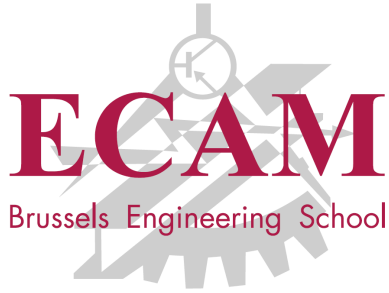

I403C Systèmes d'exploitation temps réel

Séance 3

Gestion des processus d'un système embarqué

Sébastien Combéfis

2017–2018

Rappels

- Trois modèles de structure de programmes embarqués
  - Modèle super-loop simpliste
  - Modèle orienté évènements et orienté interruptions
  - Modèles basés sur les processus et caractéristiques
- Méthode de design d'un programme pour système embarqué
  - Machine à états finis et construction d'une FSM
  - Algorithme de minimisation d'une FSM
  - Transformation d'une FSM en un programme C

Objectifs

- Comprendre la notion de processus sur une machine
  - PCB et gestion de processus statiques dans un kernel
  - Gestion dynamique de processus
- Ordonnanceur de processus sur un système embarqué

  Terminologie et caractéristiques
- Mécanismes de synchronisation de processus

  Sleep/Wakeup, sémaphore et driver de périphériques


Multitâche

Multitâche

- Exécution de plusieurs tâches indépendantes en même temps

  Parallélisme ou non selon qu'il y a un ou plusieurs processeurs

- Multiplexage du CPU sur uniprocesseur (UP)
  - Switching du CPU d'une tâche vers une autre
  - Impression de parallélisme si le switching est assez rapide
- Exécution parallèle réelle sur multiprocesseur (MP)

  Chaque processeur peut également faire du multitâche

Processus

- Exécution concurrente de plusieurs processus
  - Contrôlé par le kernel du système d'exploitation (OS)
  - Ensemble de fonctions de gestion des processus
- Un processus est l'exécution d'une image d'exécution
  - Zone mémoire contenant le code, les données et la pile
  - Utilisation de ressources (CPU, mémoire, E/S)

Process Control Block (PCB)

- Processus représenté par un Process Control Block dans kernel

  Contient toute l'information associée à un processus

- Exemple de structure simple pour représenter un processus
  - Pointeur vers le processus suivant pour faire liste chainée
  - Pointeur vers la pile lorsqu'il n'est pas exécuté
  - Pile d'exécution du processus

```
1  typedef struct proc {
2     struct proc *next;
3     int *ksp;
4     int kstack[1024];
5  } PROC;
```

Changement de contexte (1)

- Nécessite d'un changement de contexte entre deux processus

  Changement de l'environnement d'exécution des processus

- Trois étapes lors d'un changement de contexte

  1. Sauvegarde des informations du processus actuel
  2. Exécution de l'ordonnanceur pour choisir nouveau processus
  3. Reprise de l'exécution du processus choisi

Changement de contexte (2)

- Fonction `tswitch` fait un changement de contexte

  Sauvegarde puis restauration de l'état d'un processus

```
1  .global main, proc0, procsize
2  .global reset_handler, tswitch, scheduler, running
3
4  // ...
5
6  tswitch:
7  SAVE:
8    STMFD sp!, {r0-r12, lr}     // save registers
9    LDR r0, =running            // r0 = &running
10   LDR r1, [r0, #0]            // r1->runningPROC
11   STR sp, [r1, #4]            // running->ksp = sp
12 FIND:
13   BL scheduler                // scheduler()
14 RESUME:
15   LDR r0, =running
16   LDR r1, [r0, #0]
17   LDR sp, [r1, #4]            // sp = running->ksp
18   LDMFD sp!, {r0-r12, lr}     // restore registers
19   MOV pc, lr                  // return
```

Démarrage du programme

- Chargement du `.bin` en mémoire et démarrage fonction `main`
  1. Démarrage du kernel et lancement du processus d'initialisation
  2. Appel de `tswitch` et de l'ordonnanceur
  3. Retour à la fonction `main`

```
1  PROC proc0, *running;
2
3  int scheduler() { running = &proc0; }
4
5  main()
6  {
7     running = &proc0;
8     printf("call tswitch()\n");
9     tswitch();
10    printf("back to main()\n");
11 }
```

Système multitâches (1)

- Liste circulaire de plusieurs processus

  Seront exécutés l'un après l'autre, toujours dans le même ordre

- Mémorisation d'un tableau de PCB pour les processus

  Processus créés de manière statique au démarrage

```
1  #define NPROC 5
2  #define SSIZE 1024
3
4  typedef struct proc {
5     struct proc *next;
6     int *ksp;
7     int pid;
8     int kstack[SSIZE];
9  } PROC;
10
11 PROC proc[NPROC], *running;
```

