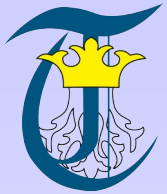


Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare distribuite și în timp real

– curs –
master SECI anul I

ș.l. dr. ing. Kertész Csaba-Zoltán

Despre curs

- Programa analitică, notițe, bibliografie:
  http://etc.unitbv.ro/~csaba.kertesz/sotr
- Contact:
  csaba.kertesz@etc.unitbv.ro

1. Introducere

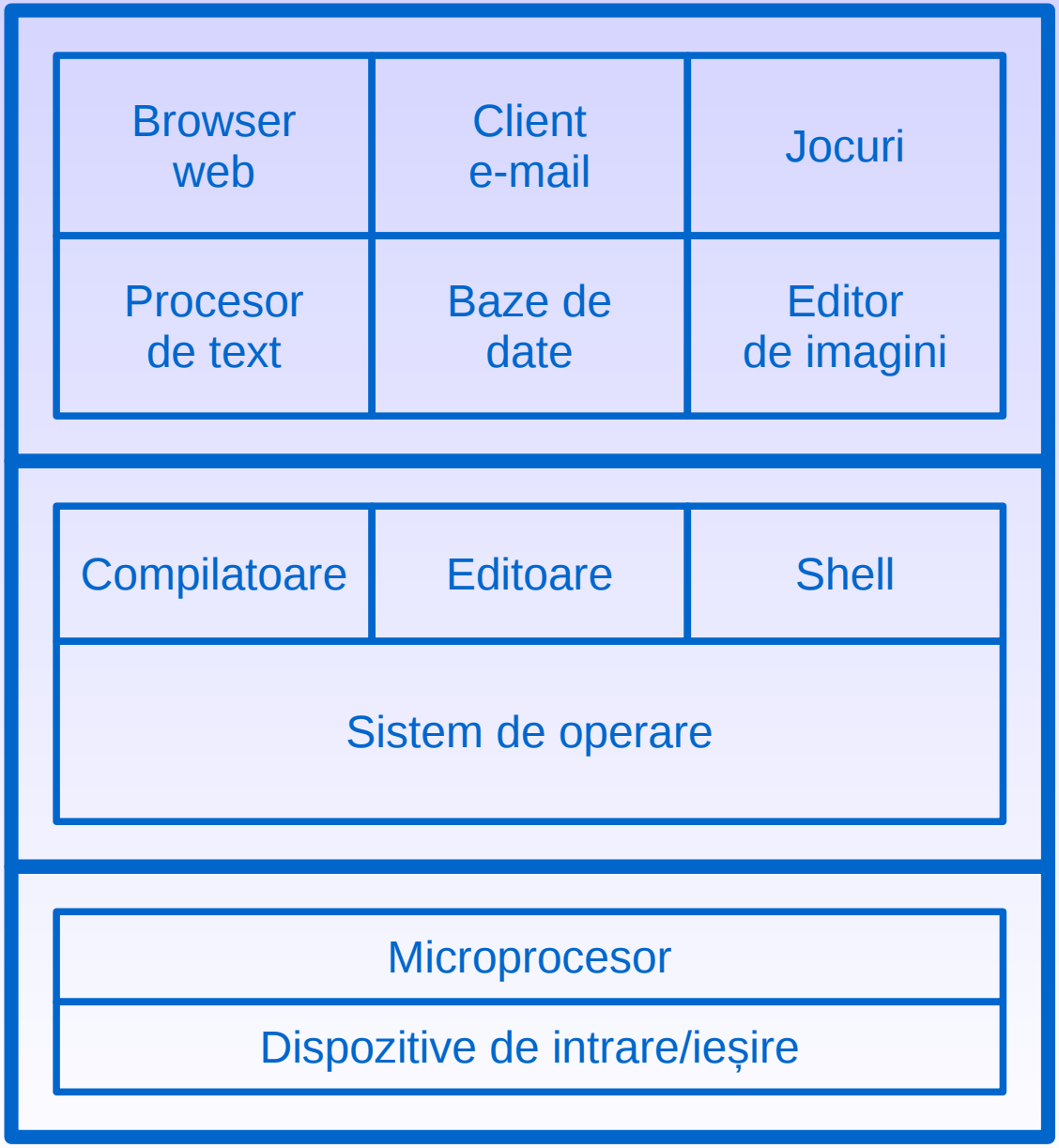
Cuprins

1. Ce este SO? Ce este RTOS?
2. Istoria sistemelor de operare
3. Concepte de bază ale SO
4. Structura RTOS

1.1. Ce este un SO?

- Software:
  - programe de sistem (controlează activitatea sistemului)
  - programe utilizator (rezolvă problemele utilizatorilor
- SO reprezintă o componentă de bază a clasei programelor de sistem
- SO controlează toate resursele calculatorului și oferă acces la acestea programelor utilizator într-un mod convenient și sigur

Software



Programe de aplicație

Programe de sistem

Hardware

Software

- microprocesorul controlează dispozitivele IO prin registre speciale
- SO ascunde acest nivel, oferind o interfață comună pentru programele aplicație
- compilatoare, editoare, shell-uri sunt furnizate împreună de SO, dar nu fac parte din aceasta

Ce este RTOS

- Sistem de operare în timp real:
  - realizează aceleași funcții ca un SO, dar permite implementarea sistemelor în timp real
- Timp real:
  - răspunsul sistemului pentru evenimentele apărute ajunge în timp oportun pentru evenimentul respectiv

