RTS6: Conception et programmation de Systèmes Embarqués

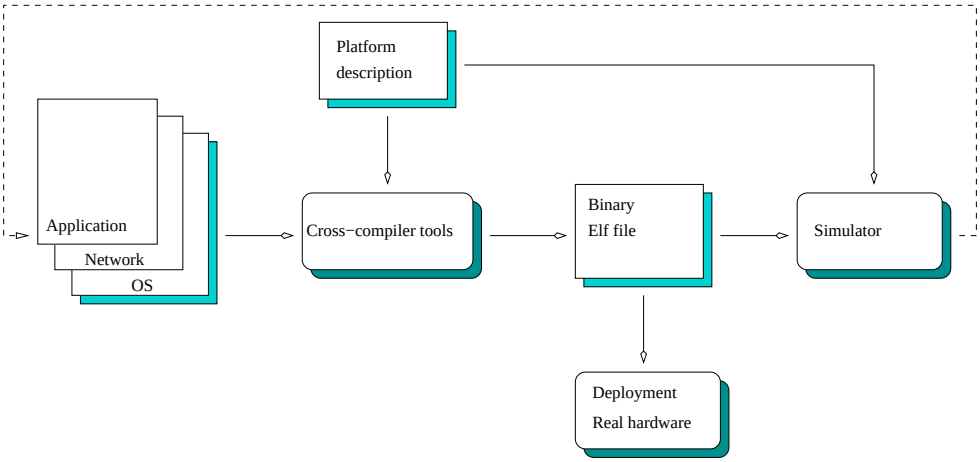
cours 2: compilation et systèmes d'exploitation

Antoine Fraboulet, Fabrice Jumel, Lionel Morel, Tanguy Risset
antoine.fraboulet@insa-lyon.fr
Lab CITI, INSA de Lyon

Présentation

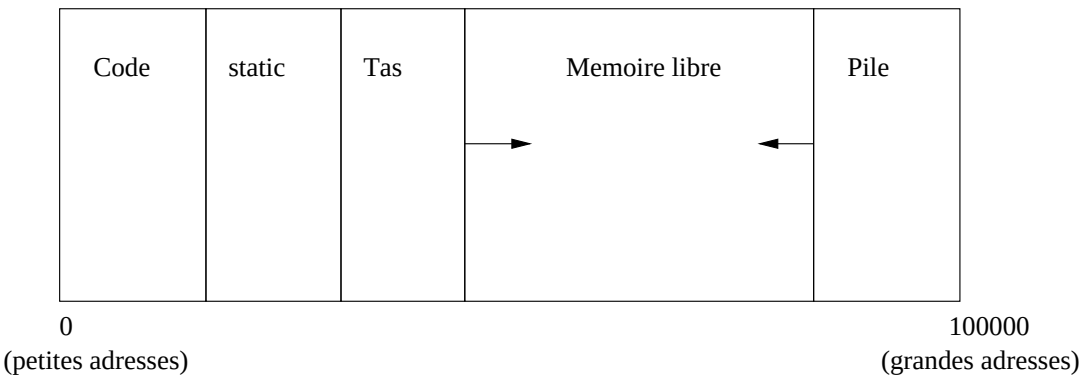
- Principes de compilation
- Modèle de programmation et d'exécution
- Aperçu des principaux systèmes
- Programmation de pilotes

Compilation



Pile d'exécution

- Le mécanisme de transfert de contrôle entre les procédures est implémenté grâce à la *pile d'exécution*.
- Le programmeur à cette vision de la mémoire virtuelle:

