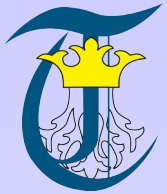


Universitatea *Transilvania* Brașov
Facultatea de Inginerie Electrică și Știința Calculatoarelor
Departamentul de Electronică și Calculatoare

Sisteme de operare distribuite și în timp real

– curs –
master SECI anul I

ş.l. dr. ing. Kertész Csaba-Zoltán

Compilarea sistemului

- în mod tradițional compilarea sistemului embedded se realizează prin compilarea sistemului de operare și a aplicațiilor într-o singură imagine
  - rezultă într-un singur fișier binar ce poate fi încărcat în memoria nevolatilă a sistemului
  - legarea împreună permite înlăturarea codului nefolosit din sistemul de operare rezultând în dimensiuni reduse a sistemului
  - se poate realiza o optimizare a codului între kernel și aplicații rezultând în sistem mai mic și mai rapid

Compilarea embedded Linux

- Linux urmărește principiul de pe desktop: sistemul de operare și aplicațiile sunt compilate separat
  - ușurință în dezvoltarea aplicațiilor și în modificarea funcționalității sistemului
  - portabilitate crescută
  - siguranță ridicată
  - dimensiunea sistemului este mai mare, cu funcții din kernel nefolosite, cu rutine speciale de interfațare aplicații – kernel
  - complexitatea compilării sistemului crește

Compilarea kernelului

- kernelul Linux vine cu un sistem de compilare încorporat (kbuild) bazat pe GNU make
- mecanismul kbuild oferă un sistem simplificat și ușor de folosit pentru configurarea și compilarea kernelului
- sistemul kbuild poate fi ușor extins cu rutine proprii de configurare și scripturi de automatizare ce permit adaptarea ușoară la diferite sisteme

Pașii necesari pentru compilarea kernelului

- instalarea unui mediu de cross-compilare
  - implicit sistemul vine cu un mediu de compilare pentru host (binutils, gcc)
  - compilarea pentru sistemul embedded are nevoie de un cross-compiler: compilatorul este executat pe un host, dar creează cod pentru sistemul țintă (ex: gcc-arm, gcc-mips, etc.)
- configurarea kernelului
  - procesul de selectare a componentelor din kernel ce vor fi compilate
  - mai multe metode: make [config | menuconfig | xconfig]

Pașii necesari pentru compilarea kernelului

- compilarea surselor și legarea fișierelor obiect
  - make
  - se compilează sursele selectate la faza de compilarea și se leagă toate împreună într-o imagine vmlinux
  - pe 2.4 este nevoie de crearea dependințelor de fișiere header (make dep) înainte de comanda make, pe 2.6 nu este necesar
  - pe 2.4 dacă comanda make se emite înainte de terminarea configurării, se intră într-o interfață interactivă make config, pe 2.6 se generează un mesaje de eroare

