Inter-process communication (FIFO and Shared Memory)

RTOS Support

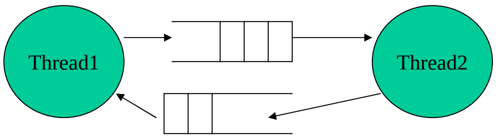
- NOT necessarily responsible everything
- RTOS system calls

Interrupt Management	Task Management	Task Communication
rtl_request_irq	pthread_create	FIFO
rtl_free_irq	pthread_setschedparam// pri. sched	Shared Memory
rtl_hard_enable_irq	pthread_make_periodic_np	Signal
rtl_hard_disable_irq	pthread_wait_np	
	pthread_delete_np	
	pthread_cancel	
	pthread_join	

Time Management	Mutual Exclusion	Device drivers
clock_gethrtime	Lock	rt_com
clock_gettime	Semaphore	rtsock
clock_settime		
gethrtime		
nanosleep		

Real-Time FIFO

- A kernel object that allows FIFO order communications among threads (Most RTOS provide something like this)



- In RT-Linux, it allows communications among rt-threads and linux(user space)-processes.

FIFO related functions

- FIFO object creation and destroy
  - In “init_module()” and “cleanup()”
    - #include <rtl_fifo.h>
    - int rtf_create(unsigned int fifo, int size); // fifo = 0, 1, 2, …, RTF_NO
    - int rtf_destroy(unsigned int fifo);
- Communication through FIFO
  - Read and write (<u>Non-blocked read/write in rt-threads</u>)
    - int rtf_get(unsigned int fifo, char *buff, int size); // return actual data size
    - int rtf_put(unsigned int fifo, char *buff, int size); // return size on success, otherwise return negative
  - Asynchronous communication with a Linux-process
    - int rtf_create_handler(unsigned int fifo, int (*handler)()); // invoked whenever a linux process read from or write to the fifo

Accessing rt_fifo from a Linux process

- In a Linux-process
  - Regard rt_fifo just like standard device (or special file)
    - each rt_fifo has its predetermined file name, /dev/rtf0, /dev/rtf1, etc.
    - fd = open("/dev/rtf0", O_WRONLY);
    - close(int fd);
    - ssize_t read(int fd, char *buf, size_t size); // return actual read data size
    - ssize_t write(int fd, char *buf, size_t size); // return actual written data size
  - By default, they are blocked io. For Non-blocked i/o, use "select"

Project

- Problem 1
  - Run "prj3_fifo"
  - Explain which time components are included in the measured data? Discuss about the accuracy of the measured data.

Project (Problem 1)

```c
#include <rtl.h>
#include <time.h>
#include <rtl_sched.h>
#include <rtl_fifo.h>

#define NO_OF_ITERATIONS 500
#define SENDER_DELAY_TIME 20000000 //nano seconds
#define RECEIVER_DELAY_TIME 100 //nano seconds

void *Receiver(void *param);
void *Sender(void *param);

pthread_t receiver, sender;
#define FIFO0 0
int init_module(void)
{
  rtl_printf("Ready to insert\n");
  rtf_destroy(FIFO0);
  if(rtf_create(FIFO0, 4000) < 0) {
      rtl_printf("fifo0 create error\n");
      return -1;
  }
  pthread_create(&receiver, NULL, Receiver, NULL);
  pthread_create(&sender, NULL, Sender, NULL);
  return 0;
}
```

```c
void cleanup_module(void)
{
        pthread_cancel(receiver);
        pthread_join(receiver,NULL);
        pthread_cancel(sender);
        pthread_join(sender, NULL);
        rtf_destroy(FIFO0);

        rtl_printf("Cleaned up\n");
}



/* The following structure is a data structure of the
information that needs to be transferred across the real-time
fifo queue.  It contains a message number (msg_no) and a
send time (send_time).  For message queues, this structure
can contain  any necessary information. */

typedef struct {
        unsigned int msg_no;
        hrtime_t send_time;
} data_t;
```