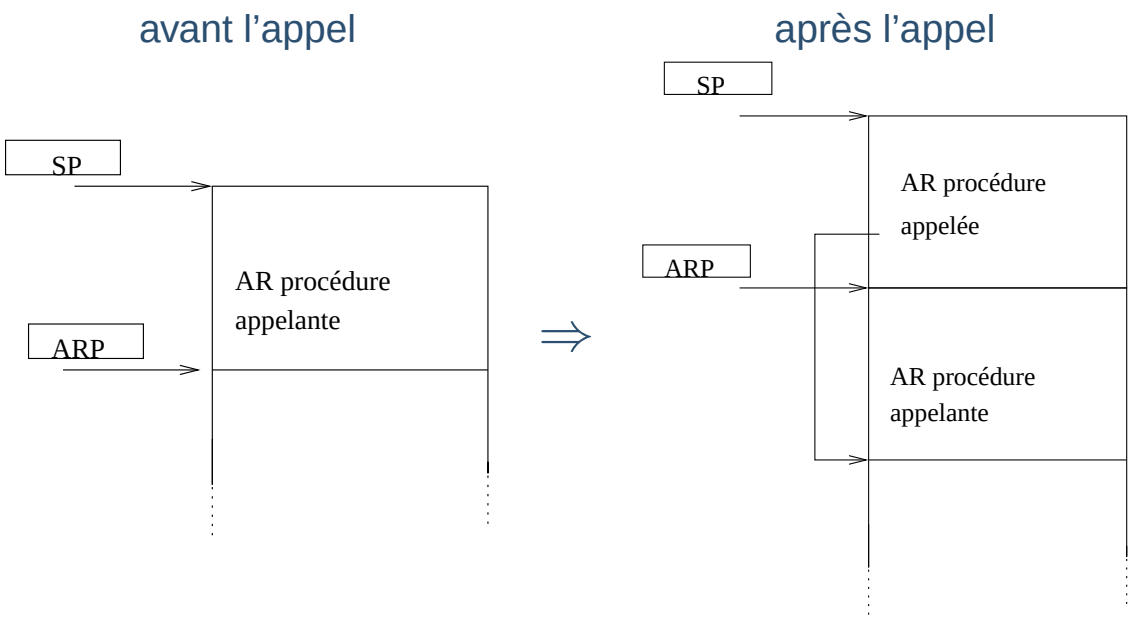


- Le tas (*heap*) est utilisé pour l'allocation dynamique.
- La pile (*stack*) est utilisée pour la gestion des contextes des procédures (variable locales, etc.)

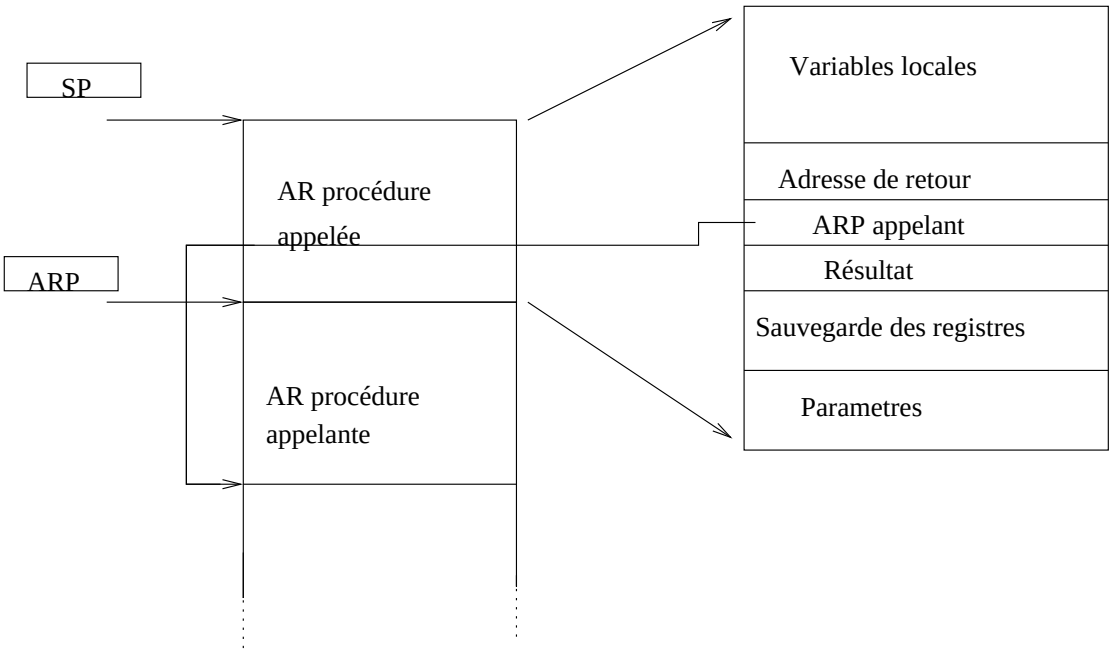
Enregistrement d'activation

- Appel d'une procédure: empilement de l'*enregistrement d'activation* (AR pour *activation record*).
- L'AR permet de mettre en place le *contexte* de la procédure.
- Cet AR contient
  - L'espace pour les variables locales déclarées dans la procédure
  - Des informations pour la restauration du contexte de la procédure appelante:
    - Pointeur sur l'AR de la procédure appelante (ARP ou FP pour *frame pointeur*).
    - Adresse de l'instruction de retour (instruction suivant l'appel de la procédure appelante).
    - Éventuellement sauvegarde de l'état des registres au moment de l'appel.