Rolul RTOS

- realizează o mașină extinsă:
  - ascunde arhitectura internă a procesorului
  - permite portare ușoară între diferite arhitecturi
  - mai ușor de programat decât hardware-ul direct
- permite implementarea unor funcții cu timp de răspuns determinabil
  - apelurile de sistem sunt predictabile
  - ciclii de mașină în apelurile sistem sunt numărabile

1.2. Istoria sistemelor de operare

- Evoluția SO poate fi împărțită în 4 generații în strânsă corelație cu evoluția calculatoarelor
  - tuburi electronice
  - tranzistoare
  - circuite integrate
  - microprocesoare
- La generația a 4-a a calculatoarelor apare nevoia de diversificare a sistemelor de operare
  - microprocesoare de uz general
  - microcontrolere
  - multiprocesoare

Generația I.
(1945 – 1955)

- calculatoare cu tuburi electronice, fără SO
- programare în limbaj mașină prin legături fizice
- nu există limbaje de programare
  - de la 1950 s-a început folosirea cartelelor perforate
- nu există SO

Generația II.
(1955 – 1965)

- calculatoare cu tranzistoare, SO: sisteme batch
- joburi rulate de pe cartele perforate
- programe se scriau în asamblare sau FORTRAN (și „compilate” în cartele perforate)
- în scopul creșterii eficienței mașinii au apărut sistemele batch, care eliminau operațiile manuale între joburi
  - mai multe joburi se scria pe bandă magnetică
  - de pe bandă mașina prelua joburile pe rând și oferea rezultate pe o bandă de ieșire

Generația III.
(1965 – 1980)

- calculatoare cu IC, SO: multiprogramare
- multiprogramarea era introdusă pe familia de calculatoare OS/360
- utilizare eficientă a mașinilor:
  - în timpul execuției unui job, aceasta trece prin faze care folosesc intensiv procesorul și faze care lucrează cu dispozitive IO lente
  - s-ar putea executa două joburi simultan fără să se deranjeze dacă aceste se află în faze complementare

Generația III.

- soluție:
  - partiționarea memoriei în câteva partiții având fiecare câte un job
  - când un job așteaptă terminare unei operații IO, procesorul este oferit unui alt job din memorie
- creștere a utilizării timpului de procesor
- mecanisme de protecție între joburi

Generația III.
tehnici SO

- spooling (Simultaneous Peripheral Operation On-Line)
  - după terminarea fiecărui job, SO încarcă un nou job de pe disc în partiția goală
- time-sharing
  - joburile concurente pe sistem pot deține controlul procesorului maxim o perioadă prestabilită, după care controlul este oferit altui job

Generația IV.
(1980 – ...)

- integrare de nivel înalt, PC-uri, SO complexe
- software user-friendly
- multe procese rulate în paralel
- SO dominante:
  - DOS/Windows
  - UNIX/Linux
  - MacOS X

Microprocesoare de uz general

- putere mare de calcul vs. un singur utilizator
  - multe procese rulate în paralel
- securitate
- ușurință de implementare

în detrimentul performanței

Multiprocesoare

- creșterea performantei prin distribuirea sistemului de operare pe mai multe procesoare
- funcții de comunicații inter-procesoare avansate
- modele:
  - client-server
  - client-server pe 3 sau mai multe nivele
  - clustere

Microcontrolere

- calculatoare încorporate
- timpii de răspuns și performanța sistemului sunt cele mai importante
- predictibilitatea sistemului
- scopul sistemului și dimensiunea sistemului țintă pot fi foarte variate:
  - SO generic modificat (hard și soft real-time)
  - nuclee mici și foarte mici
  - siguranță ridicată

Sisteme de operare în timp real

- RTLinux — hard real time
- uCLinux — MMU-less, soft
- VxWorks — routere
- Nucleus — Mentor Graphics
- eCos — RedHat
- µC/OS-II — safety critical + reţele
- OSEK — automotive
- TinyOS — monitor + nesC
- FreeRTOS — small GNU kernel

1.3. Concepte de bază ale SO

- Interfața între programele utilizator și SO este definită printr-un set de apeluri sistem (system calls)
- apelurile sistem operează asupra obiecte software:
  - procese
  - fișiere
- între utilizator și SO pot fi intercalate și alte programe sistem, care nu fac parte din SO

Procese

- un proces este un program în execuție
- procesul cuprinde:
  - programul executabil
  - datele și stiva programului
  - registrele de uz general
  - registrele speciale: program counter, stack pointer
- toate informațiile despre un proces sunt stocate într-o tabelă (process table) administrat de SO (câte o intrare pentru fiecare proces)

Procese real-time

- crearea și terminarea proceselor se realizează prin apeluri sistem
  - apelul sistem trebuie să aibă o durată bine definită
- schimbarea între procese (planificarea):
  - preemptiv
  - non-preemptiv
- tratarea întreruperilor trebuie să aibă prioritate ridicată
- durata dezactivării întreruperilor (secțiuni critice) trebuie să fie minimă și bine determinabilă

Procese
Comunicații interproces

- comunicația trebuie făcută sigur
  - secțiuni critice
  - semafoare
  - message-passing
- semnale (întreruperi software) generate de sistemul de operare sau de rutinele de tratare a întreruperilor hardware

Apeluri sistem

- programele utilizator comunică cu SO pentru a cere anumite servicii SO prin intermediul apelurilor sistem
- un apel de sistem este o funcție dintr-o bibliotecă (furnizată de SO)
- apelurile sistem pot fi împărțite în:
  - apeluri non-deterministe ce pot fi apelate din procesele utilizator și pot fi întrerupte
  - apeluri real-time
    - secțiuni critice cu dezactivarea întreruperilor
    - durate deterministe

