu mai multe procese
3. Perechile de procese pot avea in comun mai multe linkuri
4. Linkul poate fi unidirectional sau bidirectional

**Operatii**

1. Crearea de mailbox (port)
2. Send and receive
3. Delete

**Primitive** send(A, msg), receive(A, msg)

### Pasarea de mesaje

1. Blocking
   (a) Blocking send
   (b) Blocking receive
2. Non-blocking
   (a) Non-blocking send
   (b) Non-blocking receive
3. Alte combinatii: daca avem send si receive blocking, avem **rendezvous**

### Buffering

**Queue** atasata unui link

**3 implementari:**

1. Zero capcity - fara mesaje pe link. Senderul asteapta pentru receiver (rendezvous)
2. Bounded capacity - lungime finita de n mesaje. Senderul asteapta daca linkul e full
3. Unbounded capacity - lungime infinita. Senderul nu asteapta niciodata

- POSIX Shared Memory
  - Process first creates shared memory segment
    `shm_fd = shm_open(name, O CREAT | O RDWR, 0666);`
  - Also used to open an existing segment
  - Set the size of the object
    `ftruncate(shm_fd, 4096);`
  - Use `mmap()` to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by `mmap()`.

## Mach

### Message based

1. syscallurile sunt mesaje
2. toate taskurile au 2 porturi la creare: kernel si notify
3. mesajele sunt trimise si primite cu mach_msg()
4. portul e creat cu mach_port_allocate()
5. send si receive sunt flexibile, 4 optiuni daca mailboxul e full
   (a) asteapta nedefinit
   (b) asteapta max n ms
   (c) returneaza imediat
   (d) cachuieste un mesaj temporar

## Pipes

### Comunicare intre 2 procese

## Ordinary pipes

### Nu pot fi accesate din afara procesului care le-a creat

1. Comunicare standard in stil producer-consumer
2. Producerul: write-end of the pipe
3. Consumerul: read-end of the pipe
4. Unidirectionale
5. Au nevoie de parent-child

### Named pipes

**Pot fi accesate fara relatie parent-child**

1. Comunicarea e bidirectionoala
2. Nu e nevoie de parent-child
3. Mai multe procese pot folosi acelasi pipe

### Sistemele client-server

**Sockets**

1. endpoint de comunicare
2. IP:PORT
3. <1024 well known
4. 127.0.0.1 loopback

**RPC**

1. Abstractizeaza procedura callurilor dintre procese si sistemele din retea
2. foloseste porturi
3. stubs - client-side proxy pentru procedura actuala din server
4. stubul localizeaza serverul si marshalls parametrii
5. server-side stub primeste mesajul, dezpacheteaza parametrii si face procedura
6. reprezentarea datelor se face prin XDL (External Data Representation)
7. comunicarea are mai multe failure scenarios (mesajele pot fi trimise o SINGURA data sau CEL MULT o data)
8. OS-ul are un rendezvous (sau matchmaker) service ca sa conecteze clientul si serverul

## 2 Threads and Concurrency

**Beneficii**

1. Responsivness - continuarea executiei daca o parte din proces e blocata (UI)
2. Resource Sharing - partajeaza cu procesele mai usor ca shared memory sau message passing
3. Economie - mai ieftin decat crearea proceselor, iar threa switching mai ieftin ca message passing
4. Scalabilitate - procesele pot folosi arhitecturile multicore

**Paralelism** - un sistem poate face mai mult de 1 lucru simultan

**Concurrency** - sustine mai mult de un task care face progres (single processor/core - scheduler providing concurrency)

**Tipuri de paralelism**

1. Data paralelism - distribuie submultimi ale datelor pe mai multe coreuri, aceeasi operatie pentru fiecare
2. Task parallelism - distribuie threadurile pe coreuri, fiecare thread facand o actiune unica

**Legea lui Amdahl**

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

**title** S = portiunea seriala, N = numarul de coreuri

**User vs Kernel Threads**

## Modele multithreading

1. Many-to-One
2. One-to-One
3. Many-to-Many

**Many-to-one**

1. Mai multe threaduri user-level mapate pe un thread de kernel
2. Un thread blocking le face pe toate sa se blocheze
3. Mai multe threaduri pot sa nu ruleze in paralel pe un sistem multicore pentru ca numai unul poate fi in kernel la un moment dat
4. Exemple: Solaris Green Threads, GNU Portable Threads

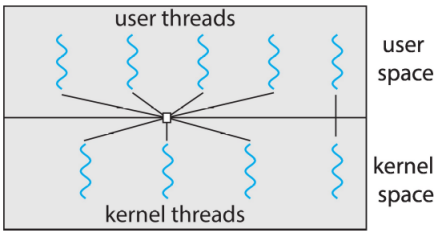
**One-to-One**

1. Fiecare thread user-level mapeaza pe un thread al kernelului
2. Crearea unui thread la user level creaza un thread in kernel
3. Mai multa concurenta decat many-to-one
4. Numarul de threaduri pe proces restrictionat din cauza overheadului

**Many-to-Many**

1. Mai multe user level threads mapate pe mai multe kernel threads
2. Sistemul poate crea un numar suficient de threaduri de kernel
3. Exemplu (necomun): ThreadFiber pe Windows

**Two-level Model** Ca M:M, doar ca un user thread poate fi bound catre un kernel thread



## Pthreads

1. User-level sau kernel-level
2. Un API POSIX standard pentru crearea si sincronizarea threadurilor
3. Specificatie, nu implementare
4. API-ul specifica comportamentul librariei, nu al implementarii
5. Comun in UNIX

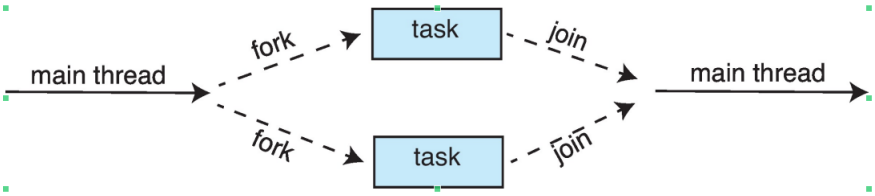
## Implicit threading

1. Thread Pools
2. Fork-Join
3. OpenMP
4. Grand Central Dispatch
5. Intel Threading BUilding Blocks

**Thread Pools** Creaza un numar de threaduri intr-o piscina unde asteapta munca

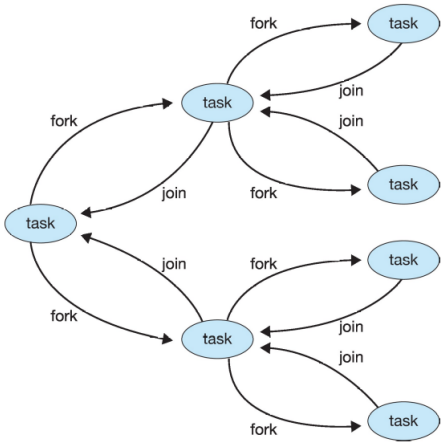
1. De obicei ceva mai rapide sa preia un request cu unul existent decat cu crearea unuia nou
2. Nr de threaduri maxim marimea poolului
3. Separarea taskurilor de mecanismele de creare a taskurilor permit diferite strategii de run (ex: periodical schedule)

**Fork-join Parallelism** Mai mult threaduri sunt forkuite, apoi joinate




**Algoritmul general**

```
Task(problem){
    if (problem is small enough){
        solve directly;
    } else {
        subtask1 = fork (new Task (subset of problem));
        subtask2 = fork (new Task (subset of problem));

        result1 = join(subtask1);
        result2 = join(subtask2);

        return combined results;
    }
}
```




**OpenMP**

1. O multime de directive de compilator si un API pentru C, C++, FROTRAN
2. Suport pentru programarea paralela in mediile cu shared-memory
3. Identifica regiunile paralele (blocheaza codul paralel)
4. Se creaza threaduri pe cat de multe coreuri sunt

**Grand Central Dispatch**

1. Apple technology for macOS and iOS
2. Extensii C, C++ si Objective-C, API si librarie run-time
3. Permite identificarea sectiunilor paralele
4. Gestioneaza majoritatea detaliilor in threading
5. Blocul este intre ^{}
6. Blocurile sunt puse in dispatch queue si sunt asignate unui thread disponibil din pool cand sunt scoase din coada
7. Doua tipuri de dispatch queues
   (a) serial - FIFO, queue per proces, numit main queue
   (b) concurent - FIFO, dar mai multe o data

## Threading issues

**Semantica fork() si exec()**

**Fork()** de obicei duplica doar calling thread (Linux), dar pe alte sisteme, toate threadurile

**Exec()** de obicei inlocuieste procesul care ruleaza inclusiv threadurile sale

**Signal handling** folosit pentru procesarea semnalelor

1. Semnal generat de un anumit eveniment
2. Semnalul este dat unui proces
3. Semnalul este ahndeluit de 1 din cele doua handleluri
   (a) default
   (b) user-defined

**Thread Cancellation** Terminarea unui thread (numit si target thread) inainte de sfarsit

**2 metode:** Asincrona (imediata), Deferred (target threadul verifica periodic daca trebuie cancelat). Tipul implicit e deferred. Pe Linux thread cancellation e obtinut prin semnale

**TLS (Thread Local Storage)** permite fiecarui thread sa aiba copia proprie a datelor si e folositor la thread pooluri (unde nu ai control asupra crearii)

### Scheduler Activations

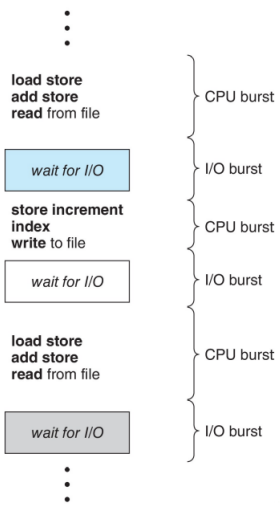
M:M si Two-level au nevoie de comunicare pentru a aloca numarul potrivit de threaduri de Kernel. De obicei se foloseste o structura de date intermediara intre threadurile de user si de kernel LWP (lightweight process). Acesta apare ca un procesor virtual unde procesele pot face scheduleing pentru rularea user threadurilor, fiecare LWP e atasat unui thread de kernel. Scheduler activations au upcalls (un mecanism de comunicare de la kernel la upcall handler)

### Linux threads

1. Li se zice tasks
2. Se fac prin syscallul clone()
3. clone() permite taskurilor copil sa partajeze address space-ul parintelui (procesului)

## 3 CPU Scheduling

**CPU - I/O Burst cycle** un ciclu de cpu execution si I/O wait



## CPU Scheduler

Selecteaza din procesele din queue (ordonata in moduri diferite) si aloca un core de CPU unuia

**Deciziile pot avea loc atunci cand:**

1. Running → waiting
2. Running → ready
3. Waiting → ready
4. Terminates

**Optiuni:** Pentru 1 & 4 **NU** exista, pentru 2 & 3 exista. Asadar pentru 1 & 4 este nonpreemptive (o data alocat, procesul pastreaza CPU-ul pana se termina sau se schimba la waiting), altfel este preemptive.

**Preemptive** poate duce la race conditions

## Dispatcher

**Dispatcher-ul** da controlul CPU-ului procesului selectat de **scheduler**. Acest lucru inseamna: schimbarea de context, schimbarea in user mode, jump la locatia din programul userului pentru a restarta programul

**Dispatch latency** - timpul care ii ia dispatcher-ului sa opreasca un proces, apoi sa ruleze altul

## Scheduling criteria

1. **CPU utilizations** - sa fie cat mai ocupat
2. **Throughput** - nr de procese care isi completeaza executia per unitate de timp
3. **Turnaround time** - timpul de executie al unui proces
4. **Waiting time** - timpul pe care un proces l-a petrecut in ready queue
5. **Response time** - timpul pe care un proces il petrece de cand a facut o cerere pana cand primul raspuns este produs

## FCFS (First-Come, First-Served)

**Convoy effect** - short process behind long process

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |


**Waiting time:** $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

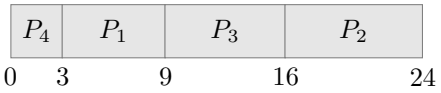
**Average waiting time:** $\frac{(0+24+27)}{3} = 17$

## SJF (Shortest-Job-First)

**SJF e optim** - are media timpurilor de asteptare pentru o multime de procese ca fiind minima

**Shortest-remaining-time-first** este numele versiunii preemptive

| Process | Burst Time |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 8 |
| $P_3$ | 7 |
| $P_4$ | 3 |




**Average waiting time:** $\frac{(3+16+9+0)}{4} = 7$

**Cum determinam lungimea CPU burst?**

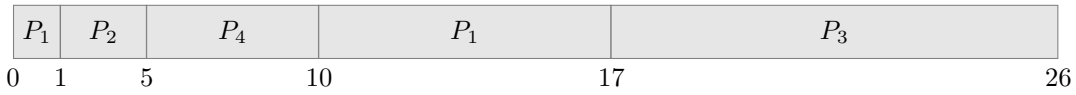
**Estimare:** ar trebui sa fie asemanatoare cu cele anterioare. Poate fi folosita cu exponential averaging

1. $t_n$ = lungimea reala a celui de-al n-ulea CPU burst
2. $\tau_{n+1}$ = valoarea prezisa pentru urmatorul CPU burst
3. $\alpha, 0 \leq \alpha \leq 1$ (de obicei e setat la $\frac{1}{2}$)
4. Definim: $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

