# Notite examen - Sisteme de Operare

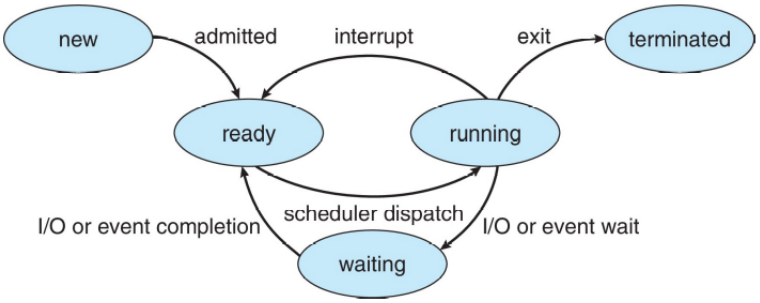
Dinu Florin-Silviu
grupa 231

## Contents

# 1 Processes

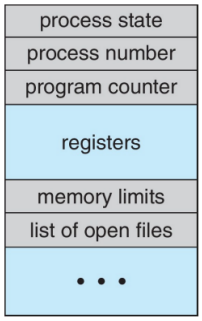
## States

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)

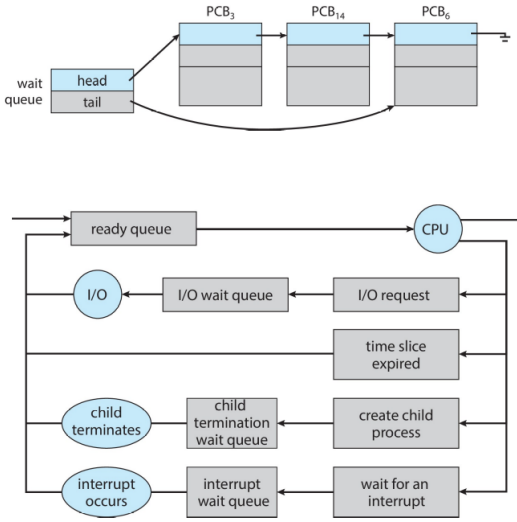



## Process scheduling

**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

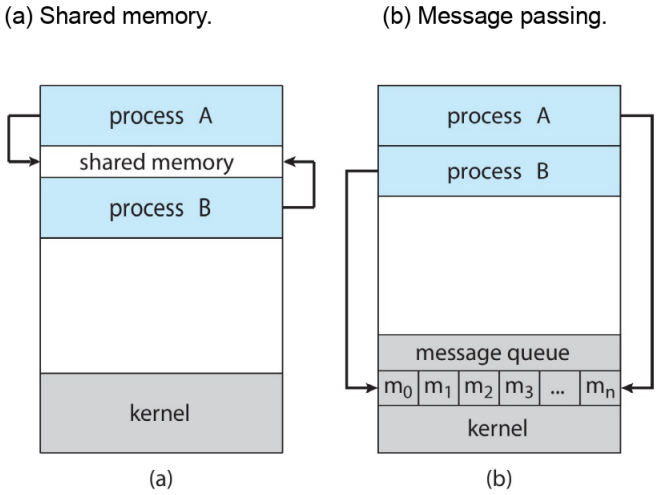
**Cooperarea** inseamna ca procesul poate fi afectate sau afecta alte procese, inclusiv datele partajate

### 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

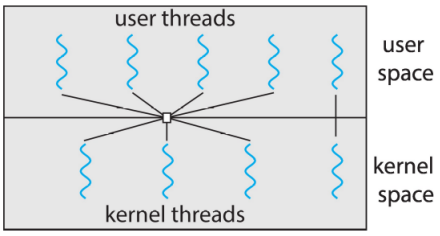
**One-to-One**

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

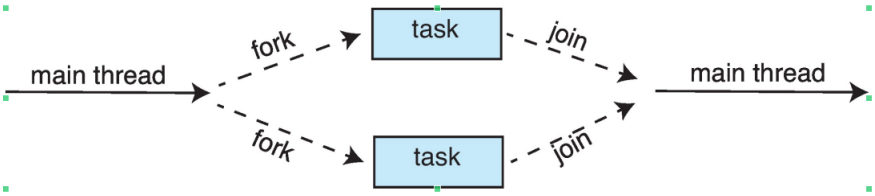
## Implicit threading

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

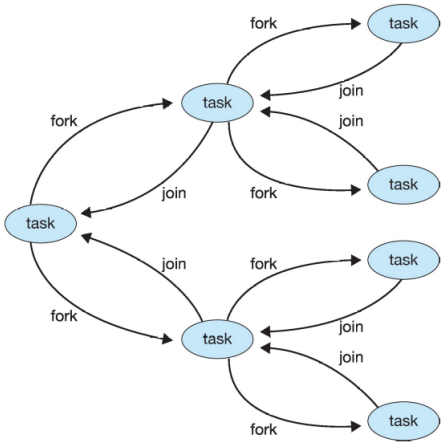



**Algoritmul general**

```
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

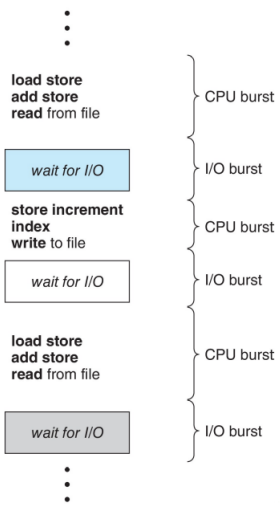
M:M si Two-level au nevoie de comunicare pentru a aloca numarul potrivit de threaduri de Kernel. De obicei se foloseste o structura de date intermediara intre threadurile de user si de kernel LWP (lightweight process). Acesta apare ca un procesor virtual unde procesele pot face scheduleing pentru rularea 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:** $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

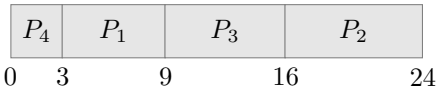
**Average waiting time:** $\frac{(0+24+27)}{3} = 17$

## 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?**

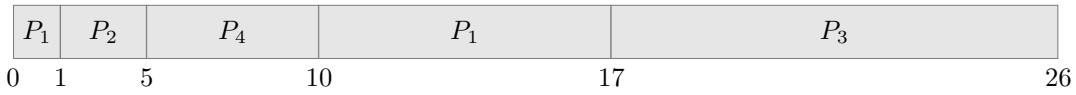
**Estimare:** ar trebui sa fie asemanatoare cu cele anterioare. Poate fi folosita cu exponential averaging

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 |



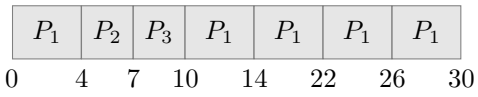

| Process | Completion Time | Turnaround Time (CT-AT) | Waiting time (TAT-BT) |
|---|---|---|---|
| $P_1$ | 17 | 17 | 9 |
| $P_2$ | 5 | 4 | 0 |
| $P_3$ | 26 | 24 | 15 |
| $P_4$ | 10 | 7 | 2 |

**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 |




**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 $\leq 10\mu s$

# 4 Synchronization Tools

# 5 Synchronization Examples