Système multitâches (2)

- Instructions d'un processus représentées par la fonction `body`

Demande à l'utilisateur un caractère, puis rend la main

```
1  int body()
2  {
3     char c;
4     printf("proc %d resume to body()\n", running->pid);
5     while (1) {
6        printf("proc %d in body(), input a char : ", running->pid);
7        c = kgetc();
8        printf("%c\n", c);
9        tswitch();
10    }
11 }
12
13 int scheduler()
14 {
15    printf("proc %d in scheduler\n", running->pid);
16    running = running->next;
17    printf("next running = %d\n", running->pid);
18 }
```

Système multitâches (3)

- Démarrage du kernel initialise tous les processus
  - Initialisation des structures `PROC`, et démarrage processus initial
  - Simulation que le processus vient de sortir de `body`

```
1  int kernel_init()
2  {
3      int i, j;
4      PROC *p;
5      printf("kernel_init()\n");
6      for (i = 0; i < NPROC; i++) {
7          p = &proc[i];
8          p->pid = i;
9          for (j = 1; j < 15; j++)
10             p->kstack[SSIZE-j] = 0;          // saved registers = 0
11         p->kstack[SSIZE-1] = (int) body;    // resume point = body
12         p->ksp = &(p->kstack[SSIZE-14]);    // saved ksp
13         p->next = p + 1;
14     }
15     proc[NPROC-1].next = &proc[0];          // boucler liste circulaire
16     running = &proc[0];
17 }
```

Système multitâches (4)

- Processus 0 est handcrafted et est le point d'entrée

  Demande un caractère au clavier qui est affiché sur le LCD

```
1  int main()
2  {
3      char c;
4      fbuf_init();
5      kbd_init();
6      printf("Welcome to WANIX in Arm\n");
7      kernel_init();
8      while (1) {
9          printf("proc 0 running, input a char : ");
10         c = kgetc();
11         printf("%c\n", c);
12         tswitch();
13     }
14 }
```

Processus dynamique (1)

- Processus dynamiques sont créés durant l'exécution

  Contrairement aux processus statiques créés par `kernel_init()`

- Ajout du statut et de la priorité dans le PCB des processus

```
1  #define NPROC 9
2  #define SSIZE 1024
3
4  typedef struct proc {
5     struct proc *next;
6     int *ksp;
7     int pid;
8     int status;
9     int priority;
10    int kstack[SSIZE];
11 } PROC;
```

Processus dynamique (2)

- Définition de deux listes de processus
  - Liste des processus libres (situation initiale)
  - File à priorité des processus prêts à être exécutés
- Allocation et déallocation des `PROC` dans la `freeList`
  - Création processus trouve un `PROC` libre et l'alloue
  - Terminaison processus déalloue le `PROC`

```
1 PROC proc[NPROC], *running, *freeList, *readyQueue;
```

Processus dynamique (3)

- Création d'un nouveau processus exécutant une fonction

  Possibilité de spécifier la priorité du processus

```
1  int kfork(int func, int priority)
2  {
3      int i;
4      PROC *p = get_proc(&freeList);
5      if (p == 0) {
6          printf("No more PROC, kfork failed\n");
7          return -1;
8      }
9      p->status = READY;
10     p->priority = priority;
11     for (i = 1; i < 15; i++)
12         p->kstack[SSIZE-i] = 0;
13     p->kstack[SSIZE-1] = func;
14     p->ksp = &(p->kstack[SSIZE-14]);
15     enqueue(&readyQueue, p);
16     printf("%d kforked a new proc %d\n", running->pid, p->pid);
17     return p->pid;
18 }
```

Processus dynamique (4)

- Terminaison d'un processus libère le PCB alloué

  Sélection d'un autre processus à exécuter par l'ordonnanceur

```
1  int kexit()
2  {
3     printf("proc %d kexit\n", running->pid);
4     running->status = FREE;
5     put_proc(&freeList, running);
6     tswitch();
7  }
8
9  int scheduler()
10 {
11    if (running->status == READY)
12       enqueue(&readyQueue, running);
13    running = dequeue(&readyQueue);
14 }
```

Ordonnancement

Terminologie (1)

- Plusieurs processus prêts dans système multitâches

  Nombre de processus exécutables > nombre de CPUs disponibles

- Ordonnancement de processus décide quel processus s'exécute

- Cinq caractéristiques principales liées à l'ordonnancement

  - Processus orienté calcul vs. orienté E/S

    Passe plus de temps sur le CPU ou à attendre des E/S

  - Temps de réponse vs. débit

    Rapidité de réponse à un évènement, processus terminés/temps

  - Ordonnancement round-robin vs. priorité dynamique

    Chaque processus à son tour ou exécution selon priorité

Terminologie (2)

- Cinq caractéristiques principales liées à l'ordonnancement
  - Préemption vs. pas de préemption

    Processus peut être retiré ou doit libérer lui-même CPU
  - Temps réel vs. temps partagé

    Temps de réponse minimal+limite de temps ou time slice garanti
- Quatre buts à atteindre par algorithme d'ordonnancement
  - Grande utilisation des ressources systèmes, notamment CPU...
  - Réponse rapide pour processus interactifs ou temps réel
  - Temps de complétion garanti pour processus temps réel
  - Équité pour tous les processus pour bon débit...