## Shortest Remaning Time First

**SJN** - versiunea preemtiva. Cand ajunge in coada de ready, decizia de a-l programa urmatorul este refacuta cu alogirtmul SJN

| Process | Arrival Time | Burst Time |
|---|---|---|
| $P_1$ | 0 | 8 |
| $P_2$ | 1 | 4 |
| $P_3$ | 2 | 9 |
| $P_4$ | 3 | 5 |



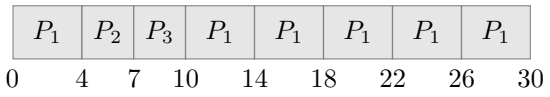

| Process | Completion Time | Turnaround Time (CT-AT) | Waiting time (TAT-BT) |
|---|---|---|---|
| $P_1$ | 17 | 17 | 9 |
| $P_2$ | 5 | 4 | 0 |
| $P_3$ | 26 | 24 | 15 |
| $P_4$ | 10 | 7 | 2 |

**Average waiting time:** $[9 + 0 + 15 + 2]/4 = 26/4 = 6.5$

## Round Robin (RR)

1. Fiecare proces ia o unitate mica de timp pe CPU (**time quantum** $q$), de obicei intre 10-100 ms. Dupa aceast timp, procesul este preempted si adaugat la sfarsitul queueului de ready
2. Daca sunt n procese si time quantum este q, atunci ficare proces ia bucati de $\frac{1}{n}$ din timpul CPU-ului de cel mult q unitati de timp o data. Niciun proces nu asteapta mai mult de $(n-1)q$ unitati de timp
3. Exista un timer care intrerupe fiecare quantum pentru scheduleing pe noul proces
4. Performanta:
   (a) q mare $\approx$ FIFO(FCFS)
   (b) q mic $\approx$ RR

| Process | Burst Time |
|---|---|
| $P_1$ | 24 |
| $P_2$ | 3 |
| $P_3$ | 3 |




**A fost folosit time quantum = 4**

**De obicei $TAT \geq SJF$, dar raspuns mai bun**

**q trebuie sa fie mai mare decat timpul de context switch.** De obicei q intre 10ms si 100ms, iar context switch $\leq 10\mu s$

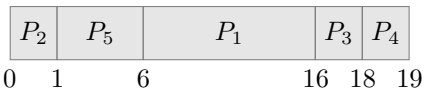
## Priority Scheduling

**Priority number** - integer asociat fiecarui proces (nr mic - prioritate mare)

**SJF** este un priority scheduling unde prioritatea este inversul timpului urmator de CPU burst prezis

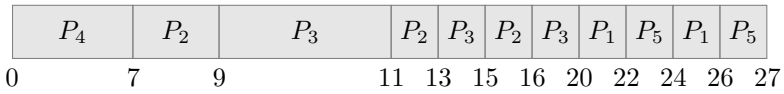
**Problema** Starvation $\Rightarrow$ **Solutie** Aging

| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 10 | 3 |
| $P_2$ | 1 | 1 |
| $P_3$ | 2 | 4 |
| $P_4$ | 1 | 5 |
| $P_5$ | 5 | 2 |



**Priority Scheduling cu Round-Robin (time quantum = 2)**

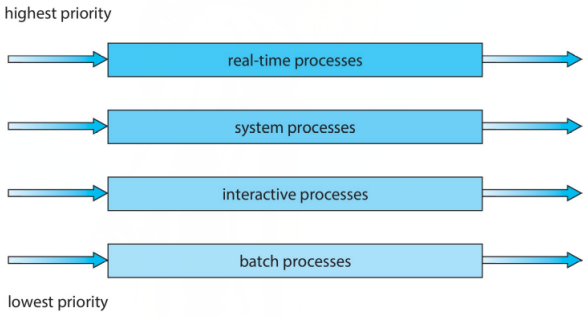
| Process | Burst Time | Priority |
|---|---|---|
| $P_1$ | 4 | 3 |
| $P_2$ | 5 | 2 |
| $P_3$ | 8 | 2 |
| $P_4$ | 7 | 1 |
| $P_5$ | 3 | 3 |




## Multilevel Queue

1. Numarul de queues
2. Algoritmul de scheduling pentru fiecare queue
3. Meoda de determinare a queue-ului in care procesul intra
4. Scheduling intre queues

- Prioritization based upon process type



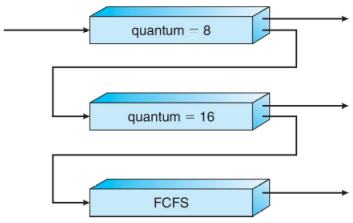

## Multilevel Feedback Queue

Un proces se poate misca prin queues diferite

1. Numarul de cozi
2. Algoritmul de scheduling pentru fiecare coada
3. Metoda de determinare a upgradarii procesului
4. Metoda de determinare a demote procesului
5. Metoda de determinare a cozii in care un proces intra
6. Aging poate fi implementat folosind multilevel feedback queue

## Thread Scheduling

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling
  - A new process enters queue $Q_0$ which is served in RR
    - When it gains CPU, the process receives 8 milliseconds
    - If it does not finish in 8 milliseconds, the process is moved to queue $Q_1$
  - At $Q_1$ job is again served in RR and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue $Q_2$




1. Distinctie intre user-level si kernel-level threads
2. Cand threadurile sunt suportate, ele sunt scheduluite, nu procesele
3. **PCS** - process-contention scope pentru ca schdedulingul e facut per proces de obicei prin prioritate data de programator
4. Kernel thread scheduling este denumit si **SCS** system-contention scope pentru ca intra in competitie cu celelalte threaduri din sistem

**Pthread scheduling** API-ul permite PCS sau SCS, dar pe Linux si macOS doar **pthread_scope_system**

## Multiple-Processor Scheduling

**Poate fi**

1. CPU multicore
2. Multithreaded cores
3. NUMA systems
4. Multiprocesare eterogena

**SMP** - symmetric multiprocessing unde fiecare procesor face self scheduling fie prin common ready queue fie prin cozi private de threaduri pe ficare procesor

**Multithreaded multicore system** - Fiecare core are $> 1$ threaduri hardware. Daca exista memory stall pe un thread, face switch la altul

**CMT** - chip-multithreading (Intel ii zice hyperthreading)

**Sunt 2 nivele:**

1. OS decide ce software thread sa ruleze pe fiecare CPU
2. Fiecare core decide ce hardware thread sa ruleze pe core-ul fizic

**Load balancing** pentru a tine workloadul distribuit uniform se fac **push migrations** (de a lua de la un cpu overloaded la altul) si **pull migrations** (procesoarele idle preiau taskuri de la cele ocupate)

**Processor affinity** - cand un thread ruleaza pe un procesor, cacheul acelui procesor tine memoria accesata de thread fie prin **soft affinity** (OS-ul incearca fara garantii) sau **hard affinity** (permite unui proces sa specifice o multime de procesoare pe care sa ruleze). Load balancingul poate afecta processor affinity.

**NUMA-aware** inseamna ca va asigna memoria apropiata de CPU-ul pe care ruleaza

**Real-Time CPU Scheduling** pe sistemele **soft real-time** face ca taskurile real-time sa aiba prioritate mare, dar nu garanteaza ca vor fi schedeluite, iar pe cele **hard real-time** taskurile trebuie sa fie facute pana la deadline.

**2 tipuri de latenta** afecteaza performata: **interrupt latency**, timpul de la sosirea intreruptului la startul rutinei care serveste interruptul, **dispatch latency**, timpul pe care se scurge de la luarea procesului curent de pe CPU si schimbarea cu altul.

**Priority based-scheduling** pentru real-time scheduling trebuie sa suporte scheduling preemptive si priority-based, dar garanteaza numai soft real-time. Pentru hard real-time trebuie sa aiba abilitatea de a intruni deadlineurile.

**Periodic** - cere CPU la intervale constante. Pnetru timpul de procesare t, deadlineul d si perioada p ($0 \leq t \leq d \leq p$) rata taskului periodic este $\frac{1}{p}$

**Rate monotonic scheduling** - perioadele scurte au prioritate mare, cele lungi, prioritate mica. Se poate intampla ca un proces sa rateze deadlineul.

**EDF (earliest deadline first scheduling)** - prioritatile se asigneaza in functie de deadlineuri (devereme - prioritate mare, tarziu - prioritate mica)

**Proportional share scheduling** sunt T shares pentru toate procesele. O aplicatie primeste N shares ($N < T$) astfel incat sa primeasca $N/T$ din timpul total de procesor.

**POSIX Real-Time Scheduling** are un api cu 2 clase de scheduling

1. SCHED_FIFO - cu strategia FCFS si coada FIFO. Fara time-slicing pentru threaduri cu prioritate egala
2. SCHED_RR - la fel ca prima, dar exista time-slicing pentru threaduri cu prioritate egala

## Linux scheduling

**Pana la 2.5** avea variatii ale algoritmului de scheduling standard din UNIX. De la 2.5 s-a mutat in timp constant O(1).

1. Preemptive, priority based
2. 2 rangeuri de prioritati: time-sharing si real-time
3. real-time intre 0 si 99 si nice value de la 100 la 140
4. Taskul este runable cat timp mai are timp in time slice (active)
5. Daca nu mai are timp (expired) nu mai este runable pana cand celelalte taskuri isi folosesc sliceurile
6. Toate taskurile runable sunt tinute per-CPU in runqueue
   (a) 2 arrayuri de proritate (active, expired)
   (b) taskuri indexate pe prioritate

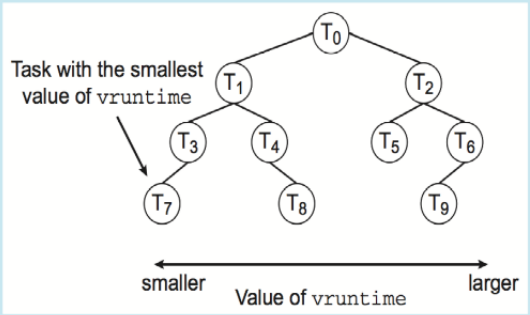
(c) cand nu mai sunt active, arrayurile se schimba

7. A mers bine, dar poor response times pentru procesele interactive

**De la 2.6.23** se foloseste **CFS** (Completely Fair Scheduler) care introduce **clase**:

1. Fiecare are prioritate specifica
2. Schedulerul se uita dupa taskul cu prioritate maxima in clasa cu prioritate maxima
3. In loc de quantum based, e bazat pe proportia din CPU time
4. 2 incluse (altele pot fi adaugate): default, real-time
5. Quantumul e calculat pe nice value de la -20 la +19 si calculeaza target latency (intervalul in care fiecare task trebuie sa ruleze macar 1 data)
6. Tine virtual run time per task (vruntime) si alege taskul cu cel mai mic virtual runtime
7. Tine totul intr-un red-black tree
8. Nice de -20 e prioritate globala de 100 si +19 de 139

The Linux CFS scheduler provides an efficient algorithm for selecting which task to run next. Each runnable task is placed in a red-black tree—a balanced binary search tree whose key is based on the value of `vruntime`. This tree is shown below:




When a task becomes runnable, it is added to the tree. If a task on the tree is not runnable (for example, if it is blocked while waiting for I/O), it is removed. Generally speaking, tasks that have been given less processing time (smaller values of `vruntime`) are toward the left side of the tree, and tasks that have been given more processing time are on the right side. According to the properties of a binary search tree, the leftmost node has the smallest key value, which for the sake of the CFS scheduler means that it is the task with the highest priority. Because the red-black tree is balanced, navigating it to discover the leftmost node will require $O(lgN)$ operations (where $N$ is the number of nodes in the tree). However, for efficiency reasons, the Linux scheduler caches this value in the variable `rb_leftmost`, and thus determining which task to run next requires only retrieving the cached value.

**Linux scheduling** suporta load balancing, dar este si NUMA-aware, grupand multimile de coreuri de CPU care pot fi in balanta intr-un scheduling domain

## Selectarea algoritmului de evaluare

**Deterministic** cu evaluare analitica

**Formula lui Little**

1. n = average queue length
2. W = average waiting time in queue
3. $\lambda$ = average arrival rate into queue
4. $n = \lambda * W$

**Simulari** dar au accuracy limitat

# 4 Synchronization Tools

## Critical section problem

**Procesele** trebuie sa ceara permisiune de intrare in critical section in **entry section**, pot continua cu **exit section**, apoi cu **remainder section**.

**Requirements**

1. Mutula exclusion - daca $P_i$ este in critical section, niciun alt proces nu mai poate executa cod de acolo
2. Progress - daca niciun proces nu executa din critical section si exista procese care cer acces, acest lucru nu poate fi amanat pe termen nedeterminat
3. Bounded waiting - exista o limitare a numarului de dati in care alte procese pot intra in critical section pana ca un proces care a facut cererea de a intra este lasat sa o faca

**Solutia 1.** Pentru 2 procese. Presupunem ca load si store sunt instructiuni machine-language atomice care shareuiesc o variabila **int turn**. Initial turn are valoarea i.

```
while(true){
    while(turn == j);

    /* critical section */
    turn = j;
    /* remainder section */
}
```

**Mutual exclusion** e pastrat, nu si progress requirement sau bounded-waiting requirement.

**Solutia lui Peterson.** 2 procese. La fel cu load si store sunt atomice. Au 2 variabile **int turn; boolean flag[2];**. Turn spune cui ii este randul, flag spune daca un proces e gata sa intre in critical section.