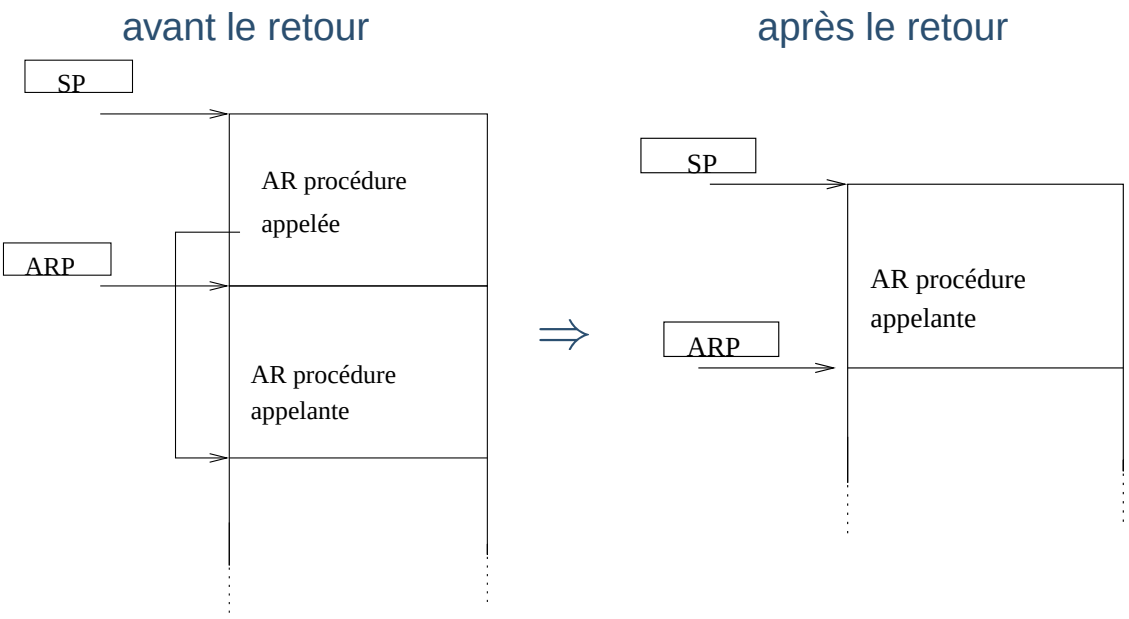
Appel de procédure: état de la pile



Contenu de l'AR



Retour de procédure: état de la pile



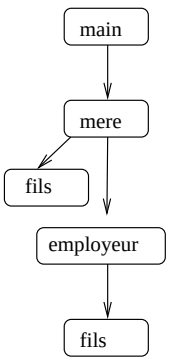
Lien statique et Lien dynamique

- Considérons une procédure `employeur` qui appelle une procédure `fils`.
- Dans l'AR de `fils`, l'ARP appelant pointe sur l'AR de `employeur`.
- Ce pointeur l'ARP est quelquefois appelé le *lien dynamique*, il pointe sur l'environnement de la procédure appelante (ici `employeur`).
- Considérons maintenant la procédure `mère` dans laquelle `fils` à été déclarée.
- Dans certains langages comme Pascal, la procédure `fils` peut accéder aux variables de `mère`
- Pour cela on a besoin d'un *lien statique* qui est un pointeur sur l'environnement de la procédure ou l'on a été déclaré.

