Informatique embarquée 8/16

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

transparents à `jmfriedt.free.fr`

12 février 2018

Au-delà du C ...

Programmation monolithique en C : un binaire contient toutes les fonctions
⇒ spécification des échanges, garantir que chaque fonction fait ce qu'elle est supposée faire (test unitaire)
Alternative :

- chaque programmeur développe une tâche
- un superviseur cadence l'appel à ces tâches (ordonnanceur)
- **problème :** plusieurs tâches, une ressource : comment garantir l'intégrité des accès ?
- **problème :** les tâches veulent échanger des données : comment garantir l'intégrité des données ?
- portabilité du code : abstraction du matériel (pilotes) + fournir des interfaces homogènes ("capteur de température", "port de communication asynchrone"...)

⇒ environnement exécutif et systèmes d'exploitation (liste d'appels systèmes faisant le lien entre espace utilisateur et superviseur – noyau)

Au-delà du C ... les environnements exécutifs

- ajout d'une couche d'abstraction supplémentaire (assembleur – C – noyau)
- les environnement exécutifs : développer comme sur un système d'exploitation, mais génèrere une application monolithique.
- Ordonnanceur $\Rightarrow$ plusieurs tâches semblent exécutées en parallèle $\Rightarrow$ notion de prorité.
- Permet àplusieurs programmeurs de développer en parallèle, chacun fournissant une tâche précise.
- l'environnement exécutif fournit les mécanismes de synchronisation des tâches et des données qui leur sont associées.
- Portabilité des applications : les couches matérielles sont (en principe) cachées au développeur.
- Par contre une contrainte additionnelle : maîtriser un nouvel ensemble de protocoles et méthodes de programmation.

Environnements exécutif

- Pas de chargement dynamique de bibliothèque ou de programme : code statique
- concept de tâches ...
- ... et des problèmes associés (concurrence d'accès aux ressources, aux variables, priorité),
- solutions : mutex, sémaphores.
- Approprié pour le développement parallèle de tâches et réutilisation de code
- quelques environnements exécutifs libres [1] : FreeRTOS [2], TinyOS [3] RTEMS [4], NuttX [5]
- l'ordonnanceur ne doit jamais être en famine de processus à traiter ⇒`xTaskCreate()` appelle des tâches avec boucle infinie.

1. choisir en fonction des ressources disponibles et des plateformes supportées
2. `www.freertos.org`
3. `www.tinyos.net`,
4. `www.rtems.com`
5. `nuttx.org` & `bitbucket.org/nuttx/nuttx`

Exemple de tâches sous FreeRTOS

```
#include "stm32f10x.h"    // hardware specific
#include <stm32/gpio.h>
#include "FreeRTOS.h"     // FreeRTOS specific
#include "task.h"
#include "queue.h"
#include "croutine.h"
int main(void){
{Led_Init();
 Usart1_Init();
 xTaskCreate( vLedsFloat, (signed char*) "Led1",1,NULL,10,NULL );
 xTaskCreate( vLedsFlash, (signed char*) "Led2",2,NULL,10,NULL );
 xTaskCreate( vPrintUart, (signed char*) "Uart",3,NULL,10,NULL );
 vTaskStartScheduler();
 return 0;         // should never be reached
}

void vLedsFloat(void* dummy)  // noter que chaque tache est infinie
{while(1)
  {GPIO_SetBits (GPIOC, GPIO_Pin_2);  vTaskDelay(120/portTICK_RATE_MS);
   GPIO_ResetBits (GPIOC, GPIO_Pin_2);vTaskDelay(120/portTICK_RATE_MS);
}}

void vLedsFlash(void* dummy)
{while(1)
 {GPIO_SetBits (GPIOC, GPIO_Pin_1);  vTaskDelay(301/portTICK_RATE_MS);
  GPIO_ResetBits (GPIOC, GPIO_Pin_1);vTaskDelay(301/portTICK_RATE_MS);
}}

void vPrintUart(void* dummy) // Writes each 500 ms
{portTickType last_wakeup_time;
 last_wakeup_time = xTaskGetTickCount();
 while(1){uart_puts("Hello World\r\n");
          vTaskDelayUntil(&last_wakeup_time, 500/portTICK_RATE_MS);
}}
```