Ordonnancement de processus

- Ensemble d'algorithmes pour choisir le processus à exécuter
  - Fonctions qui implémentent une politique d'ordonnancement
  - Pas de localisation à un seul endroit dans le code d'un OS
  - Ordonnanceur de processus contient données et code
- Plusieurs interventions possibles pour l'ordonnanceur
  - Lorsqu'un processus se suspend lui-même ou se termine
  - Lorsqu'un processus suspendu redevient exécutable
  - À l'intérieur du gestionnaire d'évènements du timer

Ordonnancement de systèmes embarqués

- Deux buts importants pour les systèmes embarqués
  - Réponse rapide aux évènements externes
  - Garantie sur le temps d'exécution
- Ajout d'une priorité aux processus pour l'ordonnanceur

  Utilisation de round robin pour processus de même priorité
- Politique d'ordonnancement est souvent non-préemptive
  - Car processus typiquement dans le même espace d'adresses
  - Exécution jusque libération volontaire (sleep, suspension, yield)

Synchronisation

Synchronisation de processus

- Ordonnancement de processus préemptifs très difficile

  Nécessite mécanismes protection mémoire pour accès concurrents

- Garantir intégrité des données partagées
  - Crucial lorsqu'on est en environnement concurrent
  - Important pour les structures de données partagées
  - Modification atomique des données en section critique

- Plusieurs mécanismes de synchronisation existants

  Sleep/Wakeup, sémaphore...

Sleep et Wakeup (1)

- Mécanisme Sleep/Wakeup est le plus simple possible
  - Utilisé dans le kernel Unix original
  - Processus s'endort lorsqu'il attend ressource indisponible
  - Processus est réveillé lorsque la ressource devient disponible
- Mise à disposition de deux fonctions `sleep` et `wakeup`
  - Les deux fonctions doivent atomiques par rapport au processus

    Doit pas être wakeup alors qu'il est en train de rentrer en sleep
  - Sur un uniprocesseur, il suffit de désactiver les interruptions

    Pour ne pas que le processus soit détourné vers un gestionnaire

Sleep et Wakeup (2)

- Algorithmes de `sleep` et `wakeup` appelés par un processus

  Ajout d'un champ `event` à la structure `PROC`

```
1  int sleep(int event)
2  {
3     int SR = int_off();
4     running->event = event;
5     running->status = SLEEP;
6     tswitch();
7     int_on(SR);
8  }
9
10 int wakeup(int event)
11 {
12    int SR = int_off();
13    for each PROC *p do {
14       if (p->status == SLEEP && p->event == event) {
15          p->status = READY;
16          enqueue(&readyQueue, p);
17       }
18    }
19    int_on(SR);
20 }
```

Driver de périphérique

- Trois parties dans un driver de périphérique
  - Partie basse contient le gestionnaire d'interruptions
  - Partie haute est celle appelée par le programme principal
  - Partie donnée avec buffer E/S et variables de contrôle
- Attente active du programme principal pour buffer E/S
  - Même avec le mécanisme d'interruptions, on a donc du polling
  - E/S par polling gaspillent CPU en environnement multitâches

Driver de périphérique en polling (1)

- Structure de données représentant le clavier

  Buffer d'entrée et variables de contrôle, au milieu du driver

- Variables de contrôle identifient caractères dans buffer

  Nombre de caractères avec `data`, début et fin avec `head` et `tail`

```
1  typedef struct kbd {
2      char *base;
3      char buf[BUFSIZE];
4      int head, tail, data;
5  } KBD;
6
7  KBD kbd;
```

Driver de périphérique en polling (2)

- Fonction de base `kgetc` dans partie haute du driver

  Attente active de la disponibilité d'un caractère

```
1  int kgetc()
2  {
3      char c;
4      KBD *kp = &kbd;
5      unlock();
6      while (kp->data == 0);    // attente active de données
7      lock();
8        c = kp->buf[kp->tail++];
9        kp->tail %= BUFSIZE;
10       kp->data--;
11     unlock();
12     return c;
13 }
```

Driver de périphérique en polling (3)