Project (Problem 1)

```c
void * Sender(void *param)
{
  struct sched_param p;
  int count;
  hrtime_t send_time;
  struct timespec sleep_time, rem;
  data_t message_buffer;
  p.sched_priority =
      sched_get_priority_min(SCHED_FIFO);
  pthread_setschedparam (pthread_self(),
      SCHED_FIFO, &p);
  sleep_time.tv_sec = 0;
  sleep_time.tv_nsec = SENDER_DELAY_TIME;
  rtl_printf("Sender: Ready to go into loop\n");
  for( count=0; count < NO_OF_ITERATIONS; count++ ) {
    send_time = clock_gethrtime(CLOCK_REALTIME);
    message_buffer.msg_no = count;
    message_buffer.send_time = send_time;
    if( rtf_put(FIFO0,&message_buffer,sizeof(data_t)) < 0 ) {
        rtl_printf( "Couldn't send message!\n");
        return 0;
    }
    nanosleep(&sleep_time, &rem);
  }
  return 0;
}
```

```c
void * Receiver(void *param)
{
  struct sched_param p;
  int count;
  hrtime_t receive_time, delta, max=0, total=0;
  struct timespec sleep_time, rem;
  data_t message_buffer;
  p.sched_priority =
      sched_get_priority_min(SCHED_FIFO); // 0
  pthread_setschedparam (pthread_self(),
      SCHED_FIFO, &p);
  rtl_printf("Receiver: Ready to go into loop\n");
  sleep_time.tv_sec = 0;
  sleep_time.tv_nsec = RECEIVER_DELAY_TIME;
  count = 0;
  while(count < NO_OF_ITERATIONS){
    if(rtf_get(FIFO0, &message_buffer, sizeof(data_t))
                        ==sizeof(data_t))
    {
      receive_time =
          clock_gethrtime(CLOCK_REALTIME);
      count++;
      delta  = receive_time  - message_buffer.send_time;
      total += delta;
      if( delta > max ) {max = delta;}
    }
    nanosleep(&sleep_time, &rem);
  }
  rtl_printf( “avg_delay and max”)’
  return 0;
}
```

Project

- Problem 2
  - Use “proj2” that checks stepping motor cycles. This time, make a linux program check the stepping motor cycle counts via real-time fifos upon user requests (for example, keyboard inputs). Thus, we have the following three rt-thread and one user-space program
    - Rt-threads
      - One highest priority task periodically performs factorial(1000000) with period 0.1sec.
      - Two lowest priority tasks run stepping motor with speeds of 1 turn/sec and 2 turns/sec, respectively.
    - User-space program (Regular linux process)
      - Whenever the user requests (input the motor number (1 or 2) to see), it requests the rotation status to the corresponding rt-thread through the request FIFO. If the rt-thread reads the request, it writes the rotation status to the response FIFO. Finally, the user thread read the response FIFO and print the result.

Shared Memory

- Some RTOSs provide address space protection among different threads (Rt linux does not!)
- In such cases, address space cannot be shared.
- So, explicit sharing mechanism is to be provided – ***<u>shared memory</u>***.
- Even in Rtlinx, shared memory is useful for communication between RT-Linux threads and Linux-processes

mbuff alloc and free in RT

- In “init_module()”
  - #include <mbuff.h>
  - ptr = (volatile char *) mbuff_alloc(“shm_name”, size);
  - Increase reference count
- In “cleanup()”
  - mbuff_free(“shm_name”, (void *) ptr);
  - Decrease reference count and when the count becomes zero, the memory is actually freed.
- If “ptr” is global, the allocated mbuff can be shared just like regular memory space.
- In rtlinux, if we only need to share a memory space among RT-threads, we can simply use global memory area.

Accessing the mbuff allocated shared memory from a Linux-process

- #include <mbuff.h>
- ptr = (volatile char *) mbuff_alloc(“shm_name”, size);
  - Increase reference count
- mbuff_free(“shm_name”, (void *) ptr);
  - Decrease reference count

- The allocated mbuff can be shared just like regular memory space.
- This is a way for Rt-linux to communicate with Linux