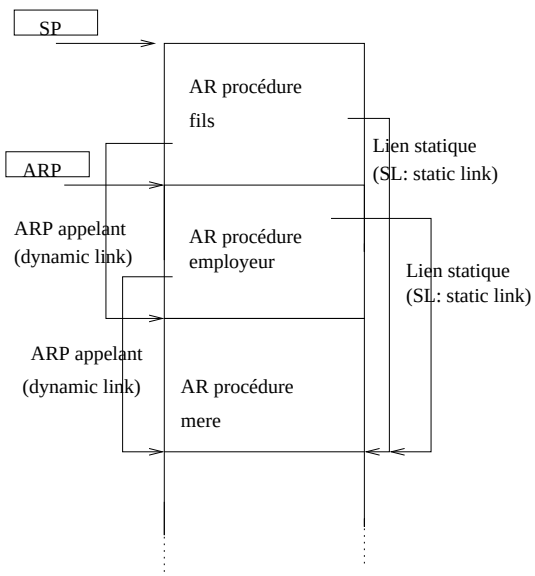
Exemple: arbre d'appel


```
procedure main();
    var y...
    procedure mere()
        var z: ...
        procedure fils();
            begin { fils()}
                ...
            end; { fils()}
        procedure employeur();
            var y....
            begin { employeur()}
                x:=...
                fils();
            end;
        begin { mere()}
            z:=1;
            fils();
            employeur();
        end;
    begin { main()}
        mere();
        ...
    end;
```

Arbre d'appel:



Pile (lors du 2^{eme} appel de fils):



Convention d'appel

- Le mécanisme de Pile n'est pas *normalisé*, son implémentation précise est déterminé par le compilateur,
- Pour pouvoir interfacer des fonctions compilées avec un certain compilateur (par exemple, les utiliser en librairie), il faut utiliser les même *conventions d'appel* (*calling convention* ou *ABI: Application Binary Interface*).
- La convention d'appel est un accord entre l'OS, le compilateur et l'ISA. En général elle est spécifiée par le compilateur.

Convention d'appel MSP430 de `gcc`

- Pour le compilateur `gcc` les conventions d'appel sont les suivantes:
  - R0 est le compteur de programme (*program counter: PC*)
  - R1 est le pointeur de pile (*Stack pointer SP*)
  - R4 est l'ARP (*Frame pointer FP*
  - Les quatre premiers arguments d'une fonction sont passés par les registres R12, R13, R14 et R15.
  - Ces quatre registres (R12, R13, R14 et R15) sont *clobbered* (ou *calle save*): il ne sont pas sauvegardés lors d'un appel de fonction.
  - Les registres R6-R11 sont *caller save* il sont sauvegardés sur la pile (si besoin) lors d'un appel de fonction
  - R5 est le pointeur d'argument: pointe sur le premier argument passé sur la pile
  - R15 est utilisé pour transmettre le résultat (R14:R15 dans le cas d'un type 32 bits)

Convention d’appel MSP430 de IAR

- R0 est le compteur de programme (*program counter: PC*)
- R1 est le pointeur de pile (*Stack pointer SP*)
- Il n’y a pas de d’ARP (directive CFI pour *call frame Information*).
- Les deux premiers arguments d’une fonction sont passés par les registre R12, R14 (+R13 et R15 pour 32 bits) ils ne sont pas sauvegardés.
- Les registres R4-R11 sont sauvegardés sauf si R8-R11 sont utilisés pour passer des paramêtres 64 bits
- R15 est utilisé pour transmettre le résultat (R14:R15 dans le cas d’un type 32 bits)
- Lors d’un appel, la fonction appelante empile dans l’enregistrement d’activation:
  - Les paramètres (sauf les deux premiers).
  - l’adresse de retour.
  - Les registres sauvegardés.

Plan

- Rappel du fonctionnement de la pile d’exécution
- **Quelques particularités de la programmation embarquées**
  - Inclure de l’assembleur dans le code source
  - Routine d’interruption
  - Manipulation au niveau bit
- Edition de liens et contrôle du binaire produit

Assembleur dans le code C avec gcc

- On peut include directement des instructions assembleur dans le code C avec la fonction `__asm__`

```
int main(void) {
        int i;
        int *p,*res;

        __asm__("mov #304, R4");        // p=0x130;
        __asm__("mov #2, @R4");         // *p=2;
        __asm__("mov #312, R4");        //p=0x138;
        __asm__("mov #5, @R4");         //*p=5;
        __asm__("mov #314, R4");
        __asm__("mov @R4, R5");         //Res=mem(0x13A);

        nop();
}
```

- Permet d'écrire des pilotes de périphériques, de contrôler la gestion des interruptions sans système d'expoitation

Assembleur dans le code C avec gcc

Présentation
Compilation pour l'embarqué
- Compilation
- Pile
- AR
- Convention d'appel
- Assembleur en C