- Gestionnaire d'interruptions partie basse du driver

  Stockage de la touche pressée dans le buffer

```
1  kbd_handler()
2  {
3      struct KDB *kp = &kbd;
4      char scode = *(kp->base + KDATA);
5      if (scode & 0x80)
6          return;
7      if (data == BUFSIZE)
8          return;
9      c = unsh[scode];
10     kp->buf[kp->head++] = c;
11     kp->head %= BUFSIZE;
12     kp->data++;
13 }
```

Driver de périphérique sans polling (1)

- Endort le processus lorsqu'il n'y a pas de caractères disponibles

Réactivation des interruptions juste avant de s'endormir

```
1  int kgetc()
2  {
3      char c;
4      KBD *kp = &kbd;
5      while (1) {
6          lock();
7          if (kp->data == 0) {
8              unlock();
9              sleep(&kp->data);
10         }
11     }
12     c = kp->buf[kp->tail++];
13     kp->tail %= BUFSIZE;
14     kp->data--;
15     unlock();
16     return c;
17 }
```

Driver de périphérique sans polling (2)

- Réveille les processus endormis, s'il y en a en attente

  De nouveau pas d'interférence entre gestionnaire et processus

```
1  kbd_handler()
2  {
3      struct KDB *kp = &kbd;
4      char scode = *(kp->base + KDATA);
5      if (scode & 0x80)
6          return;
7      if (data == BUFSIZE)
8          return;
9      c = unsh[scode];
10     kp->buf[kp->head++] = c;
11     kp->head %= BUFSIZE;
12     kp->data++;
13     wakeup(&kp->data);
14 }
```

Gestion de ressources

- Utilisation de Sleep/Wakeup pour gérer des ressources

  Acquisition et libération d'une ressource partagée

```
1  int acquire_resource()
2  {
3      while (1) {
4          int SR = int_off();
5          if (res_status == 0) {
6              res_status = 1;
7              break;
8          }
9          sleep(&res_status);
10     }
11     int_on(SR);
12     return OK;
13 }
14
15 int release_resource()
16 {
17     int SR = int_off();
18     res_status = 0;
19     wakeup(&res_status);
20     int_on(SR);
21     return OK;
22 }
```

Défaut de Sleep/Wakeup

- Un évènement est juste une simple valeur

  Aucun espace mémoire pour enregistrer occurrence évènement

- Processus dort, puis réveillé, puis dort... *(sleep-first-wakeup-later)*
  - Ordre d'exécution toujours possible de garantir sur UP
  - Problème avec les MP car exécution parallèle de processus

- Problème d'efficacité pour gérer les ressources

  Retentative d'accès à la ressource après un réveil

Sémaphore

- Résoudre défauts de Sleep/Wakeup avec sémaphores

  Un sémaphore compteur possède une valeur

- Structure de données avec plusieurs champs
  - Opérations par un seul processus à la fois avec `spinlock`
  - Seulement nécessaire pour systèmes MP, pas pour UP

```
1 typedef struct semaphore {
2    int spinlock;
3    int value;
4    PROC *queue;
5 } SEMAPHORE;
```

Opération sur sémaphore (1)

- Opération P (probeer) teste s'il reste une « entrée » disponible

  Bloque le processus en attente s'il n'y a plus d'entrée

```
1  int P(struct semaphore *s)
2  {
3      int SR = int_off();
4      s->value--;
5      if (s->value < 0)
6          block(s);
7      int_on(SR);
8  }
9
10 int block(struct semaphore *s)
11 {
12     running->status = BLOCK;
13     enqueue(&s->queue, running);
14     tswitch();
15 }
```

Opération sur sémaphore (2)

- Opération V (verhoog) libère une « entrée »

  Débloque un processus en attente s'il y en a

```
1  int V(struct semaphore *s)
2  {
3      int SR = int_off();
4      s->value++;
5      if (s->value >= 0)
6          signal(s);
7      int_on(SR);
8  }
9
10 int signal(struct semaphore *s)
11 {
12     PROC *p = dequeue(&s->queue);
13     p->status = READY;
14     enqueue(&readyQueue, p);
15 }
```

Application des sémaphores

- Semaphore lock pour protéger section critique (CR)

  Protection avec un sémaphore de valeur initiale 1

- Un seul processus à la fois se trouvera dans la CR

- Mutex lock possède en plus un propriétaire

  Ne peut être déverouiller que par son propriétaire

```
1  struct semaphore s = 1;
2
3  P(s);
4  // ... CR ...
5  V(s);
```

Crédits

- https://www.flickr.com/photos/dtcmastercrew/4192967679
- https://www.flickr.com/photos/philliecasablanca/3353918641
- https://www.flickr.com/photos/woueb/3276459266