

Notite examen - Sisteme de Operare

Dinu Florin-Silviu
grupa 231

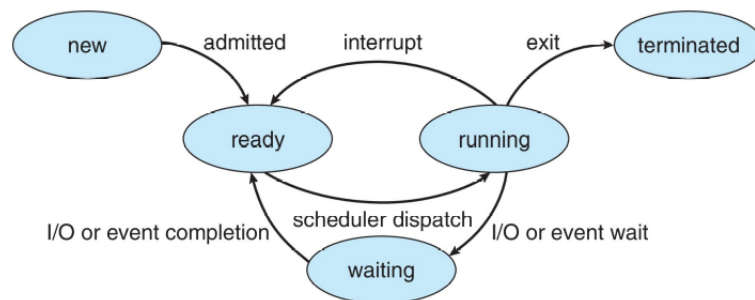
Contents

1	Ch3 Processes	1
2	Ch4 Threads and Concurrency	7
3	Ch5 CPU Scheduling	11
4	Ch6 Synchronization Tools	19
5	Ch7 Synchronization Examples	26
6	Ch9 Main memory	31
7	Ch10 Virtual memory	31
8	Ch13 File system interface	31
9	Ch14 File system implementation	31

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)

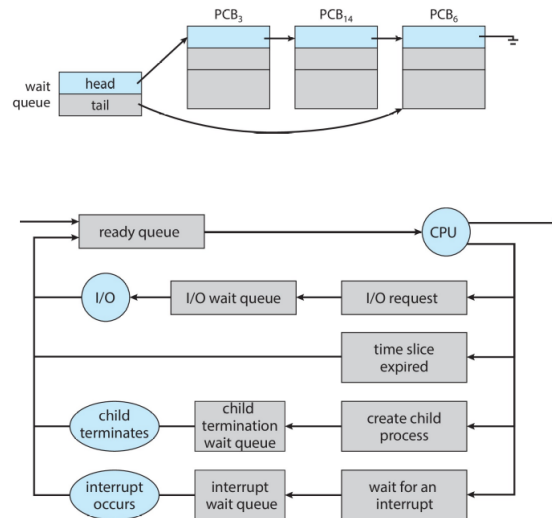
process state
process number
program counter
registers
memory limits
list of open files
...

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

Procese pot fi independente sau sa coopereze

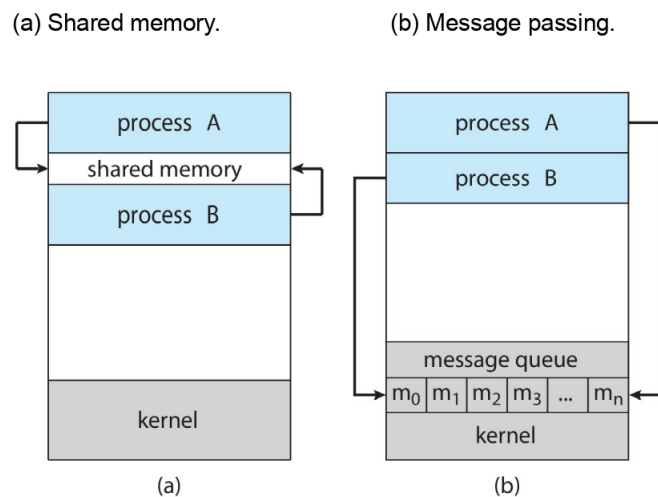
Cooperarea inseamna ca procesul poate fi afectate sau afecta alte procese, inclusiv datele partajate

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	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{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 capacity - 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) cacheaza 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. Producatorul: write-end of the pipe
3. Consumatorul: 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

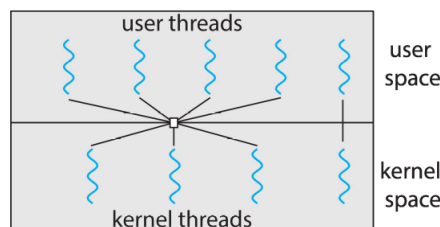
One-to-One

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 librăriei, nu al implementării
5. Comun in UNIX