OS: Catégories et fonctionnement
Aperçu des OS

- On peut aussi mettre explicitement des variables dans des registres sans connaître l'allocation de registres faite par le processeur
- Exemple: utilisation de la fonction `fsinx` du 68881:
  `__asm__("fsinx %1,%0" : "=f" (result) : "f" (angle));`
- %0 et %1 représentent le résultat et l'opérande de la fonction qui vont correspondre aux variables `result` et `angle` du programme C
- `"f"` est une directive indiquant à `gcc` qu'il doit utiliser des registres flottants

Assembleur dans le code C en IAR

- Cependant.... `asm` est peu utilisable car on ne peut pas préjuger de l'utilisation du registre `R4` par le compilateur dans la fonction.
- Il faut donc construire des fonction complète en assembleur et donc connaître les conventions d'appel.
- Exemples dans le document : "Mixing C and Assembler With the MSP430" (slaa140.pdf)
- Pour définir une fonction en assembleur, cette fonction doit
  - Se conformer aux conventions d'appel citées plus haut
  - Avoir un point d'entrée public.
  - Être déclarée comme fonction externe dans le code C l'appelant.

ISR pour MSP430 avec gcc


```
interrupt (PORT1_VECTOR) port1_irq_handler(void)
{
  if (P1IFG & (P1IE & (1 << 2)))
    {
      SWITCH_RED_LED();
    }
  P1IFG = 0;
}
```

ISR pour MSP430 avec IAR

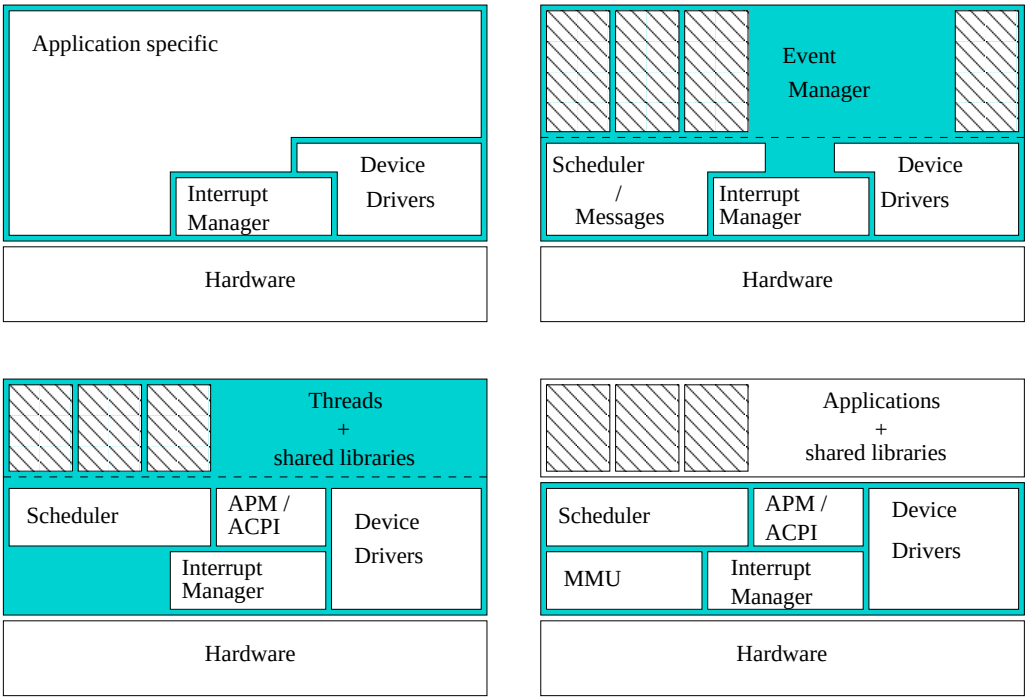

```
#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A1_ISR(void)
{
  switch (__even_in range(TAIV, 10))
     {
     case 2: P1POUT ^= 0x04;
             break;
     case 4: P1POUT ^= 0x02;
             break;
     case 10: P1POUT ^= 0x01;
     break
    }
}
```


Systèmes d'exploitation

1. Les systèmes d'exploitation peuvent aller d'une bibliothèque spécifique pour une application à un système générique type Unix.