Project

- Problem 3
  - Run "prj3_shmem"
  - Explain what is going on

Project (Problem 3)

```
#include <rtl.h>
#include <time.h>
#include <rtl_sched.h>
#include <mbuff.h>
void *Checker(void *param);
void *RT_Writer(void *param);

pthread_t checker, writer;
volatile char *my_shm;

int init_module(void)
{
  rtl_printf("Ready to insert\n");
  my_shm = (volatile char *) mbuff_alloc("my_shm", 1024);
  if(my_shm == NULL){
    rtl_printf("mbuff_alloc failed\n");
    return -1;
  }
  sprintf((char *)my_shm, "hello world");
  pthread_create(&checker, NULL, Checker, NULL);
  pthread_create(&writer, NULL, RT_Writer, NULL);
  return 0;
}
void cleanup_module(void)
{
  pthread_cancel(checker);
  pthread_join(checker,NULL);
  pthread_cancel(writer);
  pthread_join(writer, NULL);
  mbuff_free("my_shm", (void*)my_shm);
  rtl_printf("Cleaned up\n");
}
```

```
void * Checker(void *param)
{
  struct sched_param p;
  p.sched_priority =
     sched_get_priority_max(SCHED_FIFO); // 1000000
  pthread_setschedparam (pthread_self(),
             SCHED_FIFO, &p);
  pthread_make_periodic_np (pthread_self(),
            gethrtime(), 1000000000);

  while(1){
    pthread_wait_np();
    rtl_printf("Checker found `%s` in my_shm\n",
                  (char *)my_shm);
  }
  return 0;
}

void * RT_Writer(void *param)
{
  struct sched_param p;    int count;
  p.sched_priority =
      sched_get_priority_max(SCHED_FIFO); // 1000000
  pthread_setschedparam (pthread_self(),
          SCHED_FIFO, &p);
  pthread_make_periodic_np (pthread_self(),
          gethrtime(), 2000000000);
  for( count = 0; 1; count++ ) {
    pthread_wait_np();
    sprintf((char *)my_shm,
        "RT_Writer running %dth cycle", count);
  }
  return 0;
}
```

Project (Problem 3)

```
#include <stdio.h>
#include <mbuff.h>

volatile char *user_shm;

int main(int argc, char **argv)
{
  user_shm = (volatile char*)
  mbuff_alloc("my_shm",1024);
  if( user_shm == NULL) {
    printf("mbuff_alloc failed\n");
    return -1;
  }
  while(1){
    scanf("%s", user_shm);
    if(strcmp((const char *)user_shm, "exit")==0)
            break;
    printf("User wrote `%s` to my_shm\n",
         user_shm);
  }
  mbuff_free("my_shm",(void*)user_shm);
  return 0;
}
```

Project

- Problem 4
  - Use “proj2” that checks stepping motor cycles. This time, make a linux program check the stepping motor cycle counts via shared memory upon user requests (for example, keyboard inputs). Thus, we have the following three rt-thread and one user-space program
    - Rt-threads
      - One highest priority task periodically performs factorial(1000000) with period 0.1sec.
      - Two lowest priority tasks run stepping motor with speeds of 1 turn/sec and 2 turns/sec, respectively.
    - User-space program (Regular linux process)
      - Whenever the user requests (input the motor number (1 or 2) to see), it asks the corresponding rt-thread to report the rotation status on the shared memory. When the data becomes available, the linux process read and print it. (Hint: Define tag variables on the shared memory for request and response. Will your synchronization mechanism with tags work without explicit Mutex? Explain your answer.)

Mutual Exclusion and Priority Inversion

RTOS Support

- NOT necessarily responsible everything
- RTOS system calls

Interrupt Management
rtl_request_irq
rtl_free_irq
rtl_hard_enable_irq
rtl_hard_disable_irq

Time Management
clock_gethrtime
clock_gettime
clock_settime
gethrtime
nanosleep

Task Management
pthread_create
pthread_setschedparam// pri. sched
pthread_make_periodic_np
pthread_wait_np
pthread_delete_np
pthread_cancel
pthread_join

Task Communication
FIFO
Shared Memory
Signal

Mutual Exclusion
Lock
Semaphore

Device drivers
rt_com
rtsock

Mutual Exclusion

- Some resources (e.g., a shared variable) should be shared in mutual-exclusive way
- Most RTOSs provide lock/unlock mechanism for this

mutex functions

- Create and destroy mutex kernel object
  - In “init_module()” and “cleanup()”
    - #include <pthread.h>
    - int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex_attr_t *mutexattr);
    - int pthread_mutex_destroy(pthread_mutex_t *mutex);
- Enter and leave the critical section
  - In rt threads
    - int pthread_mutex_lock(pthread_mutex_t *mutex); // blocked when mutex already locked
    - int pthread_mutex_trylock(pthread_mutex_t *mutex); // return EBUSY when mutex already locked
    - int pthread_mutex_unlock(pthread_mutex_t *mutex);

Project

- Problem 5
  - Run "prj3_mutex"
  - Explain the program and the results. What are differences mutual-exclusive sharing and non-mutual-exclusive sharing in this example?