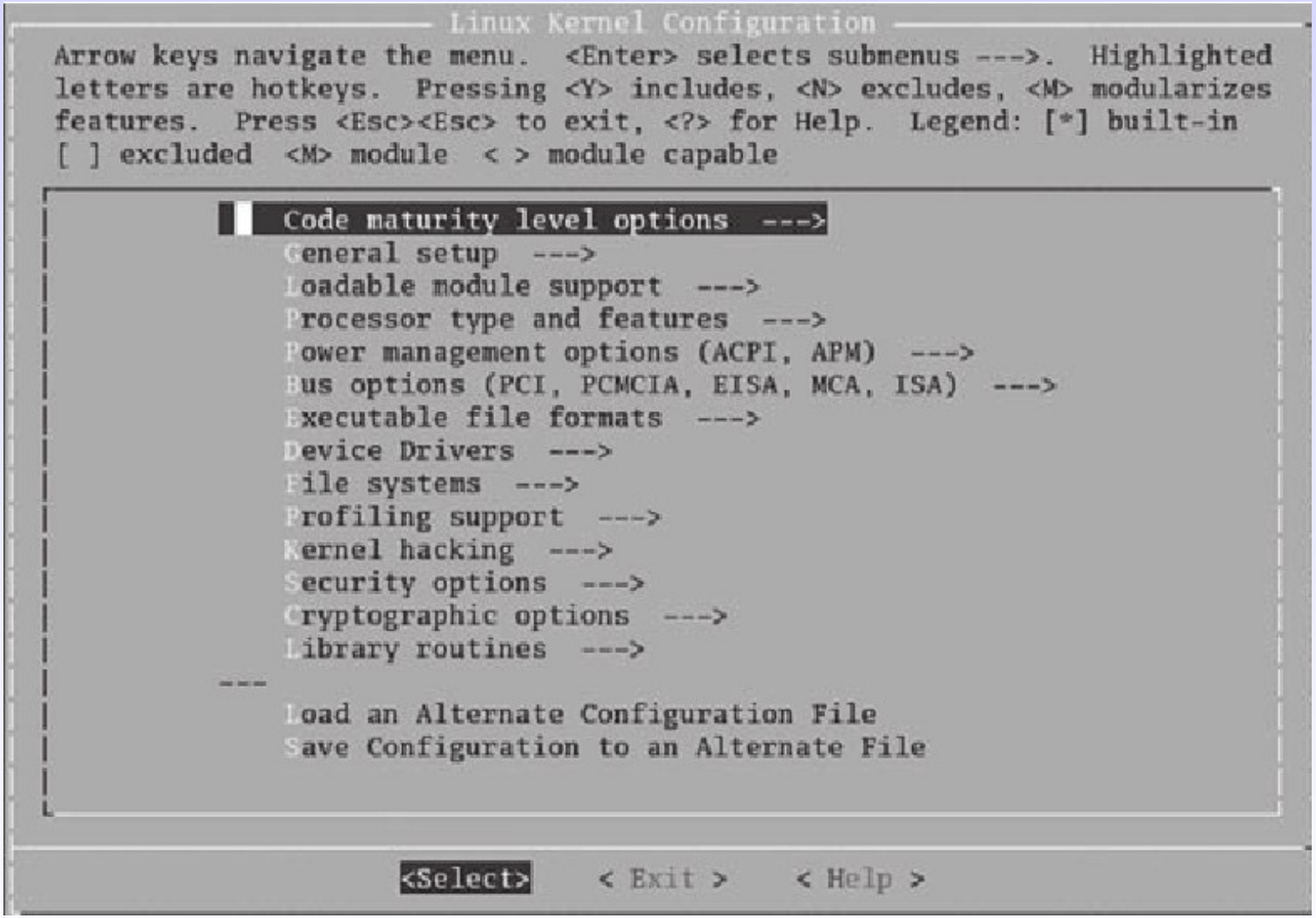
Pașii necesari pentru compilarea kernelului

- poate exista o fază de postprocesare a imaginii kernelului dependent de arhitectură și de sistemul de compilare
  - de obicei inclus în faza make, dar nu e standardizat
  - poate include comprimarea imaginii (vmlinuz), crearea unei imagini de root, cod de bootstrap, etc..
- compilarea modulelor
  - dacă kernelul a fost configurat să folosească anumite componente ca module încărcate dinamic, acestea trebuie compilate separat
  - make modules

Compilarea kernelului pe diferite arhitecturi

- la configurare utilizatorul poate seta arhitectura țintă: tipul procesorului, placa folosită, etc.
- fiecare placă are propriile setări și surse în subdirectorul arch
- make va parcurge aceste subdirectoare căutând config.in (2.4) sau Kconfig (2.6) în care sunt stocate configurările posibile pentru arhitectura specifică
- un BSP trebuie plasat în aceste directoare, cu fișiere de configurare corespunzătoare pentru a putea fi compilat în kernel

Configurarea kernelului



Configurarea kernelului

- pasul cel mai important în compilarea sistemului
- poate fi realizat prin comenzile
  - make config
    - sistem bazat pe interacţiune la linia de comandă
  - make menuconfig
    - sistem bazat pe meniuri (în mod text)
  - make gconfig
    - sistem grafic bazat pe GTK
  - make xconfig
    - sistem grafic bazat pe Qt