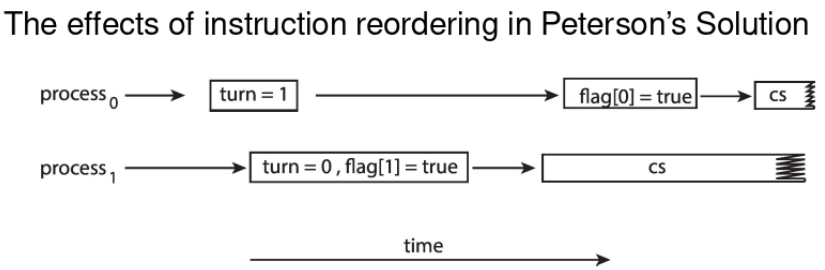
```
while(true){
    flag[i] == true;
    turn = j;
    while(flag[j] && turn == j);

    /* critical section */
    flag[i] = false;
    /* remainder section */
}
```

**Cele 3 CS requirements sunt indeplinite.** Mutual exclusion, progress requirement si bounded-waiting requirement. Totusi, pentru multithreaded poate produce rezultate neasteptate.

- Two threads share the data:
```
boolean flag = false;
int x = 0;
```
- Thread 1 performs
```
while (!flag)
    ;
print x
```
- Thread 2 performs
```
x = 100;
flag = true
```
- What is the expected output?

100

**Reordonarea** celor 2 instructiuni din Threadul 2 se poate intampla pentru ca flag si x sunt independente, asa ca outputul poate fi 0.




**Memory Barrier** e folosit pentru acest caz astfel incat sa nu intre ambele procese in critical section.

## Memory barrier

Garantiile pe care arhitectura calculatoarelor le face pentru programe.

1. Strongly ordered - o modificare a memoriei pe un procesor e imediat vizibila si celorlalte procesoare
2. Weakly ordered - nu e imediat vizibila

**Memory barrier** - o instructiune care foteaza orice modificare in memorie sa fie propagata si celorlalte procesoare.

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

## Synchronization Hardware

1. Uniprocessors - executa codul running fara preemption, dar sunt ineficiente si OS-urile care folosesc asta nu sunt scalabile
2. Hardware instructions
3. Atomic variables

**Hardware instructions** - test-and-set sau compare-and-swap atomice

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

1. Se executa atomic
2. Returneaza valoarea originala a parametrului
3. Seteaza noua valoare a parametrului ca true

```
do {
    while(test_and_set(&lock)); /* do nothing */
    /* cs */
    lock = false;
    /* remainder section */
} while (true);
```

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

1. Executat atomic
2. Returneaza valoarea originala din value

3. Seteaza value cu new_value, numai daca *value == expected

```
while(true){
    while(compare_and_swap(&lock, 0, 1) != 0); /* do nothing */
    /* cs */
    lock = 0;

    /* remainder section */
}
```

**Bounded-waiting cu compare-and-swap**

```
while(true){
    waiting[i] = true;
    key = 1;
    while(waiting[i] && key == 1)
        key = compare_and_swap(&lock, 0 ,1);
    waiting[i] = false;
    /* cs */
    j = (i+1) % n;
    while((j!=i) && !waiting[j])
        j = (j+1) % n;
    if(j == i)
        lock = 0;
    else
        waiting[j] = false;
    /* remainder section */
}
```

**Atomic variables** Presupunem ca sequence e o variabila atomica, iar increment() este o operatie atomica pe sequence. Asadar increment(&sequence) face ca sequence sa fie incrementat fara intrerupere.

```
void increment(atomic_int *v){
    int temp;
    do{
        temp = *v;
    } while(temp!= (compare_and_swap(v, temp, temp+1)));
}
```

**Mutex locks** Sunt ceva mai simple. Protejeaza cs cu acquire() si realease() care sunt atomice (de obice instructiuni hardware atomice cum ar fi compare-and-swap). Foloseste **busy waiting**, de aceea lockul se numeste **spinlock**

```
while(true){
    /* acquire lock */
        /* cs */
    /* release lock */
    /* remainder section */
}
```

**Semaphore** Mai sofisticat decat mutex locks. Semaforul S este integer si poate fi accesat via 2 operatii atomice wait() si signal()

```
wait(S){
    while(S <= 0); // busy wait
    S--;
}
```

```
signal(S){
    S++;
}
```

**Pot fi 2:** counting sempahore (integer peste un domeniu nerestrictionat) sau binary semaphore (intre 0 si 1, asemanator cu un mutex lock). Un counting sempahore poate fi implementat ca un binary semaphore.

**Waiting queue** asociat cu fiecare semafor. Fiecare item are value (integer) si pointer (catre urmatorul item). Se fac 2 operatii: block (procesul care invoca operatia este bagat in coada) si wakeup (scoate din waiting queue procesul si il pune in ready queue). Asa se poate face fara busy waiting.

```
typdef struct{
    int value;
    struct process *list;
} semaphore;

wait(semaphore *S){
    S->value--;
    if(S->value < 0){
        adauga la S->list;
        block();
    }
}

signal(semaphore *S){
    S->value++;
    if(S->value <= 0){
        scoate P din S->list;
        wakeup(P);
    }
}
```

**Monitors** Hihg-level abstraction. Numai 1 procesor poate fi activ cu un monitor la un moment dat

```
monitor monitor-name{
    // shared variable declarations
    procedure P1(...){...}
    procedure P2(...){...}

    procedure Pn(...){...}

    initialization code(...) {...}
}
```

**Implementarea de monitoare cu semafoare.** Variabilele **semaphore mutex; mutex = 1;**. Fiecare procedura P va avea structura:

```
wait(mutex);
        ...
        body of P;
        ...
signal(mutex);
```

**Condition variables.** condition x,y; Sunt 2 operatii: x.wait() si x.signal(). Ultima rezuma procesul care a invocat.

**Folosire** Consideram $P_1$ si $P_2$ care trebuie sa execute $S_1$ si $S_2$ (statementuri) astfel incat $S_1$ sa se intample inainte de $S_2$. Cream un monitor cu doua proceduri $F_1$ invocata de $P_1$ si $F_2$ invocata de $P_2$. O variabila conditionala x initializata cu 0. O variabila booleana **done**.

```
F1:
        S1;
        done=true;
        x.signal();
F2:
        if done == false
            x.wait()
        S2;
```

**Implementare monitoare cu semafoare**

**Variabile:**

```
semaphore mutex; // initial 1
semaphore next; // initial 0
int next_count = 0; // numarul de procese in waiting in monitor
```

**Fiecare functie P va fi**

```
wait(mutex);
        ...
        body of P;
        ...
if(next_count > 0)
        signal(next);
else
        signal(mutex);
```

**Implementare de condition variables**

**Fie x** condition variable avem:

```
semaphore x_sem; // initial 0
int x_count = 0;
```

**x.wait()**

```
x_count++;
if(next_count > 0)
        signal(next);
else
        signal(mutex);
wait(x_sem);
x_count--;
```

**x.signal()**

```
if(x_count > 0){
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```

**Rezumarea proceselor dintr-un monitor** Daca mai multe procese au facut coada in condition variable x, atunci ce se executa dupa x.signal()? FCFS nu e adecvat. Se foloseste **conditional-wait** sub forma x.wait(c). C este un integer (numar de prioritate).

**Single Resource allocation**

```
R.acquire(t);
        ...
        access the resource;
        ...
R.release;
```

**Monitor pentru alocarea single resource**

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }
    void release() {
        busy = false;
        x.signal();
    }
    initialization code() {
        busy = false;
    }
}
```

### Liveness

**Liveness** se refera la faptul ca un sistem trebuie sa obtina progres pe procese. Waiting indefinit este liveness failure

**Deadlock** - 2 sau mai multe procese asteapta nedefinit pentru un eveniment care poate fi cauzat doar de unul dintre cele din coada de waiting

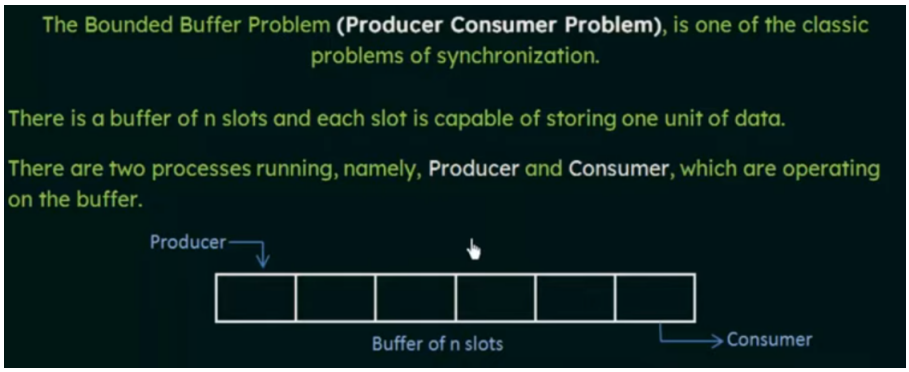
Let $S$ and $Q$ be two semaphores initialized to 1

| $P_0$ | $P_1$ |
|---|---|
| `wait(S);` | `wait(Q);` |
| `wait(Q);` | `wait(S);` |
| ... | ... |
| `signal(S);` | `signal(Q);` |
| `signal(Q);` | `signal(S);` |

**Alte forme de deadlock:** starvation (indefinite blocking - un proces poate sa nu fie scos niciodata din semaphore queue), priority inversion - cand un proes cu lower-priority are un lock necesar unui proces cu higher-priority (rezolvat prin priority-inheritance protocol).

## 5 Synchronization Examples

### Bounded-buffer problem




1. **n** buffere au un hold pe un item
2. semaforul **mutex** este initializat cu 1 - binar, folosit ca sa faca acquire si release pe lock
3. semaforul **full** este intializat cu 0 - cate sloturi sunt folosite in buffer
4. semaforul **empty** este initializat cu valoarea n - nr de sloturi din buffer

#### Structura prcesului producer

```
while (true) {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);  // wait until empty > 0, then decrement empty
    wait(mutex);  // acquire the lock
        ...
    /* add next produced to the buffer */
```

```
        ...
    signal(mutex); // release the lock
    signal(full); // increment full
}
```

**Structura procesului consumer**

```
while (true) {
    wait(full); // wait until full > 0 and decrement full
    wait(mutex); // acquire lock
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex); // release lock
    signal(empty); // increment empty
        ...
    /* consume the item in next consumed */
        ...
    }
```

## Readers-Writers Problem

**Un data set** este shareuit intre mai multe procese concurente. Unele sunt **Readers** care pot doar citi, altii sunt **Writers** care pot citi si scrie. Problema: permite ca mai multi readeri sa citeasca in acelasi timp, iar 1 singur writer poate accesa data in acelasi timp.

**Shared data**

1. Data Set
2. semaforul rw_mutex initializat cu 1 - comun intre readeri si writeri
3. semaforul mutex initializat cu 1 - mutual exclusion cand read_count e actualizat (cand readerii intra sau ies din cs)
4. integer reader_count initializat cu 0 - cate procese citesc din data set

**Writer**

```
while (true) {
    wait(rw_mutex);  // request cs
            ...
    /* writing is performed */
            ...
    signal(rw_mutex); // leaves cs
}
```

**Reader**

```
while (true){
    wait(mutex); // lock pentru read_count
    read_count++; // increase the number of readers by 1
    if (read_count == 1) /* first reader */
```

```
            wait(rw_mutex); // no writer can enter if there is even 1
               reader
            signal(mutex); // other readers can enter while the current
                reader is inside the cs
        ...
    /* reading is performed */
        ...
    wait(mutex);
    read_count--; // a reader wants to leave
    if (read_count == 0) /* last reader */
        signal(rw_mutex); // writers can enter
    signal(mutex); // reader leaves
}
```

**Probleme.** Aceasta rezolvare e numita "first reader-writer problem" pentru ca poate rezulta intr-un writer care sa nu scrie niciodata. "Second reader-writer problem" e o variatie care spune ca "O data ce un writer e gata sa scrie, niciun reader nou nu poate fi lasat sa citeasca". Ambele pot rezulta in starvation. Problema este rezolvata in unele sisteme prin reader-writer locks in kernel.

## Dining-Philosophers Problem

1. N filozifi stau la o masa rotunda cu un castron de orez la mijloc
2. Ei alterneaza intre mancat si gandit
3. Nu interactioneaza cu vecinii
4. Uneori incearca sa ia 2 chopstickuri (cate unul pe rand). Au nevoie de 2 ca sa manance si le dau release cand termina.
5. Shared data: castronul cu orez (data set), semaforul chopstick[n] initializat cu 1

**Solutia cu semafoare.** Filosoful i:

```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % n] ); // N are nevoie de (n-1) si 0

    /* eat for a while */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % n] );

    /* think for a while */

}
```

**Deadlock** Ne asigura ca nu exista 2 vecini care sa manance simultan, dar tot poate crea deadlock. Sa zicem ca toti filosofii vor sa manance in acelasi timp, asta inseamna ca toate elementele din chopstick[n] vor fi 0, dar cand fiecare filosof vrea sa ia chopstickul din dreapta va astepta la nesfarsit.