Ordonnanceur (*scheduler*)

- coopératif : une tâche qui finit son travail donne la main à la suivante
- préemptif : l'ordonnanceur alloue un temps prédéfini maximum à chaque tâche
- priorité des tâches : une tâche de niveau inférieur peut être préemptée si les tâches de niveau supérieur ont fini leur travail

Trois états de tâches

- Running (en cours d'exécution)
- Runnable (demande à être exécutée dans la queue d'allocation du temps CPU)
- Idle (en attente)
- (en cours de déchargement)

⚠ **ATTENTION** : ne pas bloquer la séquence d'ordonnancement en retenant une tâche de haut niveau de priorité active.

Gestion de la pile

- Chaque tâche a sa propre pile
- La tâche *idle* (ordonnanceur en attente) a une pile de taille `configMINIMAL_STACK_SIZE`
- une pile trop petite se traduit par une corruption de mémoire
- FreeRTOS introduit une signature dans la pile pour détecter son dépassement

```
#define INCLUDE_uxTaskGetStackHighWaterMark 1
```

et

```
val=uxTaskGetStackHighWaterMark(NULL); // NULL = my own task handler
```

- FreeRTOS peut détecter un dépassement de pile à l'allocation

```
#define configCHECK_FOR_STACK_OVERFLOW 2
```

et

```
void vApplicationStackOverflowHook(TaskHandle_t xTask, \
     signed char *pTaskName)
{[... afficher ‘‘pTaskName’’ qui est la cause du depassement ...]}
```

Gestion de la pile

Liste des tâches :

```
#define configUSE_TRACE_FACILITY        1
#define configUSE_STATS_FORMATTING_FUNCTIONS    1
```

et

```
vTaskList(c);uart_puts(c);
```

dans la tâche qui se charge d'afficher l'état de l'ordonnanceur.

```
Hello World
Uart            R       4       301     3
IDLE            R       0       0       4
LedFlash        B       4       242     2
LedFloat        B       4       242     1
```

indique la taille restante de la pile, l'état de chaque tâche et son ordre d'ordonnancement.

Thread/fork

La gestion des processus :

- le changement de contexte est une opération lourde [6]
- `thread` : occupe les mêmes ressources mémoire que le père mais contient sa propre pile ⇒ partage de ressources mais élimine les problèmes de communication entre processus
  (⇒ risque de corruption des données)
- `fork` : dupliquer un processus en héritant de tous les attributs du père ⇒ possède sa propre mémoire virtuelle ⇒ pas de problème de propriété des données
- pour les systèmes sans MMU, `vfork`

6. `https://computing.llnl.gov/tutorials/pthreads/`

Les threads

Un père peut donc donner naissance à une multitude de fils qui effectuent la même tâche (par exemple serveur – répondre aux requêtes sur un réseau)

- ces fils partagent des ressources/variables
- Gestion de plusieurs clients dans les applications serveur.
- Méthode pour diviser un processus en sous-tâches indépendantes (client-serveur ou applications graphiques).
- Gestion des variables partagées entre threads : les `mutex`[7]

Implémentation des `threads` sous Linux : la librairie `pthreads`
⇒ compiler avec `-lpthread`

7. `http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html`

Thread/mutex

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

#define NTHREADS 10

void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

int main()
{int d[10];
 pthread_t thread_id[NTHREADS];
 int i, j;

 for (i=0; i < NTHREADS; i++)
     {d[i]=i;
      pthread_create( &thread_id[i], NULL, thread_function, &d[i] );} // CREATION DES 10 THREADS
 for(j=0; j < NTHREADS; j++) {pthread_join( thread_id[j], NULL);}     // ATTENTE DE LA MORT DE CES THREADS
 printf("Final counter value: %d\n", counter);
}

void *thread_function(void *d)
{printf("Thread number %d: %lx\n", *(int *)d, pthread_self());
 pthread_mutex_lock( &mutex1 );
 usleep(500000); // 100 ms
 counter++;
 pthread_mutex_unlock( &mutex1 );
}
```

Que se passe-t-il si on *sort* le `usleep()` du mutex ?