Managementul memoriei

- SO poate conține un Memory Manager (integrat în kernel sau în bibliotecile standard asociate)
- fiecare proces trebuie să aibă propria zonă de memorie
  - date
  - stivă
  - program
- nu toate arhitecturile oferă sistem de management al memoriei

Sistemul de fișiere

- SO ascunde dispozitivele hardware de utilizator prin sistemul de fișiere
- sunt puse la dispoziție apeluri sistem unitare pentru accesul diferitelor dispozitive:
  - de stocare
  - periferice integrate
  - de comunicație

1.4. Structura SO (RTOS)

- SO este alcătuit dintr-o colecție de proceduri, compilate și legate împreună (de obicei împreună cu aplicațiile utilizator)
- nu există o protecție a informației între aceste proceduri
- pe unele arhitecturi nu există nici protecție între procedurile SO și cele de aplicație
- programele utilizator se rulează în niște taskuri care interferență între ele și cu hardware-ul prin intermediul funcțiilor SO

Structura SO

- întreruperile hardware pot fi tratate de SO sau de utilizator
  - SO tratează întreruperea printr-o rutină foarte scurtă care generează o întrerupere software care poate fi tratată la nivelul taskurilor utilizator
  - utilizatorul tratează întreruperile asigurându-și de durate de execuție reduse și comunicând cu alte procese sau cu SO numai printr-un set redus de apeluri de sistem speciale
- dispozitivele hardware pot fi accesate din programe utilizator numai prin intermediul SO

Structuri SO specifice

- sistemele mici (microcontrolere cu puține resurse) pot avea numai un SO minimal, alcătuit din planificatorul de procese
- aplicațiile trebuie scrise într-o manieră similară cu sistemele fără SO
- avantajul utilizării SO este portabilitatea ridicată și posibilitatea realizării unor funcții lente fără a fi nevoie de mașini de stare complicate

Mașini virtuale

- SO pune la dispoziție o mașină virtuală care separă ermetic procesele utilizator de către mașină respectiv unul de celălalt
- oferă securitate ridicată
- programarea aplicațiilor pot fi făcute în limbaje de nivel înalt similar cu calculatoarele de uz general

2. Generalități RTOS

- Principii SO aplicate pe RTOS
  - procese(taskuri) și threaduri
  - condiții de concurentă și secțiuni critice
  - semafoare
  - planificare Round Robin respectiv priority scheduling

Taskuri

- taskul constă din:
  - copie a programului în memorie (instrucțiuni + constante)
    - memoria de program poate fi diferită de memoria de date, caz în care se folosește numai o mapare a memoriei
  - date + stivă
    - în unele cazuri heapul lipsește: o singură zonă de memorie este alocată stivei și datelor
  - registre
    - registre de uz general, registrul de stare, program counter, stack pointer

Threaduri

- fir de execuție aparținând aceluiași task
- codul, datele sunt comune
- thread conține
  - stiva
  - registrele
  - mapare a memoriei de program și de date
- delimitarea între threaduri și taskuri este mai subțire în cazul RTOS