**Remedii deadlock**

1. Tacamuri pentru n, dar numai n-1 filosofi
2. Un filosof ar trebui sa poata lua ambele chopstickuri numai daca ambele sunt disponibile (trebuie sa le ia in cs)
3. Solutie asimetrica: impar ia mai intai stanga, apoi dreapta, iar par mai intai dreapta, apoi stanga

**Solutia cu monitoare**

**Restrictie:** un filosof poate sa ia chopstickurile, numai daca ambele sunt disponibile

---

```
monitor DiningPhilosophers
{
    /* cele 3 stateuri in care un filosof se poate afla */
    enum {THINKING; HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
            state[i] = HUNGRY;
            test(i); // verifica vecinii daca mananca
            /* Daca nu e eating in urma testului, asteapta semnalul
               altora care vine din putdown */
            if (state[i] != EATING)
                self[i].wait;
    }

    void putdown (int i) {
            /* Isi schimba statusul */
            state[i] = THINKING;
                    // test left and right neighbors
            /* Cheama test pe vecini pentru ca si ei sa poata sa
               manance */
            test((i + 4) % 5); // vecinl din stanga
            test((i + 1) % 5); // vecinul din dreapta
    }
    void test (int i) {
        /* Daca in stanga nu mananca, in dreapta nu mananca si actualul
            e hungry */
        if (
            (state[(i + 4) % 5] != EATING) &&
            (state[i] == HUNGRY) &&
            (state[(i + 1) % 5] != EATING)
        ) {
            /* Mananca */
            state[i] = EATING;

            /* A terminat de mancat si spune ca pot veni si altii */
            self[i].signal();
        }
    }

   initialization_code() {
       for (int i = 0; i < 5; i++)
       state[i] = THINKING;
```

```
     }
}

/* Asa se servesc filosofii */
DiningPhilosophers.pickup(i);

    /** EAT **/

DiningPhilosophers.putdown(i);
```

**Starvation** e posibil, dar nu are deadlock

## Transactional Memory

**Memory transaction** e o secventa de operatii read-write atomica.

**Cu mutex locks**

```
void update(){
    acquire();

    /* modify shared data */

    release();
}
```

**Cu memory transaction**

```
void update(){
    atomic{
        /* modify shared data */
    }
}
```

## OpenMP

O multime de directive de compilator si API care suporta programarea paralela.

```
void update(int value)
{
    /* codul de mai jos e tratat ca un cs si facut atomic */
    #pragma omp critical
    {
        count += value
    }
}
```

## Limbaje functionale

1. Limbajele de PF au o alta paradigma, aceea ca nu mentin stateul
2. Variabilele sunt imutabile si nu isi pot schimba stateul o data ce au avut asignata o valoare
3. Au o abordare mai interesanta a data races

# 6 Main memory

## Protection

Un proces trebuie sa acceseze numai adresele din spatiul lui de adrese. Se pot adauga registri de **base** si **limit**.

**Hardware Address Protection** - CPU-ul trebuie sa verifice daca fiecare acces de memorie generat in user mode este intre base si base + limit.

## Address Binding

Programele de pe disc care sunt gata de a fi aduse in memroie sunt executate ca un **input queue**.

**Reprezentarea adreselor**

1. Codul sursa foloseste de obicei adrese simbolice
2. Codul compilat binduieste catre adrese relocabile
3. Linkerul sau loaderul va bindui adresele relocabile la adrese aboslute
4. Fiecare binding mapeaza un address space catre altul

**Momentele din timp in care se binduiesc instructiunile si data in memorie**

1. Compile time - daca adresa de memorie e cunoscuta a priori, poate fi generat **absolute code**
2. Load time - trebuie sa genereze **relocatable code** daca locatia de memorie nu e cunoscuta la compilare
3. Execution time - bindigul e amanat pana la run time daca procesul poate fi mutat in timpul executiei dintr-un segment de memorie in altul (e nevoie de suport hardware pentru address maps)

## Logical vs Physical Address Space

**Logical address** - generata de CPU, cunoscuta si ca **virtual address**

**Physical address** - adresa vazuta de unitatea de memorie

**Adresele logice si fizice** sunt aceleasi la compile-time si load-time, dar difera la execution-time

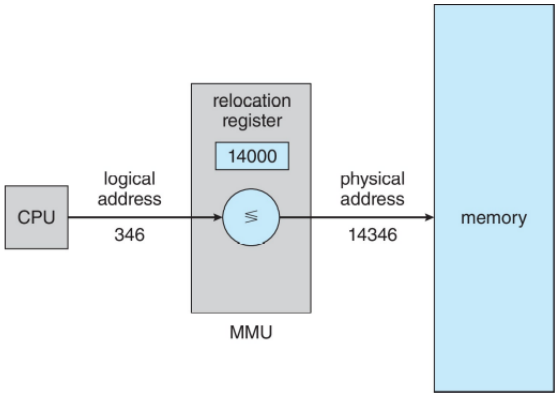
**Logical address space** - multimea tuturor adreselor logice generate de un program

**Physical address space** - multimea tuturor adreselor fizice generate de un program

## MMU (Memory Management Unit)

Dispozitivul hardware care mapeaza adresele virtuale catre cele fizice. Sunt posibile mai multe metode.

**Un caz simplu (generalizarea base-register scheme):** base register e acum numit **relocation register**. Valoarea din relocation register e adaugata catre toate adresele generate de un proces in momentul in care este trimis in memorie. Programul userului lucreaza cu adrese logice si nu le vede pe cele reale fizice. Are loc execution-time bindig cand referinta este facuta la o locatie din memorie, iar adresele logice sunt legate de cele fizice.




### Dynamic Loading

1. Nu tot programul trebuie sa fie in memorie ca sa fie executat
2. Rutinile nu sunt incarcate pana nu sunt apelate
3. Utilizare mai buna a memory-space
4. Toate rutinele sunt tinute pe disc in relocatable load format
5. Folositor cand bucati mari de cod sunt necesare pentru a gestiona cazuri rare
6. Nu e nevoie de suport special din partea OS (se implementeaza prin designul programului, dar OS-ul poate ajuta prin librarii care sa implementeze dynamic loading)

### Dynamic Linking

1. Static linking - librariile si codul programului combinate de loader intr-o imagine binara a programului
2. Dynamic linking - linkingul amanat pana la execution tim
3. Stub - o bucata mica de cod care localizeaza rutina necesara din libraria ce rezida in memorie si apoi se inlocuieste cu adresa rutine si o executa
4. OS verifica daca rutina este in adresa de memorie a programului (daca nu e in address space, o adauga)
5. Dynamic linking e folositoare pentru librarii
6. Sistemul e cunoscut ca **shared libraries**
7. Pentru patchingul librariilor e nevoie de versionare

### Contiguous Allocation

1. Main memory trebuie sa suport atat OS-ul cat si procesele utilizatorului
2. Resurse limitate, trebuie alocate eficient
3. E o metoda incipienta

4. Imparte main memory in 2 partitii
   (a) Resident operating system (low memory cu interrupt vector)
   (b) User processes (high memory, fiecare proces cu o zona continua de memorie)
5. Registrii de relocare protejeaza procesele userilor unul de altul si de schimbarea codului si datelor OS-ului
   (a) Base register contine valoarea celei mai mici adrese fizice
   (b) Limit register contine un range de adrese logice (fiecare adresa logica trebuie sa fie mai mica decat el)
   (c) MMU mapeaza adresele logice dinamic
   (d) Poate suport codul kernelului sa fie transient si kernelul sa isi schimbe dimensiunile

**Variable partitions**

1. Multiprogramming limitat de numarul de partiti
2. Marimi variabile pentru eficienta
3. Hole - bloc de memorie disponibil, gaurile de memorie sunt imprastiate prin intreg sistemul
4. Cand vine un proces, i se acorda loc dintr-o gaura suficient de mare ca sa incapa
5. Exit - paritita se elibereaza, iar partiile libere adiacente se combina
6. OS tine informatii despre
   (a) Partitii alocate
   (b) Partitii libere (hole)

**Dynamic storage-allocation problem**

1. First fit - aloca prima gaura suficient de mare
2. Best fit - aloca cea mai mica gaura suficient de mare (trebuie sa caute in intraga lista daca nu este ordonata dupa marime)
3. Worst fit - aloca cea mai mare gaura (la fel trebuie sa caute in intreaga lista)

**Fragmentare**

**Fragmentare externa** - spatiul total de memorie exista ca sa satisfaca o cerere, dar nu e continuu

**Fragmentare interna** - memoria alocata poate fi un pic mai mare decat cea ceruta, diferenta fiind interna unei partitii, dar nefiind folosita

**Analiza first fit** poate da N blocuri alocate, dar 0.5 N sunt pierdute fragmentarii (1/3 pot fi neutilizabile $\rightarrow$ 50-percent rule)

**Compaction** - metoda de a reduce fragmentarea externa. Se aranjeaza memoria astfel incat toata memoria libera sa fie impreuna intr-un bloc mare. Compactarea e posibila numai daca relocarea e dinamica si se face la execution time. Poate aparea problem I/O (leaga un job in memorie cat timp este implicat in I/O sau fa I/O in buffere de OS)

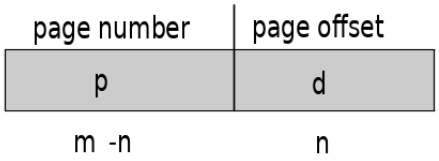
## Paging

Spatiul de adresare fizic al unui proces poate fi necontinuu, iar procesului ii poate fi alocata memorie fizica atunci cand are nevoie sau este disponibila.

1. Se protejeaza impotriva fragmentarii si a bucatilor de memorie de marimi variabile
2. **Frames** - memoria fizica e divizata in blocuri fixe de marimea puterilor lui 2
3. **Pages** - la fel dar pe memoria logica
4. Pentru un program de N pagini, trebuie N frameuri libere
5. Se face un page table care sa traduca adresele logice in fizice
6. Backing store impartit in pagini
7. Inca exista fragmentare interna

**Address translation scheme** Adresa generata de CPU este impartita in:

1. page number (p) - un index intr-un **page table** care contine adresa base a fiecarei pagini in memoria fizica
2. page offset (d) - combinat cu adresa de baza pentru a defini adresa de memorie fizica care este trimisa unitatii de memorie




- For given logical address space $2^m$ and page size $2^n$

**Calculul internal fragmentation**

1. Page size = 2,048 bytes
2. Process size = 72,766 bytes
3. 35 pages + 1,086 bytes
4. Internal fragmentation of 2,048 - 1,086 = 962 bytes
5. Worst case fragmentation = 1 frame - 1 byte
6. On average fragmentation = 1 / 2 frame size
7. So small frame sizes desirable?
8. But each page table entry takes memory to track
9. Page sizes growing over time
   (a) Solaris supports two page sizes - 8 KB and 4 MB

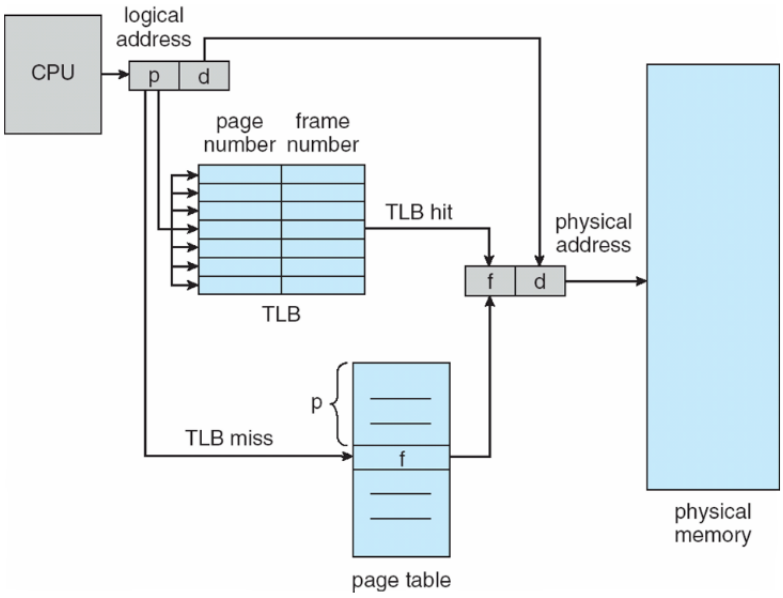
**Implementarea page table**

1. PTBR (page-table base register) - pointeaza catre page table
2. PTLR (page-table length register) - indica marimea paginii tabelului
3. Cele 2 probleme de acces pot fi rezolvate prin **TLBs** (translation-look-aside buffers) numite si **memorie asociativa**

**TLB** - unele stocheaza **ASIDs** (address-space identifiers) in fiecare TLB entry ca sa identifice in mod unic fiecare proces pentru a-i oferi address-space protection.

**Marimea** este de obicei mica (64 - 1024 intrari)

**TLB miss** - valoarea e incarcata in TLB pentru acces rapid data viitoare. Trebuie implementate politici de inlocuire. Unele intrari trebuie sa fie incastrate pentru acces permanent rapid




**Effective Access Time** - hit ratio este procentul in care pagina e gasita in TLB

**Exemplu:** 10ns pentru acces la memorie. Daca nu e in TLB, se fac 2 accesari, adica 20ns. $EAT = 0.80 * 10 + 0.20 * 20 = 12$

**Memory protection** - poate fi implementata asociind un bit de protectie cu fiecare frame ca sa indice daca accesul read-only sau write-only este permis. De fapt, pot fi mai multi biti care sa indice page execute-only etc.

**Valid-invalid bit** - atasat fiecarei intrari din page table, unde valid inseamna ca se afla in logical address space-ul procesului, deci e pagina legala, iar invalid altfel. Sau se foloseste **PTLR** (page-table length register). Orice incalcare determina un trap in kernel

## Shared Pages

**Shared Code** - doar o copie a codului read-only (reentrant) e shareuit intre procese. De asemenea pentru IPC e folositor sa fie shareuite si read-write pages.