Fișiere de configurare

- configurarea este apelată de make, dar folosește propriul sistem de configurare cu fișiere dedicate (config.in în 2.4, Kconfig în 2.6) și propriul limbaj script pentru acestea
- punctul de pornire este fișierul de configurare specific arhitecturii (în subdirectorul arch)
  - aceasta este și primul punct în meniul de configurare
  - acest fișier invocă celelalte fișiere de configurare a subsistemelor din kernel (fs, net, etc..) în funcție de posibilitățile oferite de placă

Elemente de configurare

- fiecare element de configurare este stocat în forma nume=valoare
  - valoarea poate fi
    - bool: oferă valorile y,n
    - tristate: oferă valorile y,n,m
    - string: orice șir de caractere ce trebuie inclus în kernel ca atare
    - integer: un număr oarecare ce trebuie inclus în kernel
    - hexadecimal: similar cu integer
  - elementele pot avea o valoare implicită, și pot fi dependente de alte elemente

Exemplu de configurare unui driver

- în subdirectorul arch specific procesorului, se creează un subdirector pentru driverul respectiv cu codul sursă de ex. driverul_nostru.c
- în fișierul Kconfig se include o intrare:

```
config DRIVERUL_NOSTRU
    bool
    help
        Aceasta este driverul nostru
```

- în fișierul Makefile se include

```
obj-$(CONFIG_DRIVERUL_NOSTRU) += driverul_nostru.c
```

- la configurare se va genera un fișier .config ce conține CONFIG_DRIVERUL_NOSTRU = y

Compilarea aplicațiilor

- compilarea aplicațiilor trebuie făcută de asemenea prin cross-compilare
- pentru a ușura portabilitatea pentru compilare se pot folosi tool-uri GNU:
  - autoconf
  - automake
  - libtool
- aceste tool-uri permit generarea unor sisteme de compilare make generalizate și portabile pornind de la Makefile-uri simple

Configurarea aplicațiilor

- prin folosirea tool-urilor autoconf/automake pentru configurare se poate folosi scriptul configure:

  ./configure --host=<target> --build=<build-system>

- configure va genera Makefile și config.h, ce vor compila sistemul specific arhitecturii
- probleme la cross-compilare:
  - configure nu poate executa programele de test (de ex. sizeof(int))
  - acestea vor trebui introduse manual în config.cache

Crearea sistemului root

- ultima fază în crearea sistemului embedded
- se realizează o imagine a sistemului, ce conține kernelul și aplicațiile, și care poate fi montat la bootarea sistemului pentru a permite inițializarea sistemului și încărcarea aplicațiilor
- pot fi create două tipuri de sisteme de fișiere pentru root: ramdisk și ramfs
  - ramdisk emulează un dispozitiv bloc în memorie
  - ramfs stochează toate datele asociate fișierelor cacheul kernelului
  - pentru a crea imaginile se folosesc comenzile mkinitrd respectiv mkinitramfs

Portarea aplicațiilor

- avantajul principal al sistemelor de operare este posibilitatea unei portări ușoare ale aplicațiilor de pe un sistem pe alta
- dificultăți mai mari poate reprezenta portarea aplicațiilor între diferite sisteme de operare:
  - de pe RTOS (sau RT executive) clasic pe Linux, datorită nevoii de flexibilitate mai mare sau pentru a permite folosirea unor subsisteme gata făcute (stiva TCP/IP, servere, FS, ...)
  - de pe PC (cu Linux) către un dispozitiv embedded (RTLinux sau uCLinux)

Portarea între RTOS clasic și embedded Linux

- pentru portarea aplicațiilor trebuie rezolvate diferențele arhitecturale între sistemele de operare:
  - kernel API
  - managementul memoriei
  - managementul proceselor
  - IPC
  - operațiile IO