2. Les applications sans systèmes représentent une part importante des systèmes déployés aujourd'hui.
3. Il existe tout de même deux grandes céatégories de système
   - modèle "Event driven"
   - modèle à "Thread"

Catégories de système

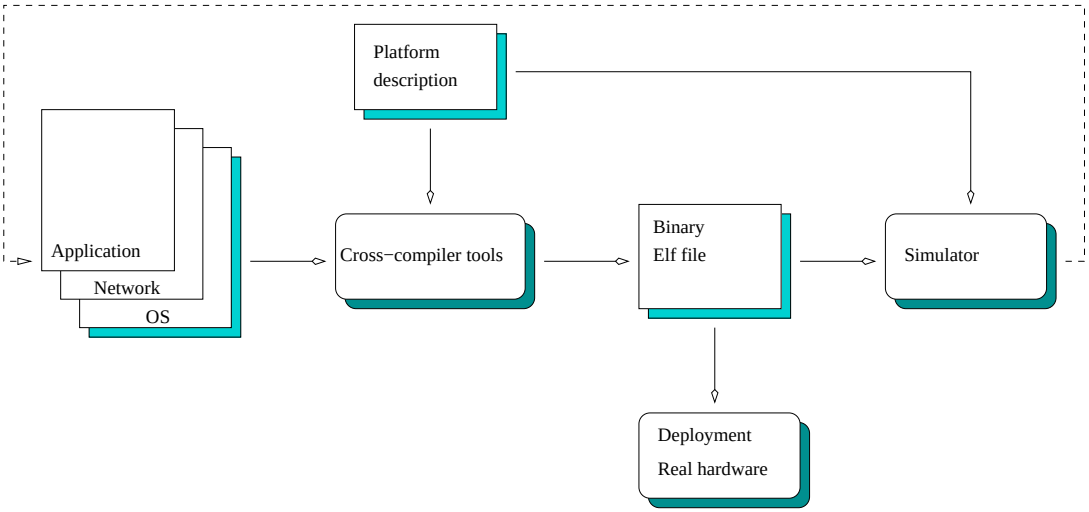


Modèles de programmation et d'exécution

1. Evénements:
   - Les événements matériels démarrent des fonctions qui s'exécutent sans interruption (*run to completion*).
   - Les changements de contexte, la gestion de pile, l'ordonnancement et la gestion de priorité sont simplifiés.
   - Exemples: TinyOS 1 & 2
2. File de programme:
   - Proche du modèle de programmation classique.
   - Mémoire partagée avec changement de contexte.
   - Les système sont plus complexes.
   - Examples: Contiki, FreeRTOS . . .

Les applications sont souvent simples. Les deux modèles sont fait pour être liés statiquement au programme et embarqués dans le système.

Environnement logiciel



Pourquoi utiliser un OS ?

Quels services demander à un OS ?

- Gestion de Tâches/Files
- Gestionnaire d'interruption
- Pile réseau intégrée
- Gestion du temps et des timers
- Gestion des modes de veille
- Pilotes de périphériques
- ...
- Environnement de programmation et outils

Gestion de la consommation

	Atmega128	PIC16	PIC18	MSP430	8051
Word Size	8 bits	8 bits	8 bits	16 bits	8 bits
Power Down	8 μA	20 μA	2.6 μA	1.8 μA	21 μA
Idle (1 MHz)	0.5 mA	220 μA	120 μA	55 μA	n/a
Idle (8 MHz)	4 mA	1.5 mA	843 μA	440 μA	n/a
Active (32 kHz)	88 μA	n/a	35 μA	19.2 μA	2.78 mA
Active (1 MHz)	2 mA	220 μA	480 μA	240 μA	4.05 mA
Active (8 MHz)	8 mA	1.5 mA	2.4 mA	1.9 mA	13.3 mA
Wakeup time	2 ms	102 μs	10 μs	6 μs	20 μs

Gestion des ressources

- Abstraction du matériel
- Gestion des périphériques
- Gestion de la consommation
- . . .

Le matériel ne peut gérer les ressources qu'à l'aide d'informations des événements *passés* de l'application.

Un système d'exploitation peut gérer le *présent*, mais seulement l'application peut gérer le *futur* et améliorer la consommation du système en fonction des événements que l'on attend.