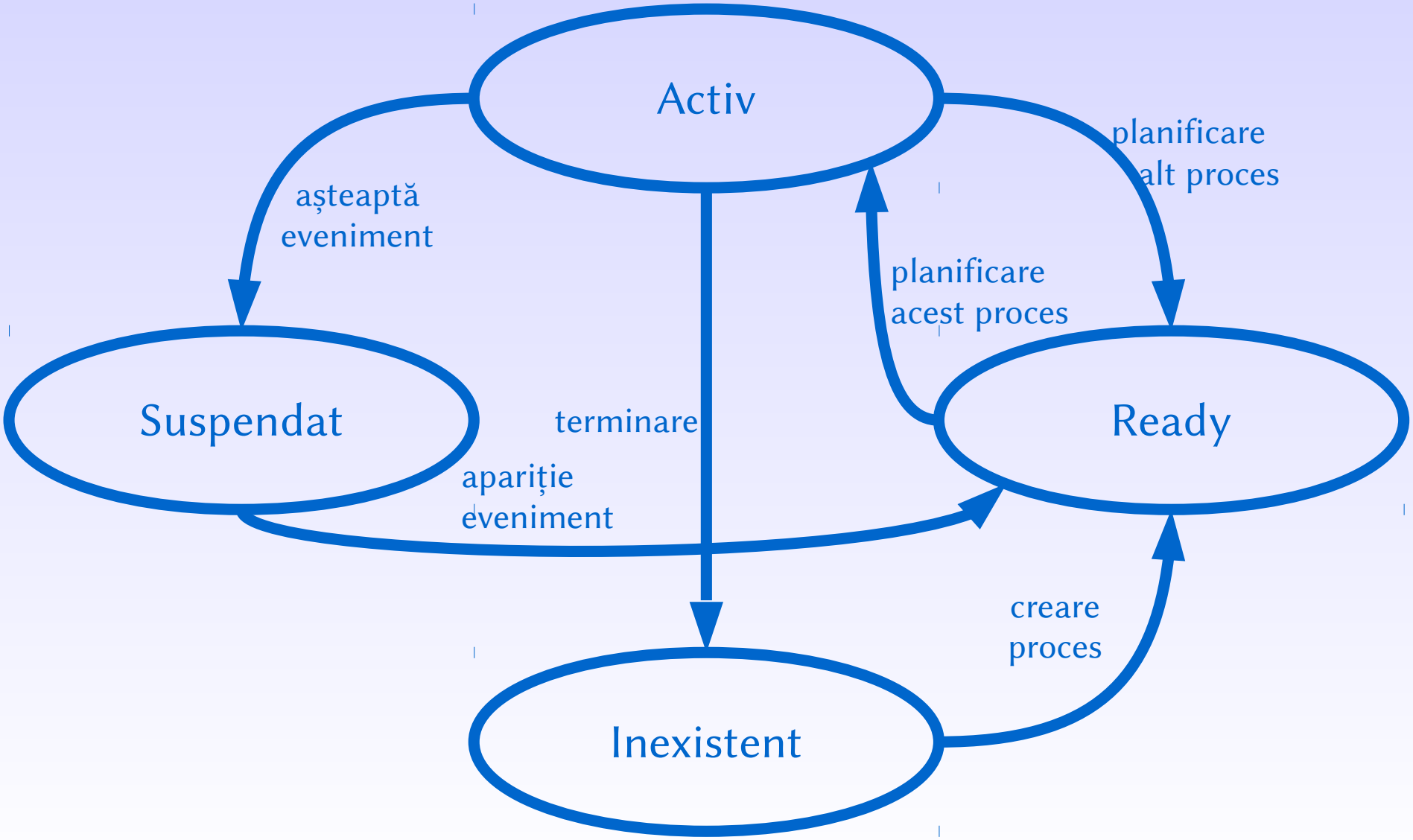
Multitasking

- fiecare proces deține propriul CPU virtual
- în realitate un singur CPU se comută de la un proces la altul
- SO asigură mecanisme pentru crearea și distrugerea proceselor
  - Linux: **fork**
    - presupune existența unui manager de memorie
- SO acordă CPU fiecărui proces pe baza unui algoritm de planificare

2.1.2. Stările proceselor

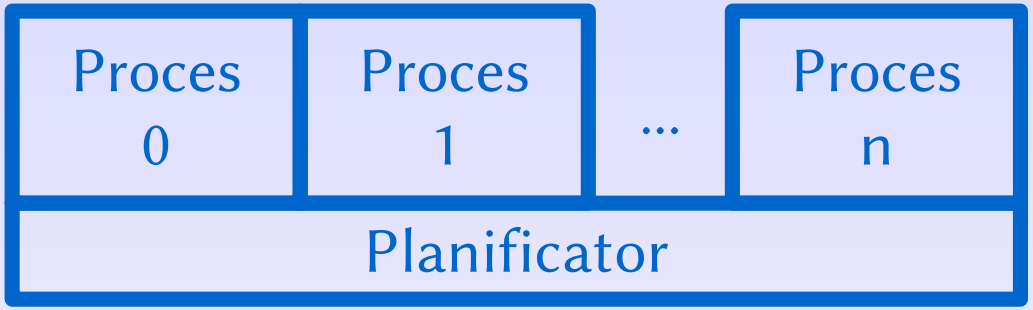
- activ
  - proces în execuție având controlul procesorului
- ready
  - are alocat toate resursele, în afara procesorului
- suspendat
  - așteaptă un eveniment, care să aducă în starea de a concura pentru procesor
- inexistent
  - un program ce nu rulează

Diagrama stărilor



Trecerea între stări

- model SO:



- nivelul inferior este planificatorul (scheduler)
- întreruperile, creările, activările și suspendările de procese sunt înglobate în scheduler
- restul SO este alcătuit din procese

Implementarea proceselor

- SO menține o tabelă (**process table**) cu intrare pentru fiecare proces
  - starea procesului
  - PC, SP, registre
- pe baza informațiilor despre starea procesului, se realizează planificarea acestora
- comunicarea se poate face prin zone de memorie la care au acces amândouă procese

Condiții de concurență

- condiții de concurență la accesul unor resurse
  - în cazul accesului simultan la date, aceste se pot corupe
- coruperea datelor în cazul condiției de concurență trebuie evitat:
  - excludere mutuală a accesului la datele partajate
- secțiuni critice
  - partea dintr-un program unde se realizează accesul la o resursă comună

Condiții de cooperare corectă

- nu pot exista 2 procese simultan în secțiunea critică
- nu se poate face nici o presupunere asupra vitezei și numărului de CPU
- nici un proces care rulează cod în afara secțiunii critice nu poate bloca alte procese
- nici un proces nu va aștepta pentru totdeauna să intre în secțiunea critică

Metode de excludere mutuală

- busy-waiting: procesul așteaptă într-o buclă până la eliberarea resurselor pentru a intra în secțiunea critică
  - nu rezolvă complet secțiunea critică
- dezactivarea întreruperilor
  - la intrarea în secțiune critică se dezactivează întreruperile, iar la ieșirea din secțiune se reactivează
  - planificatorul nu mai poate intervine în interiorul secțiunii critice

Metode de excludere mutuală

- lock variables
- alternare strictă
- soluția lui Peterson
- instrucțiunea TSL
- sleep & wakeup
- wakeup întârziat

Semafoare

- în 1965 Dijkstra a propus folosirea unei variabile care permite semnalizare asincronă între procese:

semafor

- semafor = 0: nici un semnal nu a fost emis
- semafor = n > 0: n semnale au fost emise
- se definesc două proceduri: **down** și **up**

down

- verifică valoarea semaforului
- dacă semafor > 0, decrementează semaforul și continuă execuția procesului
- dacă semafor == 0, execută sleep
- verificarea valorii, modificarea și executarea sleep (dacă e cazul) formează o operație **atomică** (o operație indivizibilă) => pe durata executării acestei operații nici un alt proces nu are acces la semafor
- atomicitate este esențială pentru evitarea condițiilor de concurență

up

- incrementează valoarea semaforului
- dacă unul sau mai multe procese executau sleep determinat de semafor (incapabil să execute **down**), unul din ele va fi ales de sistem și va executa down
- după **up** semaforul poate să rămână 0, dar vor fi mai puține procese blocate de acel semafor
- **up** este de asemenea o operație atomică
- **up** nu poate bloca procesul respectiv