```
$ time ./thread                    $ time ./thread
...                                ...
real    0m5.001s                   real    0m0.503s
user    0m0.004s                   user    0m0.000s
sys     0m0.000s                   sys     0m0.000s
```

Mutex FreeRTOS (STM32)

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"
#include "croutine.h"

int global=0;
xSemaphoreHandle xMutex;

void task_rx(void* p)
{   char aff[10];
    char *t=(char*)p;
    int myInt = 0; volatile int local;
    for (myInt=0;myInt<8;myInt++)
    {xSemaphoreTake( xMutex, portMAX_DELAY );
     local=global;              // charge la variable
     local++;
     uart_puts(t);              // operation qui prend du temps
     global=local;              // stocke resultat
     xSemaphoreGive( xMutex );
     vTaskDelay( ( rand() & 0x1 ) );
    }
    aff[0]=' ';aff[1]=global+'0';aff[2]=' ';aff[3]=0;uart_puts(aff);
    while (1) vTaskDelay( ( rand() & 0x5 ) );
}

int main()
{ Led_Init(); Usart1_Init(); srand( 567 );
  xMutex = xSemaphoreCreateMutex();
  xTaskCreate(task_rx, (signed char*)"t1", (100), "→
      ↪11111111111111111111111111111111111111111\r\n\0", 1, 0);
  xTaskCreate(task_rx, (signed char*)"t2", (100), "→
      ↪22222222222222222222222222222222222222222\r\n\0", 1, 0);
  vTaskStartScheduler();
}
```

Passage de paramètres FreeRTOS

- Une variable sur la pile sera perdue au lancement de l'ordonnanceur ⇒ valeur passée erronée.
- Stockage de la variable en RAM par le préfixe static

```
void func(void* p)
{ int numero= *(int*) p;
  uart_putc('a'+numero);vTaskDelay(500 / portTICK_RATE_MS);
  while (1) { vTaskDelay(100 / portTICK_RATE_MS); };
}

int main()
{
  static int p[5]={0,1,2,3,4};
  static char * taskNames[5] = {"P0","P1","P2","P3","P4"};

  int i;
  Led_Init();
  Usart1_Init();
  for (i=0;i<NB_PHILO;i++)
      {xTaskCreate(func, taskNames[i], (256), (void*)&p[i],1,0);}
  vTaskStartScheduler();
  while(1);
  return 0;
}
```