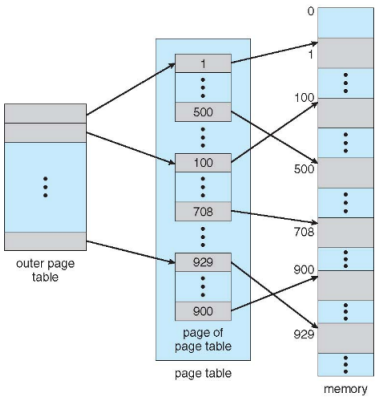
**Private code and data** - fiecare proces are o copie separata a codului si datelor, fiecare pagina pentru private code and data poate aparea oriunde in logical address space

## Structure of the Page Table

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

### Hierarchical Page Tables

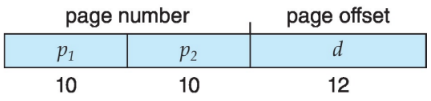
1. Imparte logical address space in mai multe pagini
2. O tehnica simpla e sa avem mai page table pe 2 nivele



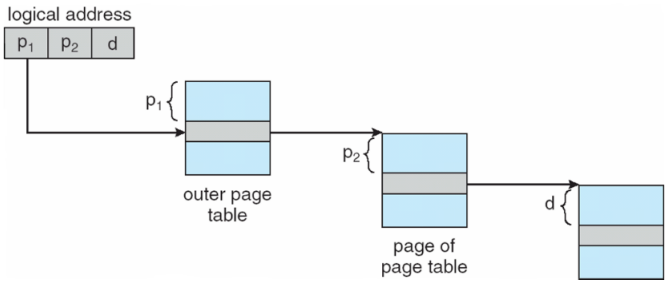

#### 2-level paging

1. O adresa logica pe 32biti are 4k si este divizata intr-un page number cu 20bits si page offset cu 12 bits
2. Din moment ce page tablelul e paged, page number e divizat in: 10bit page number si 10 bit offset

**Forward-mapped page table** $p_1$ este indexul catre outer page table, $p_2$ este displacementul in interiorul paginii pentru inner page table si $d$ este page offsetul

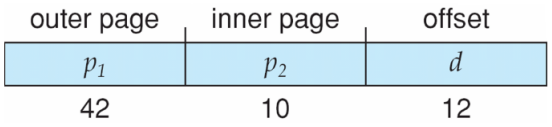


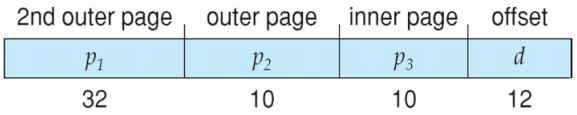

#### Address-Translation scheme




#### 3-level paging scheme pe 64bits

#### Address-Translation scheme



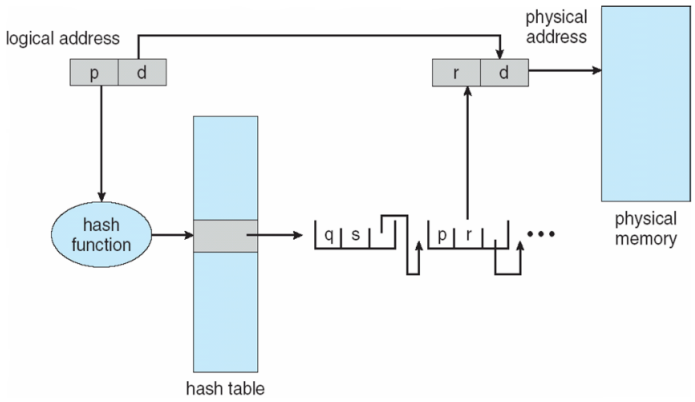




**Hashed page tables** - comune in address spaces > 32 bits. Virtual page number hashed in page table (page tableul contine un lant de lemente hashate catre aceeasi locatie).

**Fiecare element contine:** (1) virtual page number, (2) valoarea page frameului mapat, (3) un pointer la urmatorul element

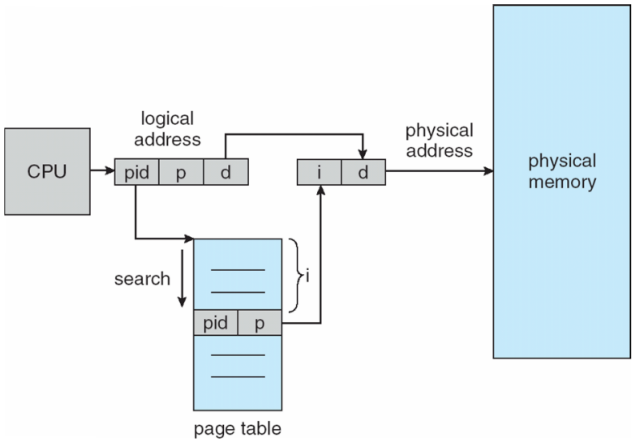
**Virtual page numbers:** comparate in aceasta cautare in lant pentru un match (daca e gasit, se extrage physical frameul)

**CLustered page tables** e o variatie pentru 64-biti. La fel ca hashed, dar fiecare entry se refera la mai multe pagini (16). Foarte folositor pentru sparese address spaces (unde referintele la memorie nu sunt continue si sunt imprastiate)




**Inverted Page Table**

1. In loc ca fiecare proces sa aiba un page table si sa tina cont de toate logical pages posibile, trackuieste toate physical pages
2. O intrare pentru fiecare pagina reala de memorie
3. Intrarea consista in adresa virtuala a paginii stocate in acea locatie reala de memorie, cu informatia despre procesul care o detine
4. Scade memoria pentru fiecare page table, dar creste timpul de cautare in tabel cand are o page reference
5. Se poate folosi hashtable ca sa se limiteze cautarea la una sau cateva page-table enteries




## Swapping

Un proces poate fi **swapped** temporar din memorie catre un backing store si apoi adus in memorie pentru continuarea executiei (memoria fizica total a proceselor poate sa depaseasca memoria fizica)

**Backing store** - disc rapid destul de mare cat sa acomodeze copii ale tuturor imaginiler din memorie pentru toti userii si sa dispuna de acces direct la aceste imagini de memorie

**Roll out, roll in** - un tip de swapping pentru priority-based scheduling unde procesele cu lower-priority sunt swappate ca sa faca loc proceselor higher-priority

**Transfer time** - cea mai mare parte din swap time

**Ready queue** - sistemul mentine aceasta coada cu procesele ready-to-run care au imagini de memorie pe disc

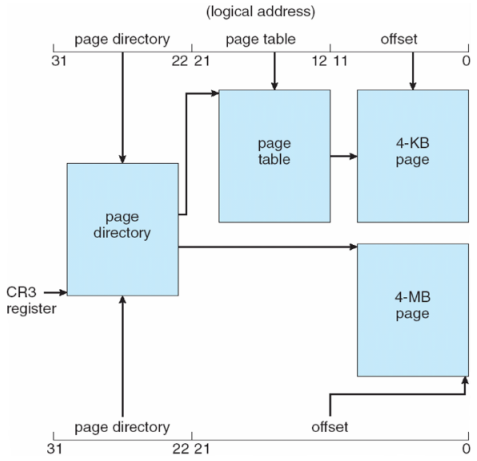
**Context switch time cu swapping** - daca urmatorul proces de pus pe cpu nu e in memorie, trebuie swapat cu altul, asa ca context switch time poate fi foarte mare. Sunt syscalluri de folosire a memoriei cu request_memory() si release_memory()

**Restrictii:** I/O in asteptare nu poate fi swapat pentru ca ar fi in alt proces sau se poate transfera I/O catre kernel space apoi pe dispozitivul I/O (double buffering).

**Strategia pe OS moderne:** swapeaza doar cand memoria libera e foarte mica

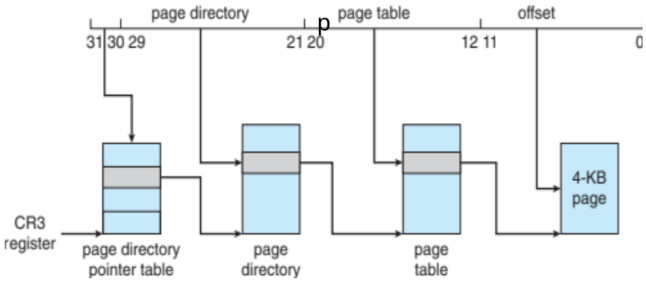
## IA-32 Arch

1. Segementation si segmantation cu paging
2. Fiecare segment e de 4GB, pana la 16K segmente per proces
3. Divizata in 2 partitii
   (a) Prima partitie pana la 8K segmente sunt procesele private (**LDT** - local descriptor table)
   (b) A doua paritie pana la 8K segmente partajate intre toate procesele (**GDT** - global descriptor table)




### IA-32 cu PAE

1. Pagingul are o schema cu 3 nivele
2. Primii 2 biti se refera la un **page directory pointer table**
3. Page-directory si page-table au intrari de 64biti
4. Efectul: cresterea spatiului de adresare la 36 biti - 64gb de memorie fizica

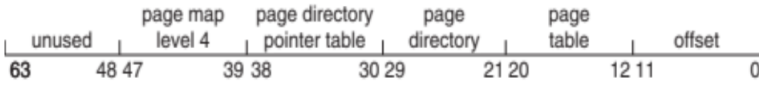



### Intel x86-64

**48 bit addressing** - in practica asta este standardul implementat

**4 levels paging hierarchy**

**PAE** pentru 52 biti




# 7 Virtual memory

Separarea dintre memoria logica a utilizatorului de memoria fizica

1. Doar o parte din program trebuie sa fie in memorie pentru executie
2. Logical address space $\geq$ physical address space

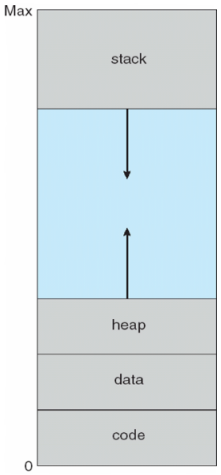
3. Address spaces shareuite intre procese
4. Crearea de procese mai eficienta
5. Mai multe programe pot rula in acelasi timp
6. Mai putin I/O necesar pentru a face load sau swap pe procese

**Implementare prin**

1. Demand paging
2. Demand segmentation

**Virtual-address Space**

1. De obicei spatiul de adresare logic pentru stack incepe la Max logical address si creste ”in jos”, in timp ce heapul creste ”in sus” maximizand folosirea address space-ului, iar spatiul de adresare dintre ele este un hole (nu e nevoie de memorie fizica pana cand heapul sau stackul cresc peste o noua pagina)
2. Foloseste sparese address spaces cu gauri lasate pentru crestere, dll-uri etc
3. Librariile de sistem sunt shareuite prin mapping in virtual address space
4. Memorie partajata prin maparea paginilor read-write in virtual address space
5. Paginile pot fi shareuite in timpul unui fork() pentru a grabi crearea procesului

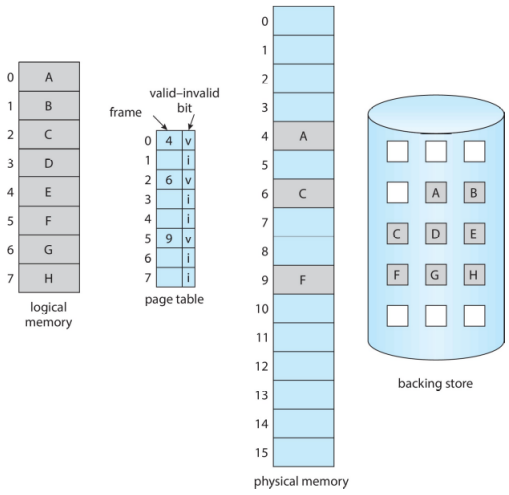



## Demand paging

1. Poate aduce intreg procesul in memorie la load sau doar o pagina cand este nevoie
2. Similar cu paging cu swapping
3. Daca o pagina e necesara → refer-o: referinta invalida → abort, not-in-memory → bring to memory
4. **Lazy swapper** - nu swapeaza o pagina in memorie decat daca e necesara. Swapperul care se ocupa de pagini este **pager**

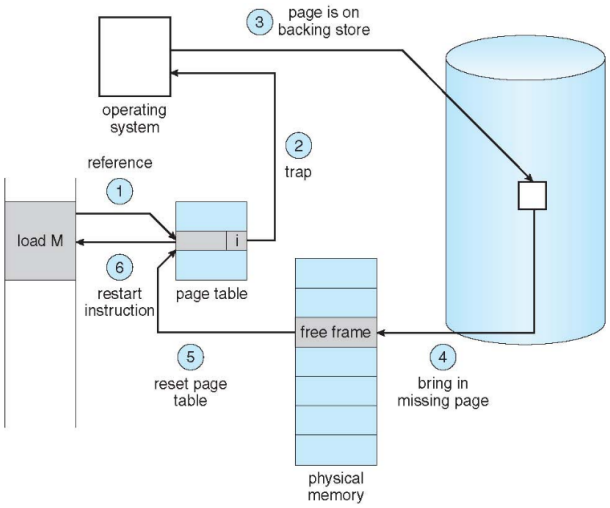
**Concepte de baza** - Pagerul ghiceste ce pagini vor fi folosite pana sa le swapeze

**Valid-invalid bit** - cu fiecare intrare in page table se determina bitul valid-invalid asociat (v → in memorie, i → nu e in memorie). Daca MMU determina ca bitul este i, atunci da page fault




**Pasi in rezolvarea page fault**

1. Daa este o referinta la o pagina, prima referinta va fi trapata in OS: page fault
2. OS se uita in alt tabel ca sa decida daca este referinta invalida (abort) sau doar nu este in memorie
3. Gaseste un frame liber
4. Swapeaza pagina in frame via scheduled disk operation
5. Reseteaza tabelele sa indice ca pagina este acum in memorie si seteaza validation bit la v
6. Reporneste instructiunea care a cauzat page fault