Problema producător / consumator

- două procese partajează un buffer de mărime fixă
  - **producător**: plasează informații în buffer
  - **consumator**: preia informații din buffer
- pentru rezolvare problemei e nevoie de 3 semafoare
  - **full**: contorizarea pozițiilor ocupate
  - **empty**: contorizarea pozițiilor libere
  - **mutex**: excludere mutuală pentru accesul la buffer

Implementare prod. / cons. cu semafoare

```
#define N 100

typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
   int item;
   while (TRUE) {
      produce_item(&item);
      down(&empty);
      down(&mutex);
      enter_item(item);
      up(&mutex);
      up(&full);
   }
}
```

```
void consumer(void)
{
   int item;
   while (TRUE) {
      down(&full);
      down(&mutex);
      remove_item(&item);
      up(&mutex);
      up(&empty);
      consume_item(item);
   }
}
```

Planificarea proceselor

- componenta SO care determină care dintre procesele din sistem va deveni activ:

scheduler

- schedulerul implementează un algoritm de planificare (scheduling algorithm) cu următoare cerințe:
  - **fairness**: fiecare proces să-și preia timpul CPU în mod cinstit
  - **eficient**: să mențină utilizarea CPU cât mai aproape de 100%
  - **timp de răspuns** minim
  - **turnaround**: minimizare timpului de aşteptare
  - **throughput**: maximizarea numărului de joburi pe oră

Strategii de planificare

- la fiecare întrerupere de ceas sau la apariţia unui eveniment se rulează schedulerul
- strategiile
  - **preemptive** care permit suspendarea temporară a proceselor care rulează
  - **non-preemptive** procesele rulează până îşi termină execuția sau se vor bloca în aşteptarea unei operații IO
  - **cooperativ** procesele își asigură (prin programare atentă) de a preda controlul din când în când

Round Robin Scheduling

- este unul din cele mai vechi și simpli algoritmi de planificare
- fiecărei proces i se atribuie un interval de timp numit **cuantă**, în care procesul se poate executa
- dacă procesul nu-și termină execuția până la expirarea cuantei de timp alocate, el va fi întrerupt iar CPU este alocat altui proces
- dacă procesul termină înainte de expirarea cuantei, se va planifica alt proces fără a se aștepta expirarea cuantei

Priority scheduling

- în cazul Round Robin, procesele aveau aceeași prioritate
- uneori însă va trebui să ținem cont de prioritățile în rezolvarea unor probleme
- fiecărei proces i se alocă o prioritate
- la un moment dat va fi rulat procesul cel mai prioritar
- pentru a previne rularea indefinită a proceselor prioritare schedulerul poate descrește prioritatea procesului activ la fiecare întrerupere

Managementul memoriei

- sistemul de operare poate pune la dispoziție și un manager de memorie:
  - alocarea și eliberarea dinamică a memoriei utilizate
    - pentru taskuri: se alocă a zonă de memorie pentru stivă
    - heap: puțin folosit din cauza necesităților mari de resurse
  - segmentare și paginare
    - funcțiile cele mai importante care permit creșterea siguranței de funcționare
    - presupune existența unui suport hardware prezent numai în cazul procesoarelor hardware

Relocatarea și protecția

- multiprogramarea conduce la apariția a 2 probleme ce trebuie rezolvate
  - relocatarea
  - protecția
- joburi diferite vor rula la adrese diferite
- link-editorul va trebui să cunoască la ce adresă de memorie va începe programul
  - dacă prima instrucțiune în program este un salt la adresa 0x100:
    - în partiția 1 va trebui să sară la 100k + 0x100
    - în partiția 2 va trebui să sară la 200k + 0x100

Relocatarea și protecția

- pentru relocatare trebuie modificate instrucțiunile programului la încărcarea acestuia
    - programele încărcate în partiția 1 vor avea 100k adăugat la fiecare adresă
    - programele încărcate în partiția 2 vor avea 200k adăugat la fiecare adresă
- relocatarea în timpul încărcării nu rezolvă problema protecției
  - programele pot întotdeauna construi orice adresă din memorie deci pot citi sau scrie orice cuvânt din memorie

Relocatarea și protecția

- nu este de dorit ca un proces să poată accesa memoria unui alt proces
- soluția: folosirea a două registre
  - bază
  - limită
- la planificarea unui proces se încarcă registrul bază cu adresa de început a partiției, iar registrul limită cu sfârșitul partiției

Relocatarea și protecția

- fiecare adresă folosită de către program va fi relativă la adresa de început a partiției (registrul bază)
- adresele vor fi verificate să nu depășească limita partiției
- numai SO poate modifica registrele bază și limită
- avantajul acestei abordări că putem muta programul oriunde în memorie

Swapping

- în cazul sistemelor time-sharing există mai multe procese care trebuie să se execute
- deseori memoria nu este suficientă pentru a memora toate aceste procese
- unele procese vor trebui mutate pe disc, iar în momentul rulării acestora trebui aduse înapoi în memorie:
- transferarea proceselor între memoria și disc se numește swapping