Kernel API

- interfața prin intermediul căreia aplicațiile comunică cu kernelul
- sistemele RTOS de obicei furnizează o serie de funcții (care conțin tot codul OS) care se leagă împreună cu aplicațiile
  - aplicațiile pot apela aceste funcții pentru a realiza operații cu kernel
  - totalitatea acestor funcții reprezintă kernel API
  - ex FreeRTOS: vTaskSuspend, vTaskResume

Linux kernel API

- funcțiile kernel și aplicațiile rulează în spatii diferite
- funcțiile din user space nu pot apela direct funcții din kernel space: numai driverele rulate în kernel space pot apela API-ul Linux
- comunicarea între aplicații și kernel se realizează prin interfețele FS sau system call:
  - aplicațiile pot accesa fișiere speciale puse la dispoziție de diferite drivere
  - aplicațiile pot folosi un apel sistem (system call) realizat prin intermediul unei întreruperi soft (trap)

Operating System Porting Layer

- pentru portarea aplicațiilor RTOS pe Linux trebuie implementat un OSPL
- minimizează numărul de modificări asupra codului aplicației prin maparea funcțiilor oferite de API-ul RTOS la API-ul Linux:
  - mapare unu-la-unu: unele funcții oferite de RTOS pot fi mapate la un apel sistem oferit de Linux

```
#define xTaskHandle int
void vTaskSuspend(xTaskHandle pid)
{
   kill(pid, SIGSTOP);
}
```

  - mapare unu-la-multe: când o funcție RTOS nu poate fi realizat de un singur apel sistem

OSPL pentru funcții kernel space

- unele funcții din aplicația RTOS pot fi rulate în kernel space:
  - driverele, funcții care lucrează numai cu driverele,...
- funcțiile RTOS pot fi mapate și pentru funcții din Linux kernel API
- dacă o funcție apelează atât funcții user-space cât și kernel-space maparea simplă devine imposibilă
  - trebuie creat un kernel API driver, care va oferi o interfață către user space pentru a apela funcția din kernel space

Managementul memoriei

- RTOS nu oferă soluții de management al memoriei avansate (numai malloc și free)
- harta memoriei în Linux este împărțit în kernel space și user space (cu fiecare proces în zona lui separată)
  - chiar și în cazul uCLinux (fără MMU) împărțirile sunt păstrate
- trebuie realizat o mapare în memorie atentă a aplicațiilor

Variabile globale

- variabilele globale nu suportă bine transferul la un sistem cu MM
- dacă o bibliotecă partajată (o funcție care poate fi apelat din mai multe taskuri) folosește o variabila globală, în linux această variabilă va deveni globală numai în cadrul unui proces:
  - linux va crea datele asociate bibliotecilor partajate pentru fiecare proces în parte (numai textul va fi partajat între procese)
- pentru a evita modificarea codului, toate taskurile ce apelează astfel de funcții vor trebui puse în același proces