Project (Problem 5)

```
//#define PROTECT
// include files …..
char lineString[] = "abcdefghijklmnopqrstuvwxyz
1234567890 ABCDEFGHIJKLMNOPQRSTUVWXYZ";

void *printLine(void *arg);
#define MAX_THREADS 3
pthread_t thread_id[MAX_THREADS];
volatile char *shm_lineIndex;
#ifdef PROTECT
static pthread_mutex_t mutex
#endif
int init_module(void)
{
  int arg[MAX_THREADS], i;
  rtl_printf("Ready to insert\n");
#ifdef PROTECT
  pthread_mutex_init (&mutex, NULL);
#endif
  shm_lineIndex = (volatile char *)
              mbuff_alloc("lineIndex", sizeof(int));
  for(i=0; i<MAX_THREADS; i++){
    arg[0] = i;
    pthread_create (&thread_id[i], NULL, printLine,
          (void *)arg);
  }
  return 0;
}
```

```
void cleanup_module(void)
{
  int i;
  for(i=0; i< MAX_THREADS; i++){
    pthread_cancel(thread_id[i]);
    pthread_join(thread_id[i], NULL);
  }
  mbuff_free("lineIndex", (void *)shm_lineIndex);
#ifdef PROTECT
  pthread_mutex_destroy (&mutex);
#endif
  rtl_printf("Cleaned up\n");
}
```

Project (Problem 5)

```
void *printLine(void *arg)
{
  struct sched_param p;
  struct timespec tv, rem;
  int number;
  int tmp_index;
  int i;


  rtl_printf("Thread %d started ...\n",
        ((int *)arg)[0]);
  p.sched_priority =
      sched_get_priority_min(SCHED_FIFO);
  pthread_setschedparam(pthread_self(),
      SCHED_FIFO, &p);
  pthread_make_periodic_np(pthread_self(),
      gethrtime(), (hrtime_t)200000000);

  while(1) {
    pthread_wait_np();
    clock_gettime(CLOCK_REALTIME, &tv);
    number = ((tv.tv_nsec / 37) % 4) + 1;
#ifdef PROTECT
    if(pthread_mutex_lock (&mutex) != 0){
      rtl_printf("Mutex lock fail\n");
      return NULL;
    }
#endif
    tmp_index = *shm_lineIndex;
    for( i = 0; i < number; i++ ) {
      if( tmp_index + i < sizeof(lineString)-1 ) {
        rtl_printf("%c", lineString[tmp_index + i]);
        tv.tv_sec = 0;
        tv.tv_nsec = 1000;
        nanosleep(&tv, &rem);
      }
    }
    *shm_lineIndex = tmp_index + i;
    if( (tmp_index + i) >= sizeof(lineString)-1 ) {
      rtl_printf("\n");
      *shm_lineIndex = 0;
    }
#ifdef PROTECT
    if(pthread_mutex_unlock (&mutex) !=0){
      rtl_printf("Mutex unlock fail\n");
      return NULL;
    }
#endif
  } /* end while */
}
```

Priority Inversion

- Mutual-exclusion can cause unbounded blocking to high priority tasks
- Solutions
  - Disable interrupts in critical sections (Problem?)
  - Priority inheritance protocol
  - Priority ceiling protocol

Project

- Problem 6
  - Run “prj3_inversion”.
  - Explain the program and results. What are the average and worst case response time of the highest priority task? Explain the scenario of chained blocking in this example

Project (Problem 6)

```
// include header files ….

long long fibonacci(int n)
{
    long long retValue=1;
    int i;
    for(i=1;i<=n;i++)
      retValue *=(long long)i;
    return retValue;
}

volatile char *lockOwner;
#define                    UNLOCKED  1
#define                    HIGH_HAS_LOCK          2
#define                    LOW_HAS_LOCK           3
static pthread_mutex_t mutex;
void *High( void *arg);
void *Middle( void *arg);
void *Low( void *arg);
pthread_t threadHigh, threadMiddle, threadLow;
int init_module(void)
{
   pthread_mutex_init(&mutex, NULL);
   lockOwner = (volatile char *) mbuff_alloc("lockOwner",
              sizeof(int));
   *lockOwner = UNLOCKED;
   pthread_create(&threadHigh, NULL, High, NULL);
   pthread_create(&threadMiddle, NULL, Middle, NULL);
   pthread_create(&threadLow, NULL, Low, NULL);
   return 0;
}
```

```
void cleanup_module(void)
{
   pthread_cancel(threadHigh);
   pthread_join(threadHigh, NULL);
   pthread_cancel(threadMiddle);
   pthread_join(threadMiddle, NULL);
   pthread_cancel(threadLow);
   pthread_join(threadLow, NULL);

   mbuff_free("lockOwner", (void *)lockOwner);
   pthread_mutex_destroy (&mutex);
   rtl_printf("Cleaned up\n");
}
```