Memoria virtuală

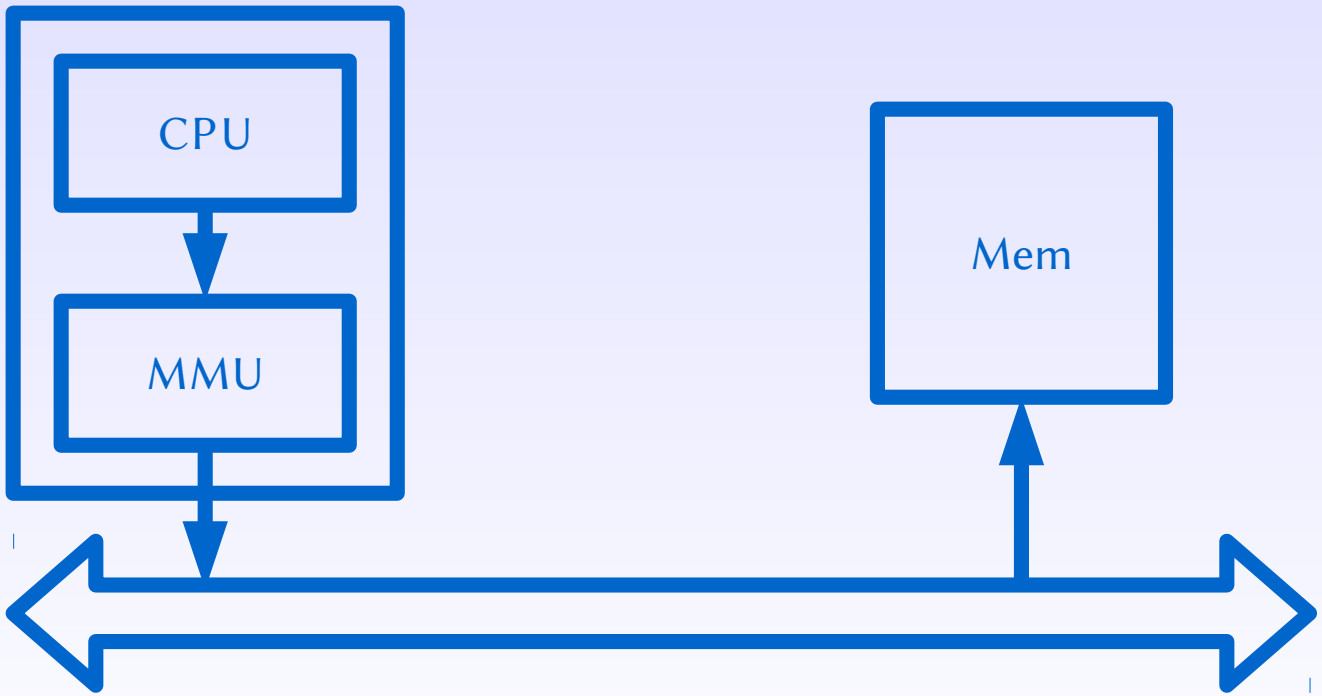
- metodă dezvoltată în 1961 de Fotheringham
- memoria combinată a programului, datelor și stivei poate depăși mărimea memoriei principale
- SO va trebui să mențină părțile care se utilizează la un moment dat în memorie, iar restul să le păstreze pe disc
- de ex. un proces de 1MB poate rula într-o memorie de 256kB, alegând care bucată se execută și transferând bucata respectivă în memorie

Paginarea

- adresele utilizate într-un program sunt numite adrese virtuale și vor forma spațiul adreselor virtuale
- în cazul calculatoarelor fără memorie virtuală, adresa virtuală este plasată direct pe magistrala de adrese: adresa virtuală = adresa fizică
- dacă se utilizează memoria virtuală, adresa virtuală nu va fi plasată pe magistrală ci va fi preluat de MMU (Memory Management Unit)

MMU

- MMU va translata adresa virtuală într-o adresă fizică, ca va fi apoi plasată pe magistrala de adrese



Paginarea

- spațiul virtual de adrese este împărțit în pagini
- paginile vor fi mapate în memorie pe anumite frame page-uri (de același dimensiune ca pagina)
- instrucțiunile de adresare vor folosi adrese virtuale din interiorul acestor pagini, iar MMU va calcula exact ce frame page îi corespunde adresei respective
- dat fiind faptul că spațiul de adrese virtuale este mai mare decât spațiul de adrese fizice, nu toate paginile pot fi mapate în memorie deodată

Page fault

- fiecare pagină precizează printr-un bit dacă este prezent în memorie sau nu
- dacă programul încearcă să utilizeze o pagină nemapată în memorie MMU va genera un trap (întrerupere software) către sistemul de operare numit page fault
- când prinde un page fault, SO va muta pagina cea mai puțin utilizată din memorie pe disc, și va aduce pagina referită de pe disc în memorie și va mapa corespunzător pagina respectivă

Dimensiunea paginilor

- spațiu pierdut mic $\Rightarrow$ **dimensiuni mici**
- pentru anumite sisteme tabela de pagini trebuie încărcată în regiștrii, la fiecare comutare de procese
- pentru o comutarea rapidă tabela de pagini trebuie să fie cât mai mică
- tabelă de pagini mică $\Rightarrow$ **dimensiuni mari**

Paginare pe RTOS

- Swappingul de pagini nu este determinist
- Taskurile de timp real nu au voie să fie eliminate din memorie:
  - extensie POSIX: blocarea paginilor în memorie

Segmentarea

- memoria virtuală realizată prin paginare este unidimensională, deoarece adresele cresc de la 0 până la o adresă maximă
  - există un singur spațiu de adrese
- pentru a evita unele probleme este bine să avem 2 sau mai multe spatii de adrese
  - de ex. dacă un program lucrează cu multe tabele dinamice, una din tabele poate să crească peste spațiul alocat
- o soluție generală, elegantă este să creăm spatii de adrese independente $\Longrightarrow$ segmente