**Aspecte ale demand paging**

1. Caz extrem: porneste procesul fara pagini in memorie. OS seteaza IP catre prima instructiune a procesului si da page fault, apoi pentru celelalte pagini la primul acces (**pure demand paging**)

2. O instructiune poate referinta mai multe pagini. De exemplu fetch si decode a unei instructiuni care aduna 2 numere si stocheaza rezultatul in memorie. Se poate folosi pentru optimizare **locality of reference**
3. Suport hardware necesar pentru demand paging
   (a) Page table cu bit valid/invalid
   (b) Memorie secundara (device de swap cu swap space)
   (c) Instructiunea restart

**Free-frame list** - in caz de page fault, OS-ul trebuie sa aduca pagina dorita in memorie din secondary storage. Se implementeaza free-frame list - un pool de free frames pentru astfel de cerinte

**Zero-fill-on-demand** - continutul frameurilor scos inainte de a fi alocat

**Start up** - toata memoria disponibila e pusa pe un free-frame list

**Stadiile demand paging - cazul cel mai rau**

1. Trap in OS
2. Salveaza user registers si process state
3. Determina daca interruptul a fost page fault
4. Verifica daca acel page reference a fost legal pentru a determina locatia paginii pe disc
5. Citeste de pe disc intr-un free frame
   (a) Asteapta in coada pentru dispozitiv pana cand toate requesturile de read au fost efectuate
   (b) Asteapta dupa latenta dispozitivului si timpul de seek
   (c) Incepe transferul paginii intr-un free frame
6. Cat timp astepti, aloca CPU altui user
7. Primeste un interrupt de la subsistemul disk I/O (completata)
8. Salveaza registrii si process state pentru celelalt utilizator
9. Determina daca interruptul a fost de la disk
10. Corecteaza page table-ul cu alte tabele pentru a arata ca pagina este acum in memorie
11. Asteapta ca CPU-ul sa fie alocat acestui proces din nou
12. Restaureaza user registers, process state si noul page table, apoi rezuma instructiunea intrerupta

**Performanta** : page fault rate $0 \leq p \leq 1$ (0 - fara, 1 - toate referintele sunt page faults)

$EAT = (1-p)*memory\_access+p*(page\_fault\_overhead+swap\_page\_out+swap\_page\_in)$

**Optimizari**

1. Viteza mai mare de swap space I/O fata de sistemul I/O chiar daca e pe acelasi dispozitiv (swap in bucati mai mari pentru ca e mai putin management necesar decat file system)
2. Copiaza intreg procesul in swap space la load time, apoi page in si page out din swap space (versiuni mai vechi de BSD Unix)
3. Cere page din binarul programului de pe disc, dar da discard decat paging out cand eliberezi un frame. Inca e nevoie de swap space: paginile neasociate cu un fisier (stack sau heap) sunt memorie anonima, paginile pot fi modificate in memorie, dar nu scrise imediat pe file system (Solaris si BSD-ul actual)
4. Mobilele nu suporta swapping, dar cer pagina din file system si iau doar read-only pages (cum ar fi codul)

**COW (copy-on-write)** - parintele si copiii shareuiesc initial aceleasi pagini in memorie (daca unul schimba o pagina, doar atunci e copiata).

In general paginile libere sunt alocate dintr-un pool de zero-fill-on-demand pages

**vfork()** e un syscall in care parintele suspenda folosirea de copil a COW in address space-ul parintelui (copilul poate executa exec() si e foarte eficient)

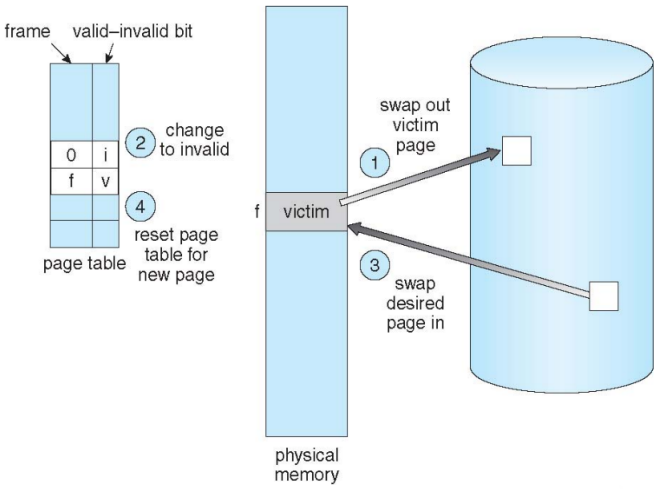
**Page replacement** (daca nu exista free frame)

1. Previne supraalocarea memoriei prin modificarea rutinei de page-fault pentru a include page replacement
2. Foloseste modify (dirty) bit pentru a reduce overheadul transfeurlui de pagini (doar paginile modificate sunt scrise pe disc)
3. Completeaza separarea dintre memoria logica si cea fizica

**Algoritmul**

1. Gaseste locatia paginii necesare pe disc
2. Gaseste free frame (daca nu exista, foloseste replacement pentru a selecta victim frame si scrie victim frame pe disk daca e dirty)
3. Adu pagina dorita intr-un nou free frame, actualizaeaza pagina si frame tables
4. Continua procesul prin restartarea instructiunii care a cauzat trapul

**2 potentiale page transfers pentru page fault → EAT mai mare**




**Algoritmii pentru page si frame replacement**

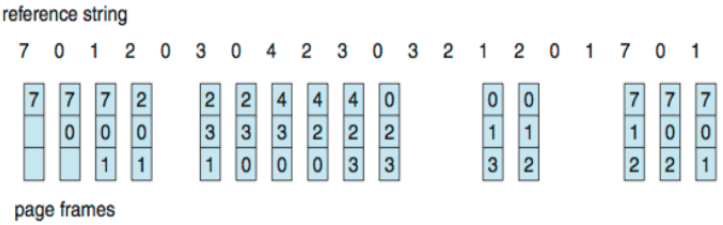
**Frame-allocation** determina cate frameuri sa dea fiecarui proces si ce frameuri sa inlocuiasca

**Page-replacement** Vrea page-fault rate minim atat la primul acces cat si la reaccesare

**FIFO**

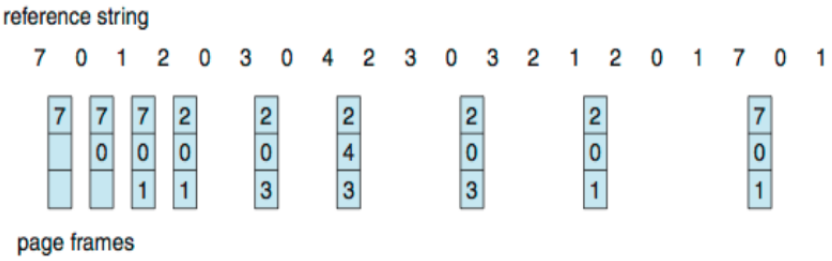
1. Stringul de referinte: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
2. 3 frameuri (3 pagini pot fi in memorie per proces)

**15 page faults**

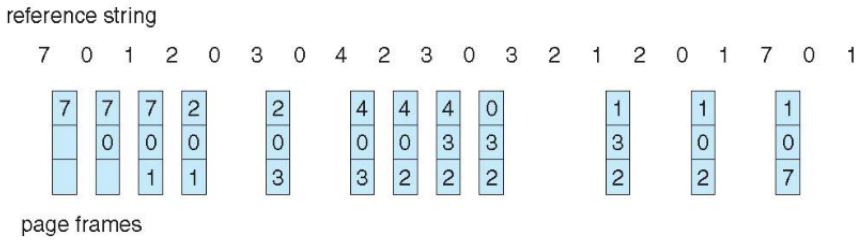



**Anomalia lui Belady** mai multe frameuri pot produce mai multe page faults

**Optimal Algorithm** Replace doar pe pagina care nu va fi folosita mai des in viitor



**LRU (least recently used)** - paginile cele mai vechi




**12 faults** - mai bun ca FIFO, mai rau ca OPT

**Implementare LRU**

1. Counter
   (a) Fiecare pagina are un counter in care punem clock cand intra
   (b) Cand trebuie schimbata o pagina, ne uitam la countere dupa valoarea cea mai mica (e nevoie de search table)
2. Stack
   (a) Pastreaza o stiva cu double link
   (b) Cand o pagina e referntiata: mut-o sus
   (c) Fiecare update e mai scump, dar nu e nevoie de search pentru replacement

**LRU si OPT** sunt cazuri de stack algorithms care nu au anomalia lui Belady

**LRU approximation algorithms** - e nevoie de hardware special si e tot incet

**Reference bit** - fiecare pagina are un bit initial 0, cand e referntiata se face 1, apoi se face replacement pe orice pagina cu bit de referinta 0 (daca exista, daca nu, nu stim ordinea)

**Second-chance algorithm** - in general FIFO plus reference bit pe hardware. Daca reference bit = 0 $\rightarrow$ replace, daca reference bit = 1 $\rightarrow$ seteaza-l ca 0 si las-o in memorie, treci la pagina urmatoare (aceleasi reguli)

**Enhanced second-chance algorithm** - cu 2 biti: reference si modify

1. Se ia perechea ordonata (reference, modify)
2. (0,0) $\rightarrow$ nici folosita, nici modificata - cea mai buna
3. (0,1) $\rightarrow$ nefolosita, dar modificata - trebuie write out inainte de replacement
4. (1,0) $\rightarrow$ folosita recent, dar curata - probabil va fi folosita curand
5. (1,1) $\rightarrow$ Recent folosita si modificata - probabil va fi folosita curand si trebuie write out pana la replacement

**Counting algorithms** - se pastreaza un counter cu numarul de referinte care au fost facute catre o pagina (nu e comun). **LFU** (least frequently used) - replacement pe pagina cu counterul cel mai mic, **MFU** - replacement cu counterul cel mai mare (se considera ca cea cu counterul cel mai mic doar a fost adusa si trebuie folosita)

**Page-buffering algorithms**

1. Tine un pool de free frames
   (a) Cand un frame e necesar, e valabil
   (b) Citeste pagina intr-un free frame si selecteaza victima evacuarii si adaugarii in free pool
   (c) Cand e convenient, evacueaza victima
2. Tine o lista de pagini modificate (posibil) - cand backing store este idle, scrie paginile acolo si seteaza-le ca non-dirty
3. TIne o lista de continut intact de free frames si noteaza ce e in el - daca e referntiata pana sa fie refolosita, nu e enevoie sa iei continutul din nou de pe disc, folosit in special pentru a reduce penalizarea daca victima gresita a fost selectata

## Allocation of frames

2 scheme majore de alocare (cu mai multe variante)

1. Fixed allocation
2. Priority allocation

**Fixed Allocation**

1. Fixa: 100 frameuri, 5 procese, da 20 frameuri per proces (se pot tine unele libere in buffer pool)
2. Proportionala: in functie de marimea procesului (poate fi dinamic, pentru ca procesele isi pot schimba marimea)

**Exemplu**

1. $s_i$ = marimea procesului $p_i$
2. $S = \sum s_i$
3. m = numarul total de frameuri
4. $a_i$ = alocarea pentru $p_i$, $a_i = \frac{s_i}{S} * m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

**Global vs. Local Allocation**

**Global replacement** - procesul selecteaza un replacement frame din multimea tuturor (poate lua frameul altuia) - execution time per proces poate varia semnificativ, dar are throughput mai mare deci e mai comun

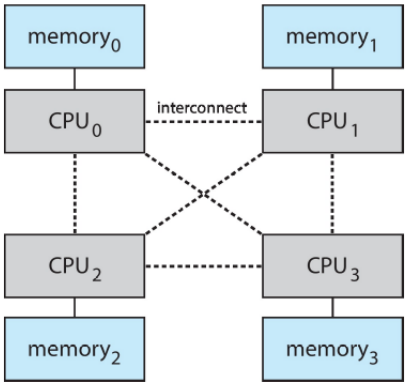
**Local replacement** - fiecare proces selecteaza din multimea frameurilor alocate - mai consistent in performanta per-proces, dar posibil sa subutilizeze memoria

## Reclaiming pages

1. Strategie de a implementa global page-replacement policy
2. Toate cererile din memorie sunt satisfacute din free-frame list, nu se asteapta ca lista sa ajunga la 0 pana incepem sa selectam pagini pentru replacement
3. Replacement trigger cand lista e mai mica de un prag
4. Strategia incearca sa faca posibil spatiu suficient de memorie pentru noi cereri in orice moment

## NUMA

Viteza de acces la memorie variaza. Performanta optima cand se aloca memorie "aproape" de CPU-ul in care threadul e scheduled. Solaris a introdus **lgroups** - structura care trackuieste CPU/Memory low latency groups si cand e posibil, pune toate threadurile unui proces si aloca toata memoria acelui proces intr-un lgroup




## Thrashing

Daca un proces nu are "suficiente" pagini, page-fault e foarte mare

1. Page fault ca sa ia pagina
2. Replacement pe frameul existent
3. E nevoie de replacement frame inapoi rapid
4. Duce la
   (a) CPU utilizat putin
   (b) OS crede ca trebuie sa creasca gradul de multiprogramming
   (c) Inca un proces este adaugat in sistem

**Locality** - o multime de pagini folosite impreuna. Modelul localitatii spune ca atunci cand un proces se executa, se muta dintr-o localitate in alta (unele pot sa se suprapuna). De exemplu, cand o functia este apelata, se defineste o noua localitate, iar cand iese, atunci si procesul iese din acea localitate

**Working-Set Model** - daca alocam suficiente frameuri pentru a acomoda localitatea curenta, va da page fault doar cand se muta intr-o alta, dar daca alocam mai putine decat localitatea curenta, procesul are sanse sa faca thrashing.