Project (Problem 6)

```
void *High( void *arg)
{
   struct sched_param p;
   hrtime_t start, finish;
   p.sched_priority = HIGH_PRIORITY;
   pthread_setschedparam(pthread_self(),
               SCHED_FIFO, &p);
   pthread_make_periodic_np(pthread_self(),
                  gethrtime()+200000, 1000000000);

   while(1) {
      pthread_wait_np();
      start = clock_gethrtime(CLOCK_REALTIME);
      rtl_printf("High started at %lld\n", start);
      rtl_printf("High trying to get the lock!\n");
      if(pthread_mutex_lock (&mutex) !=0){
            rtl_printf("Mutex lock fail\n");
            return NULL;
      }
      *lockOwner = HIGH_HAS_LOCK;
      rtl_printf("High has the lock!\n");
      fibonacci(100);
      rtl_printf("High release the lock!\n");
      *lockOwner = UNLOCKED;
      if(pthread_mutex_unlock (&mutex) !=0){
            rtl_printf("Mutex unlock fail\n");
            return NULL;
      }
      finish = clock_gethrtime(CLOCK_REALTIME);
      rtl_printf("High finished at %lld (Rsp = %lld)\n",
            finish, finish-start);
   }
}
```

```
void *Middle( void *arg)
{
   struct sched_param p;   int i;
   p.sched_priority = MIDDLE_PRIORITY;
   pthread_setschedparam(pthread_self(),
               SCHED_FIFO, &p);
   while(1) {
      usleep(200);
      if( *lockOwner == LOW_HAS_LOCK ) {
            rtl_printf("Low has lock,
                  Middle entering long loop!\n");
            for( i = 0; i < 100000000; i++ );
            rtl_printf("Low has lock,
                  Middle exiting long loop!\n");
      }
   } // end while
}
```

Project (Problem 6)

```
void *Low( void *arg)
{
   struct sched_param p;
   hrtime_t start, finish;
   p.sched_priority = LOW_PRIORITY;
   pthread_setschedparam(pthread_self(),
            SCHED_FIFO, &p);
   pthread_make_periodic_np(pthread_self(),
                  gethrtime()+999999000, 2000000000);
   while(1) {
      pthread_wait_np();
      start = clock_gethrtime(CLOCK_REALTIME);
      rtl_printf("Low started at %lld\n", start);
      if(pthread_mutex_lock (&mutex) !=0){
         rtl_printf("Mutex lock fail\n");
         return NULL;
      }
      *lockOwner = LOW_HAS_LOCK;
      rtl_printf("Low has the lock!\n");
      fibonacci(5000);
      *lockOwner = UNLOCKED;
      rtl_printf("Low release the lock!\n");
      if(pthread_mutex_unlock (&mutex) !=0){
          rtl_printf("Mutex unlock fail\n");
          return NULL;
      }
      fibonacci(10000);
      finish = clock_gethrtime(CLOCK_REALTIME);
      rtl_printf("Low finished at %lld (Rsp = %lld)\n",
            finish, finish-start);
   } // end while
}
```

Project

- Problem 7
  - Most of posix compliance RTOSs support priority inheritance protocol. However, it is not the case in Rt-linux. Mimic the priority inheritance protocol using pthread_mutex_trylock(), pthread_getschedparam(), and pthread_setschedparam(). Compare the results with the above. What are limitations of your implementation of priority inheritance protocol?

Homework

- Make your schedulability analysis (consider only fixed priority scheduling) program as follows:
    - read task parameters: no of tasks, task execution times, task periods, and task deadlines
    - read resource access information: accessing resources, the length of outermost critical section, etc
    - read priority assignment
    - perform response time analysis
      - output the worst case response time of tasks if they are schedulable and output –1 for all unschedulable tasks
    - perform utilization bound check (RM case only)
      - task by task check – identify which task is unschedulable
      - system-level check – say yes (schedulable) or no (not schedulable)
  - The report should include a short manual to explain how to use your program.