Segmente

- fiecare segment constă dintr-o secvență liniară de adrese, de la 0 până la dimensiunea maximă a segmentului
- lungimea segmentului poate fi între 0 și dimensiunea maximă permisă
- segmentele vor avea dimensiuni diferite
- dimensiunea segmentelor se poate modifica în timpul execuției
  - de ex.: stiva: segmentul crește când se adaugă în stivă, și scade când se extrage din stivă

Segmente

- deoarece fiecare segment are un spațiu de adrese independente, segmentele pot să crească independent fără să se afecteze între ele
- pentru a specifica o adresă pentru o memorie virtuală utilizând segmentarea vom utiliza 2 componente
  - un număr de segment
  - o adresă în cadrul segmentului
- de obicei un segment are un singur scop: segment de date, segment de cod, segment de stivă

Segmentare cu MMU

- Segmentele pot fi asociate paginilor tratate de MMU
- În tabela MMU se inserează și adresele de bază și limită asociată segmentelor
- La acces în afara segmentului se generează page fault ca și la accesul unei pagini inexistente
  - Sistemul de operare detectează că pagina cerută este la o adresă ilegală și termină execuția programului (segmentation fault)

Relocatare și protecție cu MMU

- Pe sistemele cu MMU, relocatarea și protecția între procese se realizează foarte ușor:
  - la crearea fiecărui proces SO asociază fiecărui segment (cod, date, etc.) paginile virtuale necesare pentru dimensiunea dorită
  - paginile virtuale asociate vor păstra identificatorul segmentului
  - la alocare în memoria fizică se va determina apartenența la segment și încadrarea în limita segmentului și se va aloca adrese separate

Sisteme fără MMU

- Nu se poate realiza paginare și segmentare
- SO alocă fiecărui proces o zonă de memorie care va fi folosit pentru toate segmentele
  - taskuri cu PIC (Position Independent Code)
  - modificare tuturor adreselor la încărcare
  - System MAP și rootfs definit de dinainte
- Nu există protecție între taskuri respectiv la datele din SO

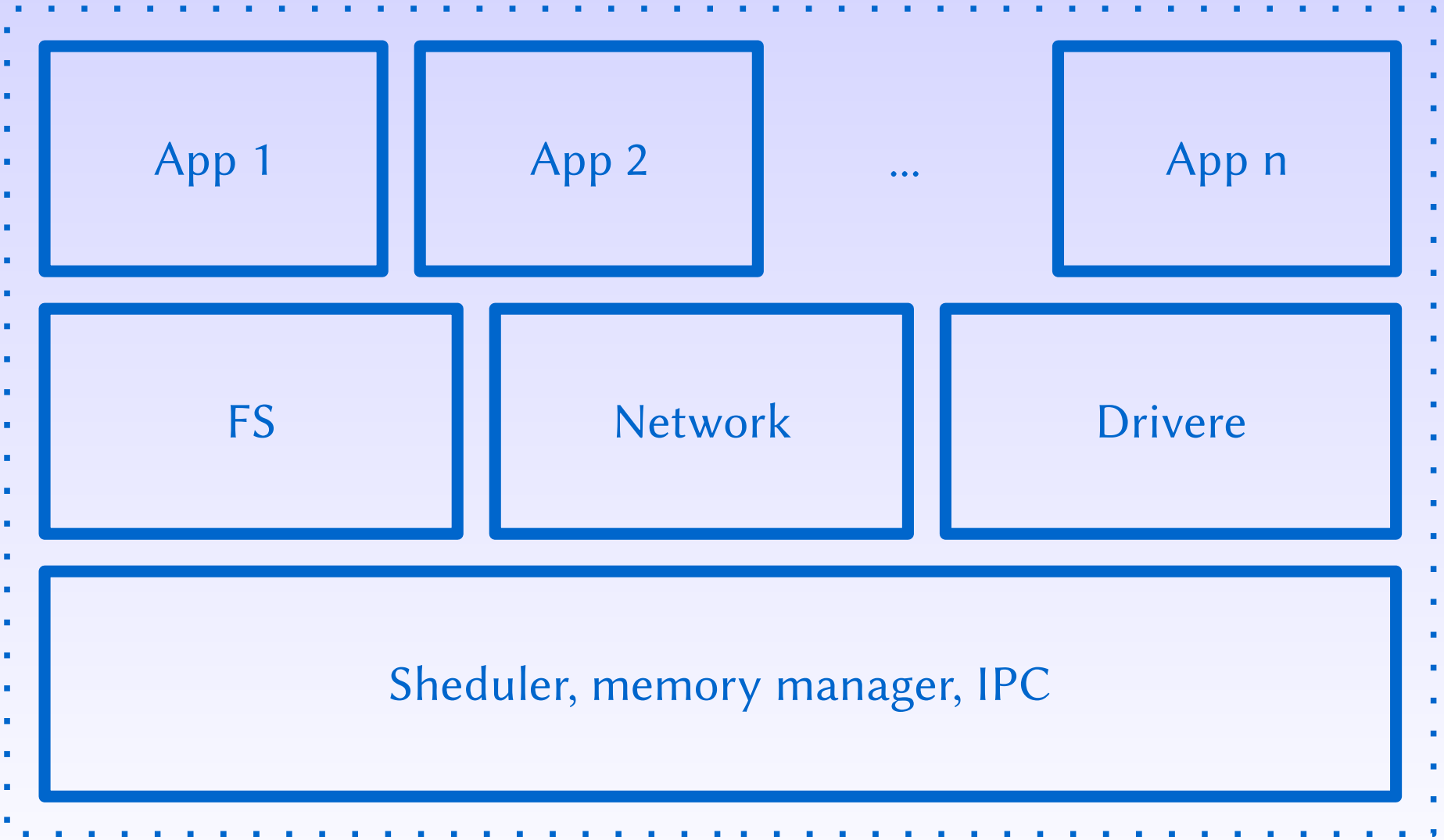
Arhitecturi de sisteme de operare

- arhitecturi posibile pentru sisteme de operare în timp real:
  - real-time executive
  - kernel monolitic
  - microkernel
- această clasificare corelează bine cu soluții de protecții oferite în hardware

