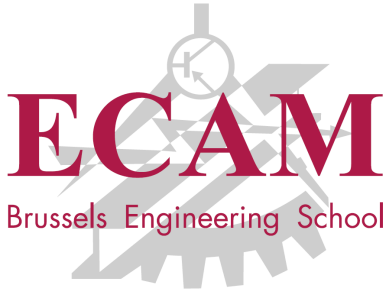

I403C Systèmes d'exploitation temps réel

Séance 5

Mode utilisateur et appels systèmes

Sébastien Combéfis

2017–2018

Rappels

- Description du Memory Management Unit (MMU)
  - Espaces d'adresses physiques et virtuelles
  - Registres, configuration et activation du MMU
  - Contrôle d'accès de domaine et statut d'erreur
- Mécanisme de fonctionnement du MMU
  - Traduction d'adresses virtuelles en physiques
  - Table de traduction et référence de section/page
  - Exemple de gestion de la mémoire dans le kernel

Objectifs

- Support de processus en mode utilisateur
  - Mode utilisateur et appel système pour exécuter code kernel
  - Mapping des adresses virtuelles vers les physiques
- Support du kernel pour les appels systèmes
  - Nouvelle version de la structure `PROC`
  - Fonction d'initialisation du kernel
  - Fonction de routage des appels systèmes

Mode utilisateur

Processus en mode utilisateur (1)

- Simple kernel uniprocesseur pour gestion de processus
  - Création dynamique, synchronisation et communication
  - Nombre fixe de processus exécuté dans même espace d'adresses
  - Processus orienté évènements avec interruptions
- Protection du système avec espace d'adresses virtuelles
  - Utilisation de support hardware (MMU) pour traduction
  - Extension du kernel pour support de deux modes d'exécution
  - Kernel non préemptif et utilisateur protégé et préemptif

Appel système

- Entrée d'un processus du mode utilisateur en mode kernel

  Via une exception, une interruption ou un appel système

- Entrée en mode kernel via un appel système
  - Mécanisme pour exécuter une fonction kernel en utilisateur
  - Retour en mode utilisateur après exécution avec valeur retour

Mapping adresses virtuelles

- Chargement du kernel en mémoire basse lors du démarrage

  Typiquement vers l'adresse physique 0 ou 16 Ko

- Activation de la traduction d'adresses en configurant le MMU
  - Un adressage sur 32 bits permet un espace virtuel de 4 Go
  - Moitié moitié pour espace en mode kernel et en mode user
- Mapping en Kernel Mapped Low (KML) ou KMH
  - One-to-one mapping pour kernel en KML permet compile-link
  - Démarrage direct de l'exécution du kernel sans MMU
  - Aucune différence du point de vue de la protection mémoire

Processus en mode utilisateur (2)

- Deux modes d'exécution des processus Kmode et Umode
  - Espace d'adresse virtuel (VA) pour chaque mode suivant KML
  - VA Kmode de 0 à quantité de RAM, UMode démarre à 2 Go
- Plusieurs fichiers pour créer une image d'un processus user
  - Fichier `u.us` commun à tous les programmes
  - Fichier `ucode.c` avec fonction d'interface appel système
  - Fichier `u1.c` avec le corps du programme user

Point d'entrée

- Trois fonctions communes à tous les programme user

  Set up environnement d'exécution par le kernel avant `entryPoint`

- Appel système passe en mode privilégié (SVC) grâce à `swi`

  Numéro de l'appel suivi de trois paramètres

```
1  .global entryPoint, main, syscall, getcsr, getAddr
2  .text
3  .code 32
4  .global _exit
5  entryPoint:       // si main() retourne, appelle _exit()
6     bl main
7     bl _exit
8  syscall:          // syscall(a, b, c, d) : paramètres en r0-r3
9     swi #0
10    mov pc, lr
11 get_cpsr:         // registre de statut actuel du CPU
12    mrs r0, cpsr
13    mov pc, lr
```

Interface appel système (1)

- Librairie partagée avec fonctions interface d'appel système

  Typiquement pré-compilé comme partie de la librairie de link

- Programme d'exemple avec des commandes utilisateur

  Numéro de l'appel système choisi par le designer du système

```
1  // Appels systèmes vers le kernel
2  int getpid()          { return syscall(0, 0, 0, 0); }
3  int getppid()         { return syscall(1, 0, 0, 0); }
4  int ps()              { return syscall(2, 0, 0, 0); }
5  int chname(char *s) { return syscall(3, s, 0, 0); }
6
7  // Fonctions commande mode utilisateur
8  int ugetpid()
9  {
10    int pid = getpid();
11    uprintf("pid = %d\n", pid);
12 }
13
14 // ...
```