Exemple : RTEMS sur STM32

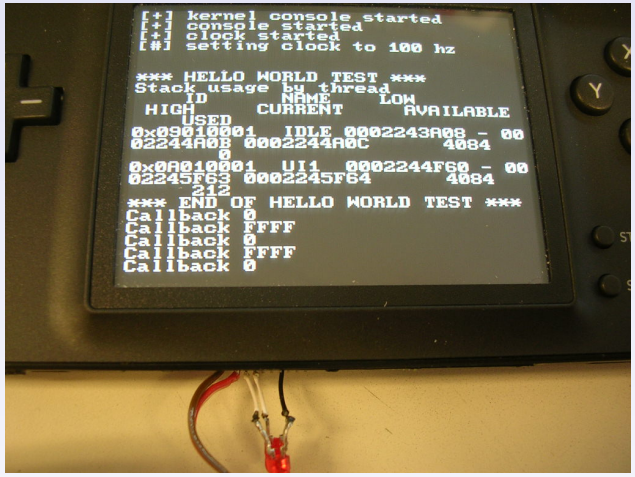
```c
#include <rtems/test.h>
#include <bsp.h> /* for device driver prototypes */
#include <stdio.h>
#include <stdlib.h>

const char rtems_test_name[] = "HELLO WORLD";

rtems_task Init(
  rtems_task_argument ignored
)
{
  rtems_test_begin();
  printf( "Hello World\n" );
  rtems_test_end();
  exit( 0 );
}

#define CONFIGURE_APPLICATION_DOES_NOT_NEED_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER

#define CONFIGURE_MAXIMUM_TASKS             1
#define CONFIGURE_USE_DEVFS_AS_BASE_FILESYSTEM

#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

#define CONFIGURE_INITIAL_EXTENSIONS RTEMS_TEST_INITIAL_EXTENSION

#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

70536 arm-rtems4.11/c/stm32f105rc/testsuites/samples/hello/hello.bin [8]

8. http ://wiki.rtems.com/wiki/index.php/STM32F105

Exemple : RTEMS sur NDS

- couche d'émulation POSIX de RTEMS : un environnement de travail proche de celui connu sous Unix
- abstraction des accès au matériel : framebuffer, réseau (wifi)
- pile TCP/IP pour la liaison haut débit
- exploitation efficace des ressources (4 MB RAM)

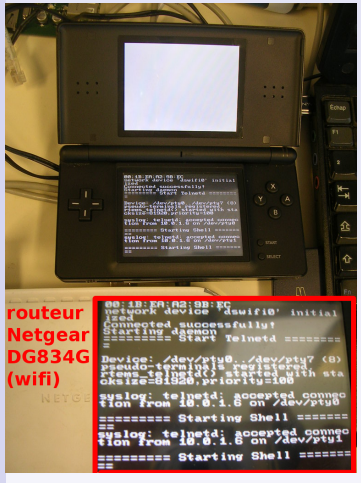


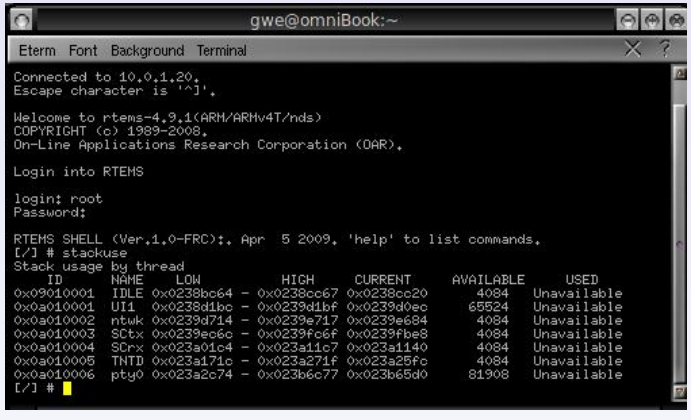
BSP RTEMS porté à NDS par M. Bucchianeri, B. Ratier, R. Voltz et C. Gestes

`http://www.rtems.com/ftp/pub/rtems/current_contrib/nds-bsp/manual.html`

Exemple : RTEMS sur NDS

- couche d'émulation POSIX de RTEMS : un environnement de travail proche de celui connu sous Unix
- abstraction des accès au matériel : framebuffer, réseau (wifi)
- pile TCP/IP pour la liaison haut débit
- exploitation efficace des ressources (4 MB RAM)





BSP RTEMS porté à NDS par M. Bucchianeri, B. Ratier, R. Voltz et C. Gestes

`http://www.rtems.com/ftp/pub/rtems/current_contrib/nds-bsp/manual.html`

Exemple RTEMS : framebuffer

Une application RTEMS doit déclarer les ressource qu'elle exploitera

```
#include <bsp.h>
#include <rtems/fb.h>
#include <rtems/console.h>
#include <rtems/clockdrv.h>
#include "fb.h"

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <rtems/mw_fb.h>

static struct fb_screeninfo fb_info;

inline void draw_pixel(int x, int y, int color)
{
  uint16_t* loc = fb_info.smem_start;
  loc += y * fb_info.xres + x;

  *loc = color;    // 5R, 5G, 5B
  *loc |= 1 << 15; // transparence ?
}

void draw_ppm()
{int x,y,bpp;char r,g,b;
 int l=0;
 for (y=0;y<161;y++)
   for (x=0;x<296;x++) {
     r=image[l++];g=image[l++];b=image[l++];
     bpp=(((int)(r&0xf8))>>3)+(((int)(g&0xf8))<<2)+ \
         (((int)(b&0xf8))<<8);
     if (x<256) draw_pixel(x, y, bpp);
   }
}
```

```
rtems_task Init(rtems_task_argument ignored)
{
  struct fb_exec_function exec;
  int fd = open("/dev/fb0", O_RDWR);

  if (fd < 0) {printk("failed\n");exit(0);}

  exec.func_no = FB_FUNC_ENTER_GRAPHICS;
  ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
  ioctl(fd, FB_SCREENINFO, (void*)&fb_info);

  draw_ppm();
  while (1) ;

  exec.func_no = FB_FUNC_EXIT_GRAPHICS;
  ioctl(fd, FB_EXEC_FUNCTION, (void*)&exec);
  close(fd);printk("done.\n");exit(0);
}