Managementul proceselor

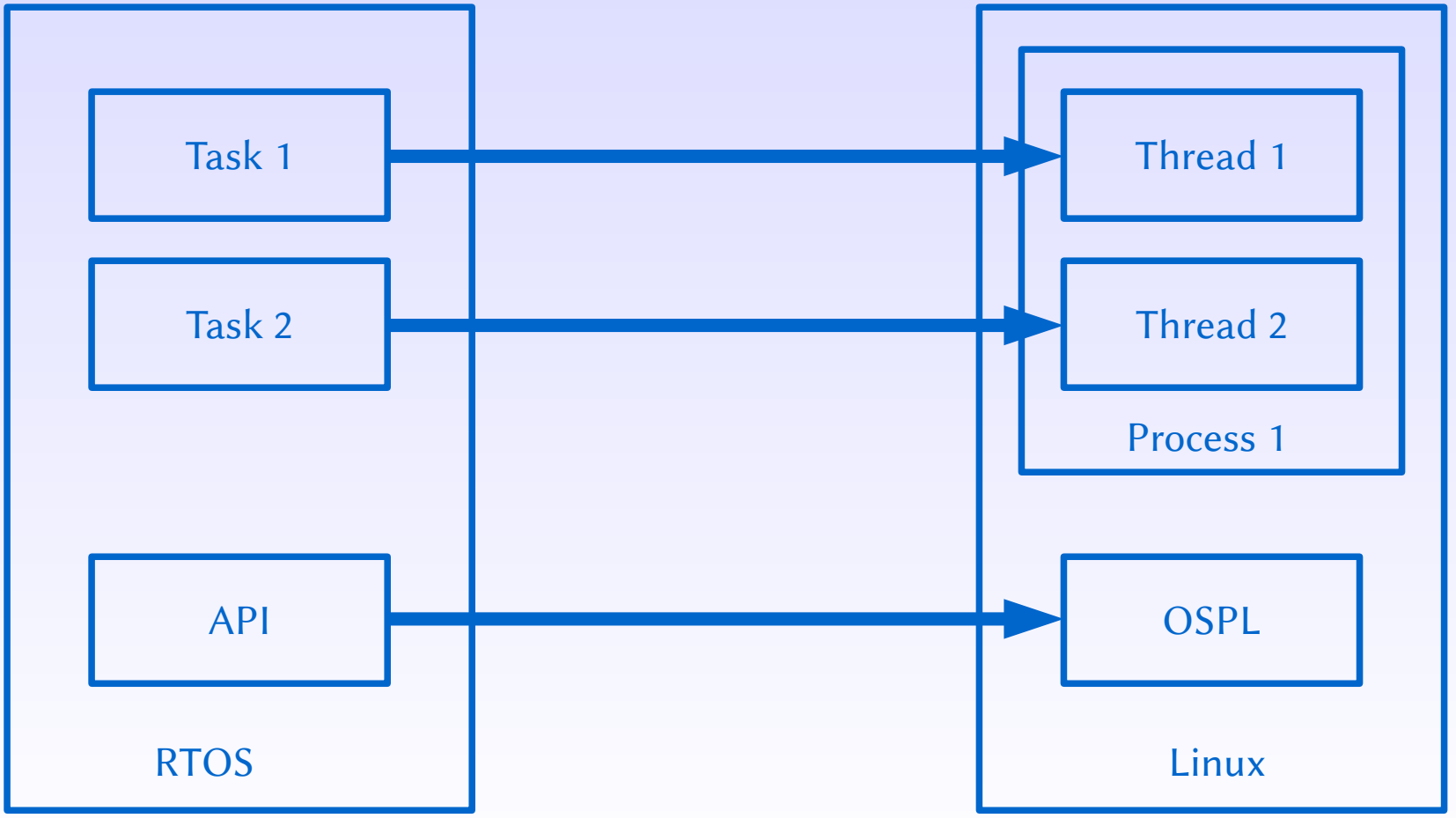
- SO sunt caracterizate prin faptul că permit rularea pseudoparalelă a mai multor aplicații
- în RTOS aceste aplicații se traduc în taskuri sau procese:
  - programe aflate în execuție cu stiva, datele și contextul asociat
  - OS permite schimbarea între aceste procese prin intermediul planificatorului
- un task poate fi executat:
  - până la terminarea funcție (de ex. rutine de init)
  - într-o buclă infinită (funcțiile de bază a sistemului)

Multitasking în Linux

- Linux oferă două nivele pentru rularea aplicațiilor în paralel:
  - procesele
    - programul aflat în execuție împreună cu datele, stiva și contextul asociat
    - numai textul poate fi partajat între procese
  - fire de execuție
    - programul aflat în execuție împreună cu stiva și contextul asociat
    - datele sunt partajate între fire de execuție în cadrul unui proces