Aperçu des systèmes

1. TinyOS
2. Contiki
3. FreeRTOS

TinyOS [Berkeley]

- Event driven kernel
- Fixed frequency with low power modes in idle periods
- Provides abstractions for
  - Communications
  - Timers
  - Storage

TinyOS 1.x main loop

```
bool TOSH_run_task(void)
{
  __nesc_atomic_t fInterruptFlags;
  uint8_t old_full;
  void (*func)(void );
  fInterruptFlags = __nesc_atomic_start();

  old_full = TOSH_sched_full;
  func = TOSH_queue[old_full].tp;
  if (func == NULL) {
      __nesc_atomic_sleep();
      return 0;
  }
  TOSH_queue[old_full].tp = NULL;
  TOSH_sched_full = (old_full + 1) & TOSH_TASK_BITMASK;

  __nesc_atomic_end(fInterruptFlags);
  func();
  return 1;
}
```


TinyOS 1.x main loop

```
int main(void)
{
  MainM$hardwareInit();
  TOSH_sched_init();
  MainM$StdControl$init();
  MainM$StdControl$start();
  __nesc_enable_interrupt();
  for (; ; ) {
    TOSH_run_task();
  }
}
```

TinyOS 2.x main loop (1)

```
void McuSleepC$McuSleep$sleep(void)
{
  uint16_t temp;
  if (McuSleepC$dirty) {   /* dirty bit */
    McuSleepC$computePowerState();
  }
  temp = 0x0008 |
    McuSleepC$msp430PowerBits[McuSleepC$powerState];
  __asm volatile ("bis  %0, r2":  :  "m"(temp));
  __nesc_disable_interrupt();
}
```


TinyOS 2.x main loop (2)

TinyOS 2.x main loop (3)

```
int main(void)
{
  __nesc_atomic_t __nesc_atomic =
  __nesc_atomic_start();
     RealMainP$Scheduler$init();
     RealMainP$PlatformInit$init();
     while (RealMainP$Scheduler$runNextTask()) ;
     RealMainP$SoftwareInit$init();
     while (RealMainP$Scheduler$runNextTask()) ;
  __nesc_atomic_end(__nesc_atomic);

  __nesc_enable_interrupt();
  RealMainP$Boot$booted();
  RealMainP$Scheduler$taskLoop();
  return -1;
}
```


Contiki [SICS, Sweden]

- Modèle mixte, orienté événements
  - Ev. “run to completion”
  - Thread et protothread
  - Pile d’exécution unique
  - Gestion préemptive
- Pile IP embarquée (uIP + uIPv6)
- Le système gère le mode LPM0 mais ne prévoit rien pour les périphériques.

Contiki 2.x main loop

```
int main(void)
{
  init();
  while (1) {
    do {
      /* Reset watchdog. */
    } while(process_run() > 0);

    /* Idle processing. */
    int s = splhigh();   /* Disable interrupts. */
    if(process_nevents() != 0) {
      splx(s);           /* Re-enable interrupts. */
    } else {
      /* Re-enable interrupts and go to sleep atomically. */
      _BIS_SR(GIE |SCG0 |CPUOFF);     /* LPM1 sleep. */
    }
  }
  return 0;
}
```


FreeRTOS 4.x

- Opération de base
  - Gestion de tâches
  - Ordonnancement
  - Timers & Synchronisation
- Utilisation de priorités
- Ordonnancement préemptif
- Primitives de synchronisation
- Piles séparées par thread
- Tâche “idle” de plus faible priorité
- Pas de pilote de périphérique

FreeRTOS 4.x main loop

```
interrupt (TIMERA0_VECTOR) prvTickISR( void )
{
  portSAVE_CONTEXT();
  vTaskIncrementTick();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();
}

int main( void )
{
  prvSetupHardware();
  vParTestInitialise();
  xTaskCreate( vErrorChecks, "Check",
        configMINIMAL_STACK_SIZE,
        NULL, mainCHECK_TASK_PRIORITY,
        NULL );
  vTaskStartScheduler();
  return 0;
}
```


Aperçu des OS

- De nombreux OS sont disponibles
- La plupart d'entre eux restent classiques et utilisent des process et des threads.
- Beaucoup de systèmes se veulent "low power" mais en réalité uniquement pour les instructions
- Peu d'efforts ont été faits pour la gestion de la consommation des périphériques.
- L'intégration de la plateforme complète est obligatoire.

Programmation bas niveau et SE

1. Choix d'un SE
   - Modèle de programmation
   - Environnement d'exécution
   - Outils disponibles
   - Pas de système ??
2. Choix d'implantation
   - Gestion de la mémoire
   - Cycles d'activité / modes de veille
   - Contrôle (Interruption / attente active)