**D** - total demand pentru frameuri si WSS este working setul pentru procesul i. $D = \sum WSS_i$

1. $D > m$ (total demand > numarul de frameuri), atunci va fi trashing pentru ca un proces nu poate avea suficiente frameuri
2. $D \leq m$, nu va fi trashing

**Page-fault frequency** - abordare mai directa decat WSS. Se stabileste o rata acceptabila de PFF si o politica de replacement locala. Daca rata e joasa, procesele pierd frame, daca e prea mare, primesc frame

## Allocating Kernel Memory

Tratata diferit de user memory si alocata de obice intr-un pool de free-memory.

**Buddy System** - aloca memorie dintr-un segment cu marime fixa consistand in pagini fizice continue. Se aloca folosind **power-of-2 allocator**.

**Avantaj** - coalesce rapid din bucatile nefolosite intr-o bucata mai mare

**Dezavantaj** - fragmentare

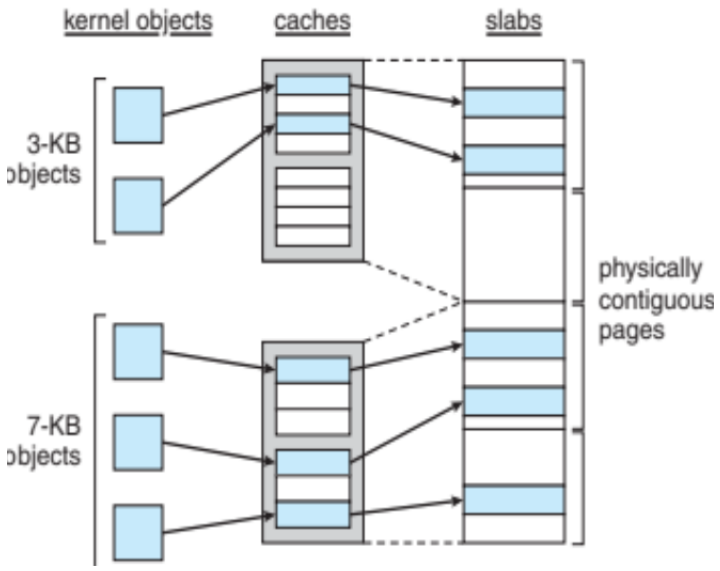
**Slab Allocator** - strategie alternanta

**Slab** e una sau mai multe pagini fizice continue

**Cache** contine mai multe slaburi

1. Un cache pentru o structura de date unica a kernelului
2. Fiecare cache e umplut cu obiecte
3. Cand e creat e marcat cu obiecte free
4. Cand sunt stocate structurile de date, obiectele sunt marcate ca used
5. Daca slabul e full cu obiecte, urmatorul obiect e alocat unui slab vid (daca nu e niciunul vid, se aloca altul)
6. Avantaj: fara fragmentare si stasiface accesul rapid la memorie

**Dupa Linux 2.2** avem SLOB (simple list of blocks - tine 3 obiecte pentru obiecte mici, medii si mari) si SLUB (salb optimizat pentru performanta, pentru ca nu are cozi per-CPU, iar metadata e stocata in structura paginii)




## Prepaging

Pentru a reduce numarul mare de page faults care se intampla la startupul unui porces. Daca sunt nefolosite, I/O si memoria au fost risipite.

**Presupunem** s pagini prepaged si $\alpha$ procentul de pagini folosite. Atunci $s * \alpha$ este costul de a salva page faults si $s * (1 - \alpha)$ numarul de pagini nenecesare. Asadar, cand $\alpha$ e aproape de zero $\rightarrow$ apar pierderi din prepaging

## Page Size

**Consideratii necesare**

1. fragmentare
2. page table size
3. resolution
4. I/O overhead
5. numarul de page faults
6. localitatea
7. marimea TLB si eficienta sa
8. Mereu puteri ale lui 2
9. In medie, cresc in timp

## TLB Reach

Cantitatea de memorie accesibila din TLB

$$TLB\ Reach = (TLB\ Size) * (Page\ Size)$$

1. De obicei intreg wss-ul unui proces e stocat in TLB, atlfel are multe page faults
2. Creste page size poate duce la cresterea fragmentarii
3. Mai multe page sizes face posibil ca aplicatiile sa ceara page sizes mai mari fara sa creasca fragmentarea

### Program structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

### I/O Interlock

Uneori paginile trebuie sa fie blocate in memorie. De exemplu la copierea fisierelor dintr-un dispozitiv, paginile trebuie sa fie blocakte (locked) de a fi evacuate de replacement. Se poate face pinning paginilor ce trebuie sa fie blocate in memorie.

## 8 File system interface

Spatiul logic de adresare continuu.

### Tipuri

1. Data
   (a) Numeric
   (b) Character
   (c) Binary
2. Program

### Atribute

1. Nume
2. Identifier
3. Type
4. Location
5. Size
6. Protection
7. Time, date, user identification
8. Poate include atribute extinse: checksum

## Directory Structure

Colectie de noduri care contin informatii despre toate fisierele. Atat structura de directoare cat si fisierele rezida pe disc

## Operatii

1. Create
2. Write (scrie la locatia pointerului)
3. Read (citeste la locatia pointerului)
4. Reposition within a file (seek)
5. Delete
6. Truncate
7. Open($F_i$) - cauta in directory structure intrarea $F_i$ si muta-i continutul in memorie
8. Close($F_i$) - muta continutul intrarii $F_i$ din memorie in structura de directoare de pe disc

## Open Files

**Open-file table** - tracking pe fisierele deschise

**File pointer** - pointer la ultima locatie de read/write, per proces care are fisierul deschis

**File-open count** - counter pe numarul de deschidere a fisierului (pentru a permite stergerea datelor din open-file table cand ultimul proces o inchide)

**DIsk location** - cache de informatii de acces ale datelor

**Access rights** - modul de acces al informatiei per proces

## File Locking

Folosit de unele OS si FS. Similar cu reader-writer locks.

**Shared lock** - ca un reader-lock: mai multe procese pot folosi concurent

**Exclusive lock** - ca un writer lock

**Mediaza accesul la un fisier**

1. Mandatory - accesul este respins in functie de lock-urile tinute si cele cerute
2. Advisory - procesul poate vedea statusul lockurilor si decide ce sa faca

## Logical records

Un fisier e logical records de marime fixa

## Sequential access

1. Read next
2. Write next
3. Reset
4. no read after last write (rewrite)

## Direct Access

1. read n
2. write n
3. position to n
   (a) read next
   (b) write next
   (c) rewrite n

Numerele de block relative permit OS sa decida unde sa puna fisierul.

## Alte metode de acces

De obicei au un **index** pentru fisier pe care il tin in memorie ca sa determine rapid unde sunt datele pe care sa opereze. Daca indexul e prea mare, se creaza un in-memory index care-l indexeaza pe cel de pe disc

## Disk structure

**Partitii** - discul poate fi subdivizat in partitii

**RAID** - discul sau partitiile pot fi protejate de failure

**Folosirea** raw sau formatata cu un FS

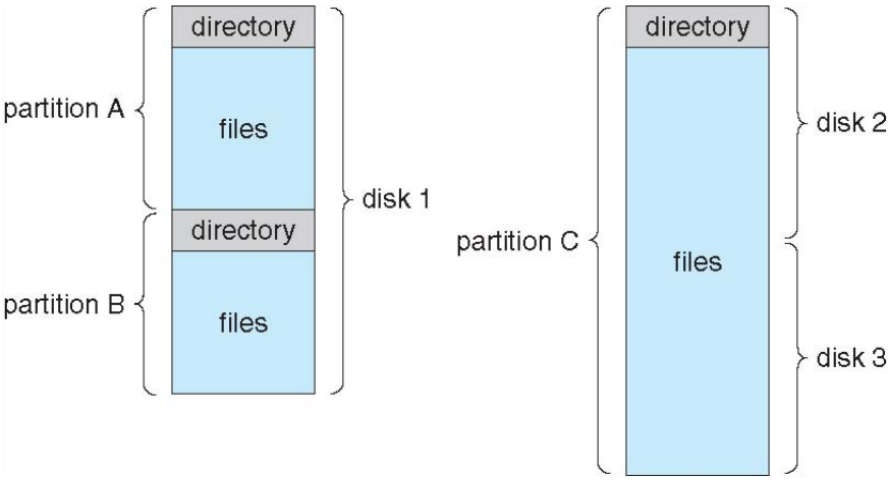
**Sinonime pentru partitii:** minidisks, slices

**Volume** - entitatea care contine FS

**Tracking pe FS** - volumul trackuieste informatiile FS-ului in device directory sau volume table of contents

**Special-purpose file systems** - folosite impreuna cu general-purpose file systems

## Program structure




## Types of File Systems

### Solaris

1. tmpfs - memory based FS volatil pentru I/O rapid si temporar
2. objfs - interfata la memoria kernelului pentru a lua kernel symbols pentru debugging
3. ctfs - contract file system pentru daemoni
4. lofs - loopback fs care permite ca un FS sa fie accesat in locul altuia
5. procfs - interfata a kernelului catre structura de procese
6. ufs, zfs - general purpose file systems

## Operatii pe director

1. Search file
2. create file
3. delete file
4. list a directory
5. rename a file
6. traverse the FS

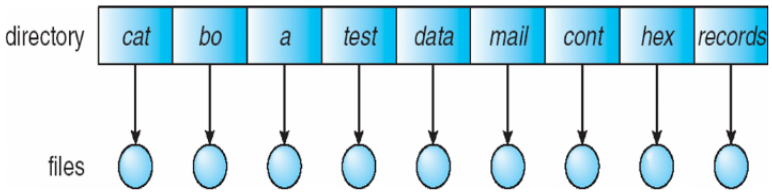
## Directory Organization

### Pentru a obtine

1. Eficienta (localizarea rapida a unui fisier)
2. Denumire
   (a) 2 utilizatori pot avea acelasi nume pentru fisiere diferite
   (b) Acelasi fisier poate avea mai multe nume
3. Grupare

## Single-level directory

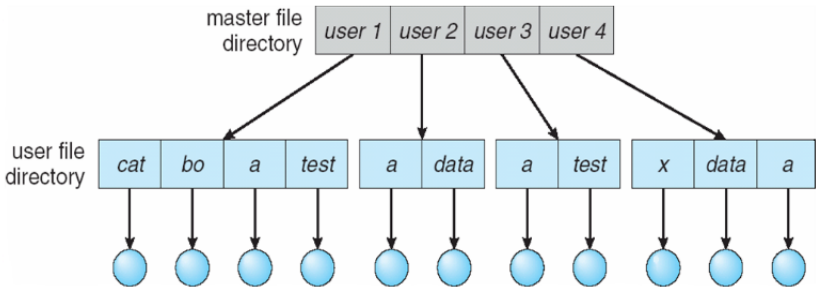
Acelasi director pentru toti userii



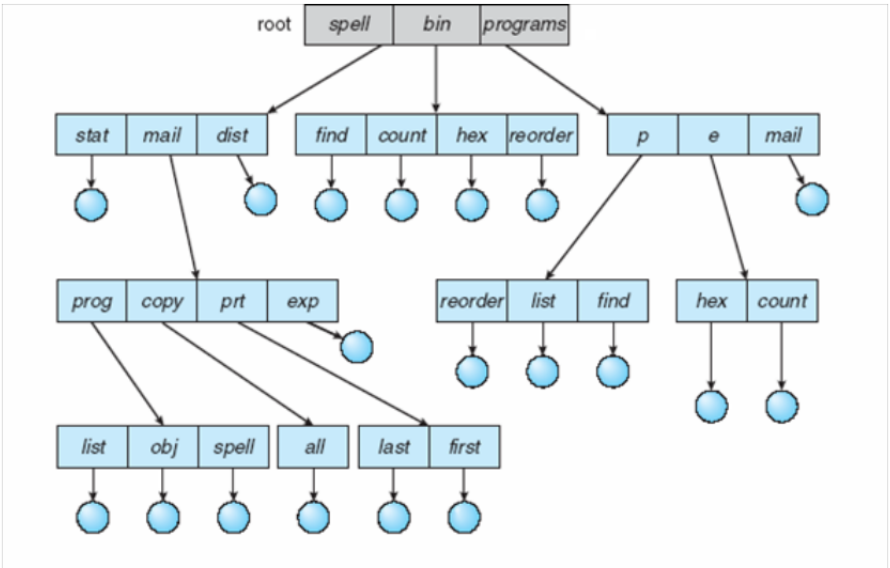

## 2-level directory

Fiecare user cu director separat

1. Poate avea fisiere cu nume identice pentru utilizatori diferiti
2. Searching eficient
3. Nu se pot grupa

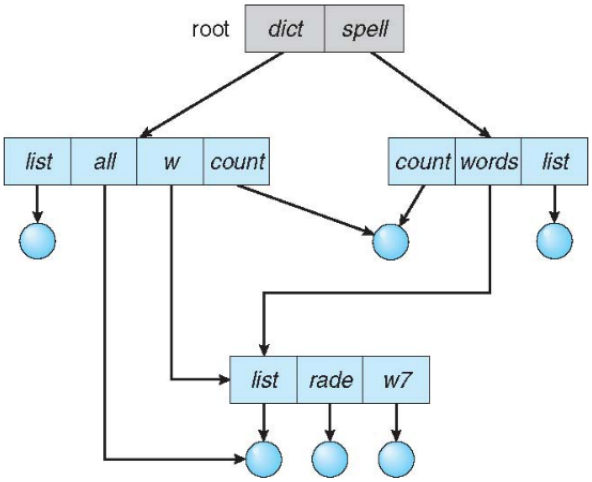



## Tree-structured Directories



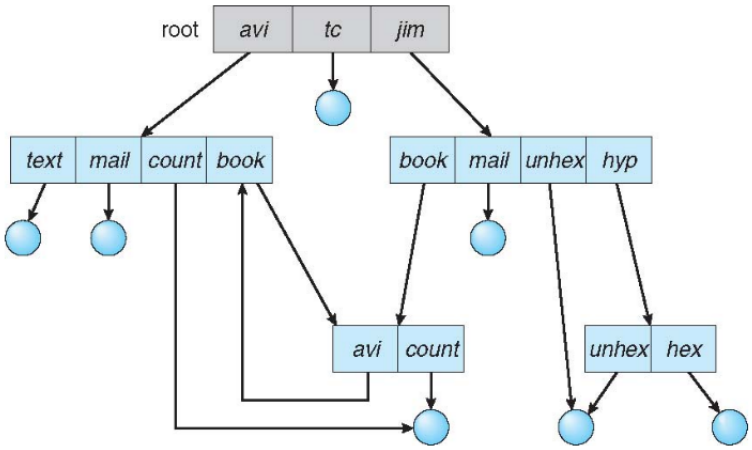

## Acyclic-Graph Directories

Au subdirectoare si fisiere shared




1. Aliasing (2 nume diferite)
2. Daca dict sterge w/list *rightarrow* dangling pointer
   (a) Se pot pune backpointeri ca sa stergem toti pointerii
   (b) Backpointeri cu daisy chain organization
   (c) Entry-hold-count
3. Tip de director nou
   (a) link - un alt nume catre un fisier existent
   (b) resolve the link - urmareste pointerul pentru a localiza fisierul