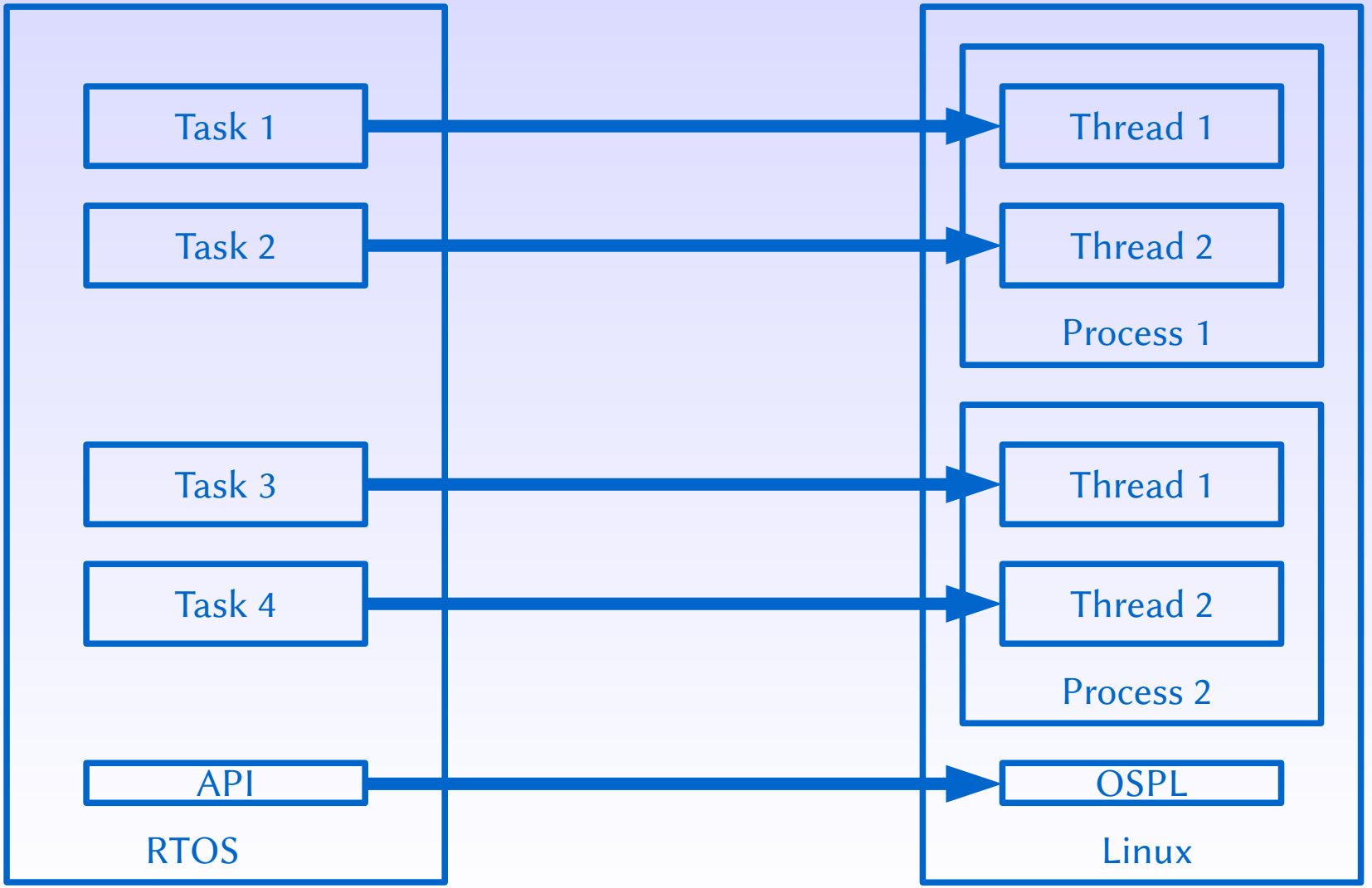
Portarea taskurilor pe Linux

- fiecare task poate fi transformat într-un thread din cadrul aceluiași proces



Portarea taskurilor pe Linux

- taskurile pot fi grupate în procese diferite



Strategii de portare a taskurilor

- taskurile strâns cuplate (prin variabile globale partajate) pot fi puse în threaduri separate din cadrul unui singur proces
  - este metoda de portare cea mai directă și mai ușoară
- taskurile independente sau cuplate prin intermediul IPC oferit de SO se pun în procese diferite
- taskurile importante (la funcționalitatea sistemului) ar trebui puse în procese diferite pentru protecție maximă