Interface appel système (2)

- Pas d'accès direct à l'espace E/S kernel pour processus user

Opérations d'E/S basiques en mode user par appel système

```
1  // ...
2
3  int ups()
4  {
5     ps();
6  }
7
8  // Opérations d'E/S
9  int ugetc()          { return syscall(90, 0, 0, 0); }
10 int uputc(char c)    { return syscall(91, c, 0, 0); }
```

Corps principal

- Template de base pour un programme en mode utilisateur

Affichage du mode, puis attente et exécution de commandes

```
1  #include "ucode.h"
2  int main()
3  {
4     int i, pid, ppid, mode;
5     char line[64];
6     mode = get_cpsr() & 0x1F;
7     printf("CPU mode=%x\n", mode);
8     pid = getpid();
9     ppid = getppid();
10    while (1) {
11       printf("This is process %d in Umode at %x: parent=%d\n", pid,
          &entryPoint, ppid);
12       uprintf("input command :"); ugetline(line); uprintf("\n");
13       if (!strcmp(line, "getpid")) {
14          ugetpid();
15       }
16       // ...
17    }
18 }
```


Support kernel

Support kernel

- Kernel système composé de plusieurs éléments
  - Gestionnaire d'interruptions, drivers de périphériques, E/S
  - Fonction de manipulation de files et de gestion de processus
- Trois parties principales au code du kernel
  - Représentation des processus avec une structure `PROC`
  - Gestionnaire du reset pour initialiser la machine
  - Code du kernel à proprement parler

Structure PROC (1)

- Représentation d'un processus avec une structure PROC

  Plusieurs champs généraux pour gérer l'exécution du processus

```
1  #define NBPROC      9
2  #define FREE        0
3  #define READY       1
4  #define SLEEP       2
5  #define BLOCK       3
6  #define ZOMBIE      4
7  #define SSIZE    1024
8
9  typedef struct proc {
10    struct proc *next;
11    int *ksp;               // Kmode sp lorsque pas exécuté
12    int status;
13    int priority;
14    int pid;
15    int event;              // Évènement sur lequel le processus dort
16    int kstack[SSIZE];
17    // ...
18 } PROC;
```

Structure PROC (2)

- Série de champs pour la gestion du mode user
  - Sauvegarde des infos du Umode avant appel système
  - Pointeur table des pages définit espace d'adresses virtuels

```
1  typedef struct proc {
2     // ...
3     int *usp;              // Umode sp lors d'un syscall
4     int *upc;              // Umode pc lors d'un syscall
5     int *ucpsr;            // Umode cpsr
6     int *pgdir;            // pointeur table des pages niveau 1
7     int ppid;              // pid du parent
8     struct proc *parent;   // pointeur vers le PROC du parent
9     int exitCode;          // code de retour
10    char name[64];
11 } PROC;
```

Code du kernel (1)

- Code du kernel structuré en cinq parties
  - Gestionnaire de reset
  - Points d'entrées des gestionnaires d'IRQ et exception
  - Switching du contexte des tâches, switch du pgdir
  - Entrée SVC, routage appel système et retour Umode
  - Fonctions utilitaires
- Appels de routines écrites en C pour gérer divers éléments

  mkPtable, *main*, *irq_handler*, *data_chandler*, *scheduler*, *svc_handler*

Code du kernel (2)

- Définition de plusieurs fonctions de bas niveau

  S'occupe de toute l'initialisation de bas niveau du kernel

```
1  .text
2  .code 32
3  .global reset_handler, vectors_start, vectors_end
4  .global proc, procsize
5  .global tswitch, scheduler, running, goUmode, switchPgdir
6  .global int_off, int_on, lock, unlock, get_cpsr
7
8  // ...
```

Code du kernel (3)

- Implémentation des fonctions kernel et structure de données

  Initialisation du kernel, scheduler, appel système...