## General Graph Directory




**Problema ciclurilor:** link catre fisiere, nu subdirectoare; garbage collection; la fiecare link foloseste un algoritm de detectie a ciclurilor pentru a vedea daca este ok

## File protection

Ownerul trebuie s controleze ce poate fi facut si de cine poate fi facut

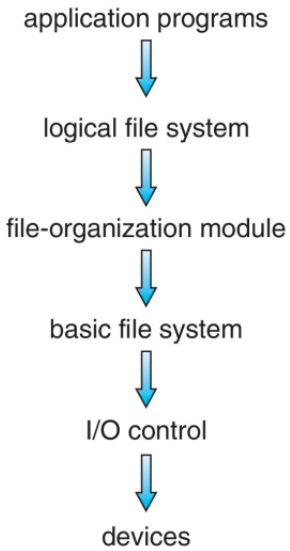
**Tipuri de acces**

1. read
2. write
3. execute
4. append
5. delete
6. list

# 9 File system implementation

## FS Structure

1. Rezida pe secondary storage (disks)
2. e o interfata a utilizatorului catre sotrage, mapand logical catre physical
3. acces eficient si convenient la disc, permitand ca datele sa fie stocate, locate si scoase usor
4. Discul pune la dispozitie in-place rewrite si random access (de obicei transferurile sunt in blocuri de sectoare - de obicei 512bytes)
5. **FCB** (file control block) - structura de stocare care consista in informatii despre un fisier
6. Device driverul controleaza dipsozitivul fizic
7. FS organizat in layere




## FS Layers

**Device drivers** gestioneaza dispozitivele I/O la layerul de control I/O

**Basic file system** - primeste o comanda si o traduce driverului

**Gestioneaza** buffere si cacheuri (bufferele tin datele in tranzit, cacheurile tin datele cele mai des folosite)

**File organization module** - intelege fisierele, adresele logice si blocurile fizice. Traduce blocuri logice in blocuri fizice. Gestioneaza spatiul liber si alocarea pe disc

**Logical file system** - gestioneaza metadatele. Traduce numele fisierului in file number, file handle si locatie prin gestionarea file control blocks (inodes in UNIX). Face managementul directoarelor si protectia.

**Avantaj** - reduce complexitatea si redundanta

**Dezavantaj** - are overhead

**Layerele logice** pot fi implementate de catre designerul de OS

## FS Operations

**Boot control block** - contine informatiile necesare sistemului sa booteze OS din acel volum (necesar daca contine OS, de obicei e primul block al volumului)

**Volume control block** - superblock, master file table - contine detaliile volumului

**DIrectory structure** - nume si numere inode, MFT (master file table)

## FCB (File Control Block)

**OS-ul** mentine FCB per fisier

## In-Memory FS Structures

**Mount table** tine toate system mounts, mount points si FS types

**System-wide open-file table** o copie a FCB pentru fiecare fisier si alte informatii

**Per-process open-file table** contine pointer la intrari din system-wide open-file table, dar si alte informatii

## Directory implementation

**Linear list** de nume de fisiere cu pointeri catre blocurile de date.

1. Simplu de programat
2. Costisitor ca executie (timp de cautare liniar, pot fi ordonate alfabetic cu o lista inlantuita sau cu un B+ tree)

**Hash table** - lista liniara cu strucura datelor hashed

1. Timp mai mic de cautare
2. Coliziuni - daca 2 nume de fisiere au acelasi hash catre aceeasi locatie
3. BUn doar daca intrarile sunt fixe sau se foloseste chained-overflow method

## Allocation Method

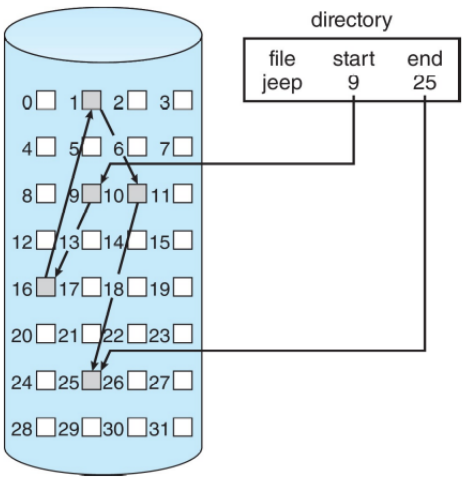
**Alocare continua** - fiecare fisier ocupa o multime continua de blocuri

1. Simplu - trebui doar locatia de start si lungimea
2. Probleme
   (a) Gasirea spatiului pe disc pentru un fisier
   (b) Sa stii marimea exacta a unui fisier
   (c) Fragmentare externa (e nevoie de compaction off-line care presupune downtime sau on-line)

**Extent-based systems** - aloca blocuri de disc in extents. Extent e un bloc continuu de disc. Un fisier poate avea unul sau mai multe extenturi. Ex: Veritas File System

**Linked allocation** - fiecare fisier e linkuita unei liste de blocuri si se termina la pointerul nul

1. Nu exista fragmentare externa
2. Fiecare bloc contine pointer catre urmatorul
3. Fara compaction, external fragmentation
4. Cand e nevoie de un bloc, se cheama free space management system
5. Se poate imbunatati eficienta prin blocks clustering in grupuri, dar creste fragmentarea interna
6. Unreliable
7. Cautarea unui bloc poate sa ia multe seekuri de I/O




**FAT** - inceputul volumului are un tabel indexat dupa block number. E ca o linked list, dar mai rapid si cacheable. Alocarea unui bloc nou e simpla.

**Indexed Allocation Method** - fiecare fisier are propriul bloc de indecsi cu pointeri catre propriile data blocks

**Fisiere mici**

1. E nevoie de un index table
2. Acces random
3. Acces dinamic fara fragmentare externa, dar are overhedului blocului de indecsi
4. Maparea din logic in fizic intr-un fisier de maximum 256KB si un bloc de 512bytes necesita doar 1 bloc pe tabel de indecsi

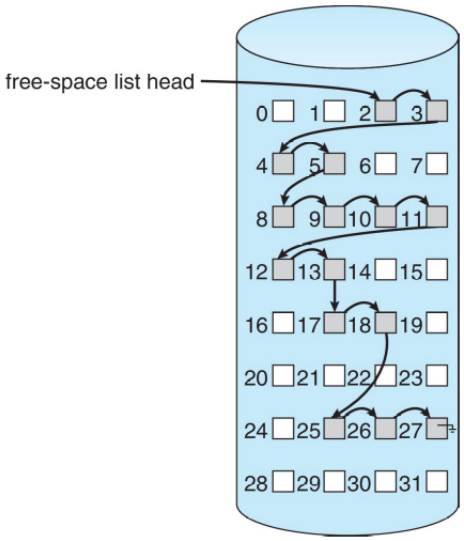
**Fisiere mari** - maparea se face cu linked scheme (link blocks of index table) si multi-level indexing

## Free space management

**Bit vector/bit map** de n blocuri

**Se calculeaza** numarul de bits pe word * numarul de 0-valued words + offsetul primului bit

- Linked list (free list)
  - Cannot get contiguous space easily
  - No waste. Linked Free Space List on Disk of space
  - No need to traverse the entire list (if # free blocks recorded)




**Grouping** - se modifica linked list sa stocheze adresa urmatoarelor n-1 blocuri libere in primul bloc liber, plus un pointer catre primul bloc care contine free-block-pointers

**Continuing** - pentru ca spatiul este frecvent folosit in mod continuu si eliberat cu alocare continua, extents sau clustering: se tine adresa primului bloc liber si numarul de blocuri care il urmeaza, iar lista spatiului liber are intrari care contin adrese si counts

**Space maps**

1. FOlosite in ZFS
2. considera metadatele I/O pe FS foarte mari - structuri mari ca bit maps nu pot intra in memorie ceea ce duce la multe I/O
3. divide spatiul in unitati metslabs si gestioneaza metslabs
4. fiecare metsalb e asociat cu un space map
5. tine evidenta in log file, nu in FS
6. activitatea metaslab e mapata ca un balanced-tree in memorie, indexata dupa offset (se poate da replay la log, se pot comnbina blocuri libere continue intr-o singura intrare)

**TRIMing unused blocks** HDD-urile pot face overwrite in place. NVM-urile sufera pentru ca trebuie sa stearga inainte de a scrie, iar stergerile sunt incete. TRIM e un mecanism care infomreaza NV-urile ca acea pagina e libera pentru garbage collection sau daca blocul e liber, acum poate fi sters.

## Eficienta si performanta

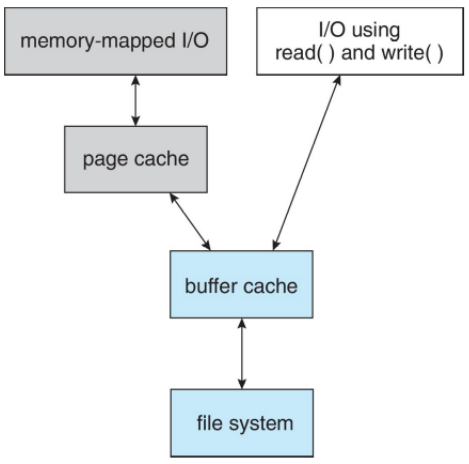
### Eficienta

1. Alocarea pe disc si algoritmii de directoare
2. Tipuirle de date tinute in intrare de fisiere din director
3. Pre-alocarea sau alocarea as-needed metadatelor structurilor
4. Structuri cu marime fixa sau variabila

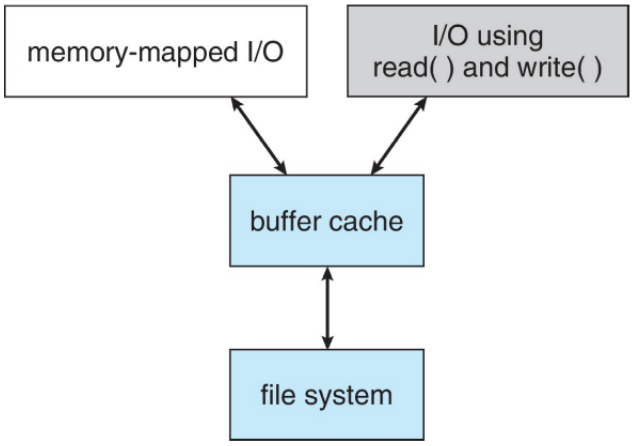
### Performanta

1. Tinerea datelor si metadatelor impreuna
2. Buffer cache - sectiune separata in main memory pentru blocurile folosite des
3. Scrieri sincrone unoeri cerute de programe sau de OS (cele asincrone sunt mai comune, bufferabile si rapide)
4. Free-behind si read-ahead pentru a optimiza accesul secvential
5. Readuri mai incete ca writeuri

**Page cache** - cacheuieste pagini in loc de disk blocks cu tehnici de memorie virtuala si adrese. I/O mapate in memorie folosesc page cache




**Unified buffer cache** - foloseste acelasi page cache ca sa cacheuiasca atat paginile mapate in memorie cat si fisiere normale de system I/O pentru a evita double caching




## Recovery

**Consistency checking** - comapra data din directory structure cu data blocks pe disc si fixeaza incosistentele.

**Backup** pe un alt dispozitiv

**Restoring** a datelor din backup

## Log structured FS

**Log structured/journaling** tin un sistem actualizat de metadate in file system ca transactii

1. Tranzactiile sunt logate - sunt considerate committed in momentul in care apar in log (secvential)
2. Tranzactiile din log sunt scrise asincron in structurile FS (cand structurile sunt modificate, sunt scoase din log)
3. Daca FS crashuieste, toate tranzactiile ramase in log trebuie facute
4. Recovery rapid de la crash cu sanse mici de inconsistente ale datelor