Lucrul cu threaduri POSIX

- Linux implementează threaduri POSIX (pthreads)
- pentru portarea taskurilor în pthreaduri, aplicațiile trebuie adaptate la standardul pthread API
- crearea threadurilor

```
int pthread_create(pthread_t* thread_id,
                   pthread_attr_t* thread_attributes,
                   void* (*start_routine)(void*),
                   void* arg);
```

  - se crează un nou fir de execuție pornind de la funcția `start_routine`

Terminarea threadurilor

- pentru a transmite starea de terminare threadul poate apela funcţia

```
void pthread_exit(void* return_val);
```

  - similar cu funcția exit, dar va termina numai threadul care a apelat
  - dacă funcția asociată thread-ului se termină atunci nu se va transmite stare de terminare
- pentru ca un thread sa citească starea de terminare se folosește funcția

```
int pthread_join(pthread_t tid, void** thread_return_val);
```

  - similar cu funcția wait

Terminarea threadurilor

- dacă starea de terminare a unui thread nu a fost citit, atunci resursele asociate threadurilor nu vor fi eliberate
- în cazul în care un thread nu furnizează starea de terminare atunci aceasta nu poate fi citită, pentru a permite totuși eliberarea resurselor, threadul trebuie detașat de threadul care l-a creat

```
int pthread_detach(pthread_t tid);
```

- portarea unei aplicații în care taskurile își încheie rularea la un moment dat, necesită și modificarea codurilor aplicației

OSPL RTOS - threaduri

- portarea aplicațiilor RTOS alcătuite din funcții în bucle infinite este foarte ușoară în threaduri
- fiecărei task va corespunde un thread (care nu se termină niciodată)
- variabilele globale accesate de taskuri pot fi accesate de threaduri ca o variabilă globală de proces
- funcția în care sunt create threadurile trebuie să aibă și el o buclă infinită el reprezentând cadrul procesului în care se execută threadurile

Sincronizarea threadurilor

- pthreads oferă metode robuste pentru sincronizare prin intermediul unui API userspace
- mutex

```
void pthread_mutex_init(pthread_mutex_t* mutex,
                        const pthread_mutexattr_t* mutexattr);
int pthread_mutex_lock(pthread_mutex_t* mutex);
int pthread_mutex_unlock(pthread_mutex_t* mutex);
```

- semafoare

```
void sem_init(sem_t* sem,
              int pshared,
              unsigned int value);
int sem_wait(sem_t* sem);
int sem_post(sem_t* sem);
```

Metode IPC

- Linux oferă metodele IPC POSIX:
  - cozi de mesaje
  - semafoare
  - memoria partajată
- portarea taskurilor în procese independente, presupune portarea IPC existente în astfel de IPC
  - semafoare
    - interfețe către structuri păstrate de kernel

```
int semget(key_t key, int nsems, int semflg);
int semop(int semid, struct sembuf* sops, unsigned nsops);
```

IPC în kernel space

- threadurile rulate în kernel space (ex. driverele) pot accesa direct structurile kernel asociate cu metodele IPC
- astfel aceste threaduri pot folosi metodele IPC prin intermediul unor funcții API simple
- portarea unor aplicații RTOS pe Linux cu procese independente poate fi ușurată prin implementarea unor drivere de kernel API, care vor interfața funcțiile IPC folosite din RTOS

Portarea în kernel space

- toată funcția este portată în kernel space:
  - cel mai ușor de realizat
  - nu oferă nici o protecție
- se extinde funcțiile de apel sistem pentru a crea interfețele aplicație kernel necesare
  - toate apelurile sistem sunt înregistrate în tabela system calls din kernel
  - upgrade-ul la un kernel nou presupune crearea din nou a apelurilor sistem

Portarea în kernel space

- se realizează funcții user space care vor apela la un driver kernel API încărcat ca un modul
- driverul crează o intrare în /dev și furnizează o funcție de ioctl pentru dispozitivul respectiv
- funcțiile user space (care emulează funcțiile API din RTOS) vor deschide acest dispozitiv și vor executa un ioctl corespunzător