Real-time executive

- sisteme RTOS tradiționale
- sisteme fără MMU
- resurse (CPU, memorie) limitate
- kernelul rulează în același spațiu de adrese ca și aplicațiile
- tot sistemul (kernel + aplicații) este compilat într-o singură imagine
- este greu de adăugat aplicații în mod dinamic
- nu se pretează pentru software complex

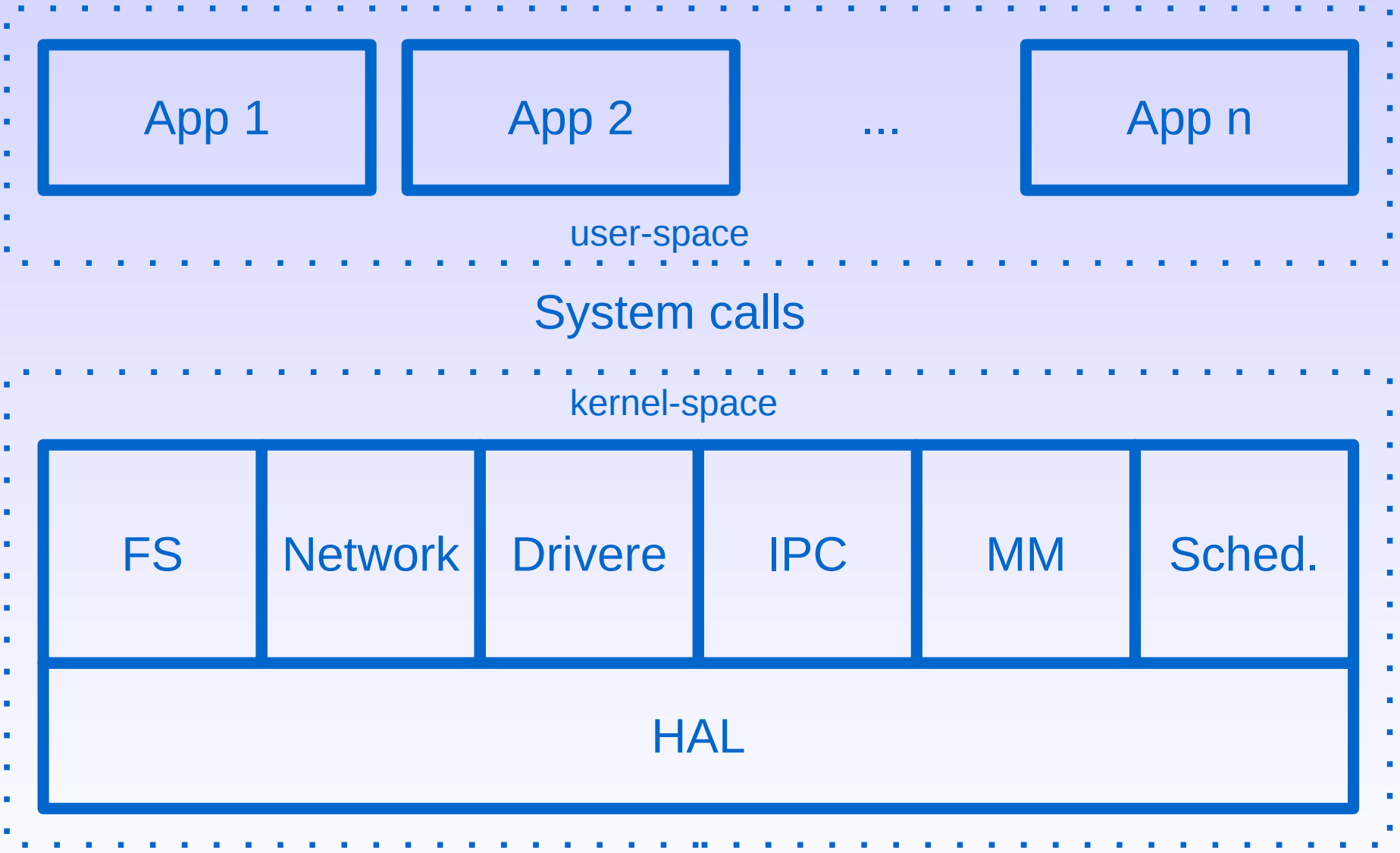
Real-time executive



Kernel monolitic

- distincție între user-space și kernel-space
  - aplicațiile rulează în user-space
  - nucleul și driverele rulează în kernel-space
  - între ele există un strat de apeluri sistem
- suportă sistem complexe cu multe aplicații
- oferă protecție între aplicații
  - dacă o aplicație se blochează sau funcționează incorect atunci nu afectează pe celelalte
- poate rula și pe arhitecturi fără MMU, dar cu protecție limitată

Kernel monolitic



Microkernel

- teoretic cea mai performantă arhitectură de SO
- practic implementările existente prezintă prea multe limitări din cauza complexității
- toate funcțiile SO rulează în user-space în afara unui strat foarte subțire de message passing
- sistemul are un overhead mare datorită mulțimii de mesaje ce trebuie să treacă din user-space în kernel-space și invers
- poate fi folosit numai pe sisteme cu MMU, altfel pierde avantajul protecției maxime

Microkernel