- Fonction `reset_handler` appelle `main` qui appelle `kernel_init`

  Lors du démarrage du système, pour initialiser le kernel

```
1  PROC proc[NPROC], *freeList, *readyQueue, *sleepList, *running;
2  int procsize = sizeof(PROC);
3  char *pname[NPROC] = {"sun", "mercury", ..., "uranus", "neptune"};
4
5  int kernel_init() { /* ... */ }
6  int scheduler() { /* ... */ }
7  int svc_handler(volatile int a, int b, int c, int d) { /* ... */ }
```

Initialisation du kernel (1)

- Création et initialisation de toutes les structures PROC

  Tous les processsus sont FREE initialement, et sans parent

```
1  int kernel_init()
2  {
3      int i, j;
4      PROC *p; char *cp;
5      int *MTABLE, *mtable, paddr;
6      printf("kernel_init()\n");
7      for (i = 0; i < NPROC; i++) {
8          p = &proc[i];
9          p->pid = i;
10         p->status = FREE;
11         p->priority = 0;
12         p->ppid = 0;
13         p->parent = 0;
14         strcpy(p->name, pname[i]);
15         p->next = p + 1;
16         p->pgdir = (int *) (0x600000 + p->pid * 0x4000);
17     }
18     proc[NPROC-1].next = 0;
19     // ...
20 }
```

Initialisation du kernel (2)

- Initialisation des listes de processus et démarrage *P*0

  *Chargement de P*0 *en Kmode avec priorité la plus basse* 0

```
1  int kernel_init()
2  {
3      // ...
4
5      // Initialisation des listes de processus
6      freeList = &proc[0];
7      readyQueue = 0;
8      sleepList = 0;
9
10     // Création et exécution de P0
11     running = get_proc(&freeList);
12     running->status = READY;
13     printList(freeList);
14     printQ(readyQueue);
15
16     // ...
17 }
```

Initialisation du kernel (3)

- Création des tables de pages des processus

  Espace VA kernel en KML, tables des pages en 6 Mo de 16 Ko

```
1  int kernel_init()
2  {
3     // ...
4
5     MTABLE = (int *) 0x4000;
6     mtable = (int *) 0x600000;
7     for (i = 0; i < 64; i++) {          // 64 PROC mtables
8        for (j = 0; j < 2048; j++) {
9           mtable[j] = MTABLE[j];       // copie 2048 entrées basses de MTABLE
10       }
11       mtable += 4096;                 // avancer au 16 Ko suivant
12    }
13    mtable = (int *) 0x600000;          // PROC mtables commencent à 6 Mo
14    for (i = 0; i < 64; i++) {
15       for (j = 2048; j < 4096; j++) {
16          mtable[j] = 0;               // mise à zéro 2048 entrés hautes
17       }
18       if (i)                          // attributs de page=0xC3E,AP=11;domain=1
19          mtable[2048] = (0x800000 + (i-1) * 0x100000 | 0xC3E;
20       mtable += 4096;
21    }
22 }
```

Ordonnanceur

- Choisir un processus pour lui donner accès au CPU

  Changer le contexte si c'est un nouveau processus

```
1  int scheduler()
2  {
3      PROC *old = running;
4      if (running->status == READY)
5          enqueue(&readyQueue, running);
6      running = dequeue(&readyQueue);
7
8      // Charger nouveau pgdir, flusher TLB et caches I&D
9      if (running != old)
10         switchPgdir((u32) running->pgdir);
11 }
```

Appel système (1)

- Définition des fonctions kernel pour appel système

  Fonctions vers lesquelles seront routés les appels systèmes

```
1  int kgetpid() { return running->pid; }
2  int kgetppid() { return running->ppid; }
3  int kchname(char *s)
4  {
5     // fetch *s from Umode;
6     strcpy(running->name, s);
7  }
8  char *pstatus[] = {"FREE", "READY", "SLEEP", "BLOCK", "ZOMBIE"};
9  int kps()
10 {
11    int i; PROC *p;
12    for (i = 0; i < NPROC; i++) {
13       p = &proc[i];
14       printf("proc[%d]: pid=%d ppid=%d", i, p->pid, p->ppid);
15       if (p == RUNNING)
16          printf("RUNNING");
17       else
18          printf("%s", pstatus[p->status]);
19       printf(" name=%s\n", p->name);
20    }
21 }
```

Appel système (2)

- Gestionnaire d'appel système avec routage vers fonction kernel

Paramètres dans registres R0-R3, valeur de retour dans `kstack`

```
1  int svc_handler(volatile int a, int b, int c, int d)
2  {
3     int r = -1;
4     switch (a) {
5        case 0:  r = kgetpid();           break;
6        case 1:  r = kgetppid();          break;
7        case 2:  r = kps();               break;
8        case 3:  r = kchname((char *) b); break;
9        case 90: r = kgetc() & 0x7F;      break;
10       case 91: r = kputc(b);            break;
11       default: printf("invalid syscall %d\n", a);
12    }
13    running->kstack[SSIZE-14] = r;
14 }
```

Crédits

- https://www.flickr.com/photos/ben8472/4212445717
- https://www.flickr.com/photos/calamity_sal/3670752202