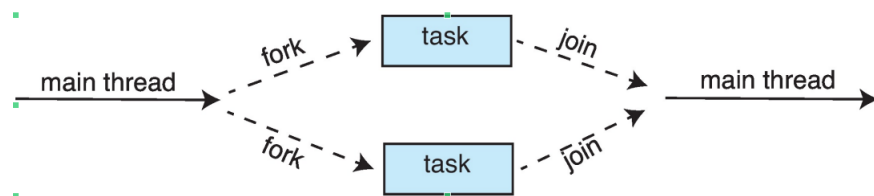
Implicit threading

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 unui 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

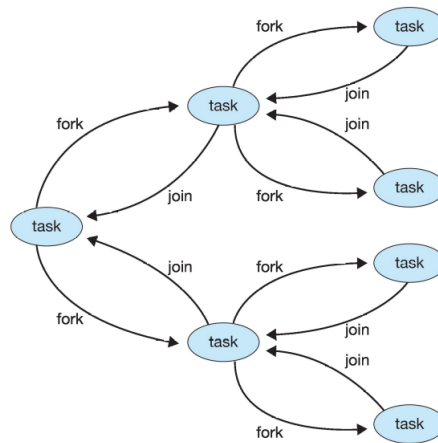


Algoritmul general

```
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++, FORTRAN
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 librerie 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) concurrent - 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

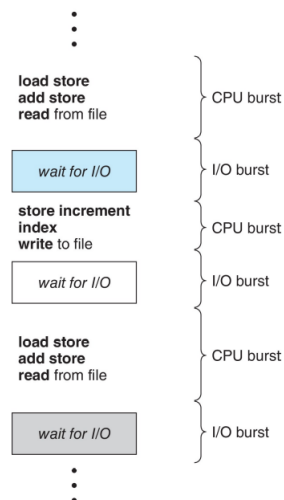
M:M si Two-level au nevoie de comunicare pentru a aloci numarul potrivit de threaduri de Kernel. De obicei se foloseste o structura de date intermediara intre threadurile de user si de kernel LWP (lightweight process). Acesta apare ca un procesor virtual unde procesele pot face scheduling pentru rulara 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



Selectează din procesele din queue (ordonată în moduri diferite) și alocă un core de CPU unui.

1. Running \rightarrow waiting
2. Running \rightarrow ready
3. Waiting \rightarrow ready
4. Terminates

Preemptive poate duce la race conditions

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

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

Convoy effect - short process behind long process

P_1	P_2	P_3
0	24	27 30

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

P_4	P_1	P_3	P_2	
0	3	9	16	24

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. τ_{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 preemptiva. 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

P_1	P_2	P_4	P_1	P_3	
0	1	5	10	17	26

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 fiecare 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 scheduling 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

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1	
0	4	7	10	14	18	22	26	30

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

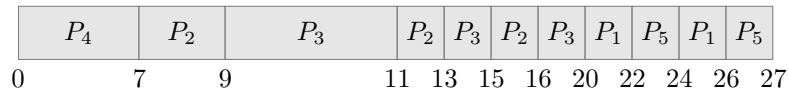
Problema Starvation \Rightarrow **Solutie** Aging

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

P_2	P_5	P_1	P_3	P_4	
0	1	6	16	18	19

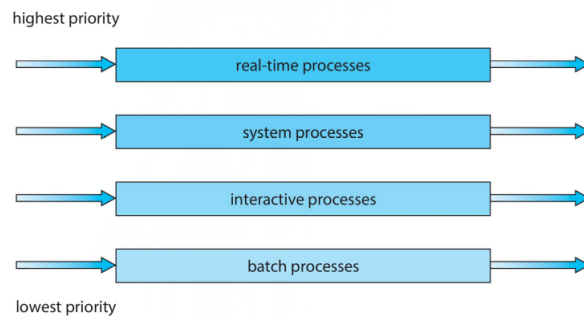
Priority Scheduling cu Round-Robin (time quantum = 2)

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



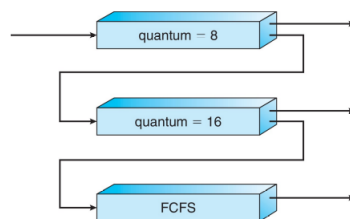
Multilevel Feedback Queue

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 schedulingul 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 fiecare 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 asigura 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 performanta: **interrupt latency**, timpul de la sosirea interruptului 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. Pentru 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 runnable cat timp mai are timp in time slice (active)
5. Daca nu mai are timp (expired) nu mai este runnable pana cand celelalte taskuri isi folosesc sliceurile
6. Toate taskurile runnable sunt tinute per-CPU in runqueue
 - (a) 2 arrayuri de prioritate (active, expired)
 - (b) taskuri indexate pe prioritate

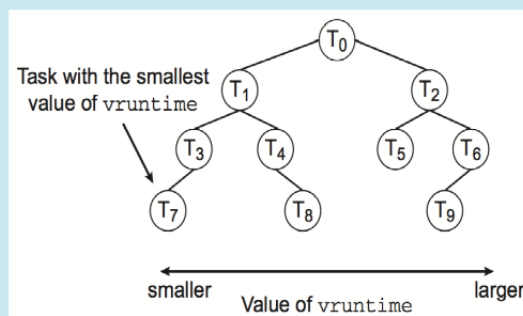
(c) cand nu mai sunt active, arrayurile se schimba

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 $O(\lg N)$ 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

Determinist cu evaluare analitica

Formula lui Little

1. n = average queue length
2. W = average waiting time in queue
3. λ = 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. Mutual 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

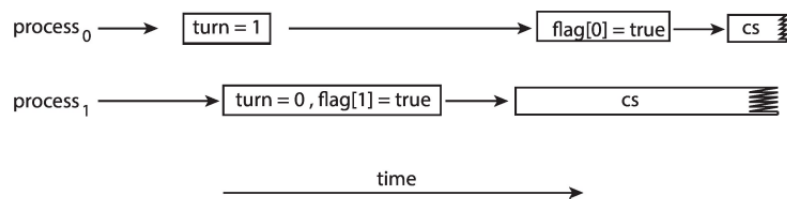