/* configuration information */
#define CONFIGURE_HAS_OWN_DEVICE_DRIVER_TABLE

rtems_driver_address_table Device_drivers[] =
{ CONSOLE_DRIVER_TABLE_ENTRY,
  CLOCK_DRIVER_TABLE_ENTRY,
  FB_DRIVER_TABLE_ENTRY,
  { NULL,NULL, NULL,NULL,NULL, NULL }
};

#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS      10
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE
#define CONFIGURE_MAXIMUM_TASKS 1
#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

Exemple RTEMS : GPIO

```
#include <bsp.h>
#include <stdlib.h>
#include <stdio.h>
#include <nds/memory.h>

rtems_id timer_id;
uint16_t l=0;

void callback()
{ printk("Callback %x\n",l);
  (*(volatile uint16_t*)0x08000000)=l;
  l=0xffff-l;
  rtems_timer_fire_after(timer_id, 100, callback, NULL);
}

rtems_task Init(rtems_task_argument ignored)
{ rtems_status_code status;
  rtems_name timer_name = rtems_build_name('C','P','U','T');

  printk( "\n\n*** HELLO WORLD TEST ***\n" );
// sysSetCartOwner(BUS_OWNER_ARM9);
  (*(vuint16*)0x04000204) = ((*(vuint16*)0x04000204) \
         & ~ARM7_OWNS_ROM);

  status = rtems_timer_create(timer_name,&timer_id);
  rtems_timer_fire_after(timer_id, 1, callback, NULL);
  rtems_stack_checker_report_usage();
           // requires #define CONFIGURE_INIT

  printk( "*** END OF HELLO WORLD TEST ***\n" );
  while(1)
    ;
  exit( 0 );
}

/* configuration information */
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_RTEMS_INIT_TASKS_TABLE

/* configuration information */
#define CONFIGURE_MAXIMUM_DEVICES                     40
#define CONFIGURE_MAXIMUM_TASKS                       100
#define CONFIGURE_MAXIMUM_TIMERS                      32
#define CONFIGURE_MAXIMUM_SEMAPHORES                  100
#define CONFIGURE_MAXIMUM_MESSAGE_QUEUES              20
#define CONFIGURE_MAXIMUM_PARTITIONS                  100
#define CONFIGURE_MAXIMUM_REGIONS                     100
/* This seetings overwrite the ones defined in confdefs.h */
#define CONFIGURE_MAXIMUM_POSIX_MUTEXES               32
#define CONFIGURE_MAXIMUM_POSIX_CONDITION_VARIABLES   32
#define CONFIGURE_MAXIMUM_POSIX_KEYS                  32
#define CONFIGURE_MAXIMUM_POSIX_QUEUED_SIGNALS        10
#define CONFIGURE_MAXIMUM_POSIX_THREADS               128
#define CONFIGURE_MAXIMUM_POSIX_TIMERS                10
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS      200

#define STACK_CHECKER_ON
#define CONFIGURE_INIT

#include <rtems/confdefs.h>
```

J.-M Friedt, G. Goavec-Merou, *Interfaces matérielles et OS libres pour Nintendo DS : DSLinux et RTEMS*, GNU/Linux Magazine France Hors Série 43, août 2009

Mise en pratique

1. FreeRTOS sur STM32 : un "OS" embarqué fournissant des fonctionnalités telles que tâches, ordonnanceur, mutex, sémaphores.
2. RTEMS : environnement exécutif portable, notamment sur STM32
3. exécution sur émulateur lorsque les ressources sont insuffisantes sur la plateforme matérielle disponible

 **ATTENTION** : nous sommes susceptibles de compiler pour deux cibles, STM32F100 (plateforme matérielle STM32VL-Discovery) et STM32F103 (émulateur `qemu`).

Ces deux plateformes ne sont pas compatibles : la sélection se fait par le choix des options de compilation dans le `Makefile`
`-DSTM32F10X_LD_VL` pour le F100
`-DSTM32F10X_MD` pour le F103