```
while (!flag)
    ;
print x
```
- Thread 2 performs


```
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.

The effects of instruction reordering in Peterson's Solution



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 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 obicei 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 semaphore (integer peste un domeniu nerestricționat) sau binary semaphore (intre 0 si 1, asemanator cu un mutex lock). Un counting semaphore 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.

```
typedef 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 High-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

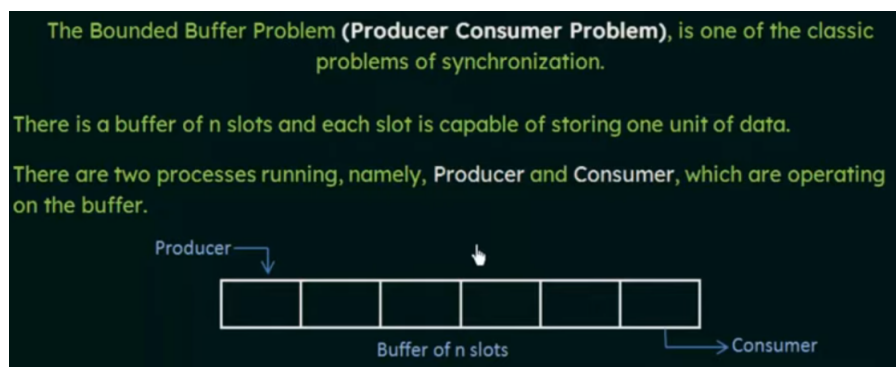
Let S and Q be two semaphores initialized to 1

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>...</code>	<code>...</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

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 filozofi 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 memorie 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 absolute
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 - bindingul 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)

7 Virtual memory

8 File system interface

9 File system implementation