Systemy operacyjne czasu rzeczywistego

Synchronizacja zadań. Komunikacja międzyprocesowa.

Paweł Strączyński
pstraczynski@tu.kielce.pl
Katedra Urządzeń Elektrycznych i Automatyki

Synchronizacja zadań

Synchronizacja zadań jest jedną z podstawowych funkcjonalności w środowisku, w którym wiele wątków działa równolegle i niezależnie od siebie.
W takim przypadku ważnym jest aby istniały mechanizmy które pozwolą na zachowanie determinizmu oraz zachowanie kolejności dostępu do tzw. sekcji krytycznej.
Systemy RT posiadają takie narzędzia do synchronizacji zadań jak:

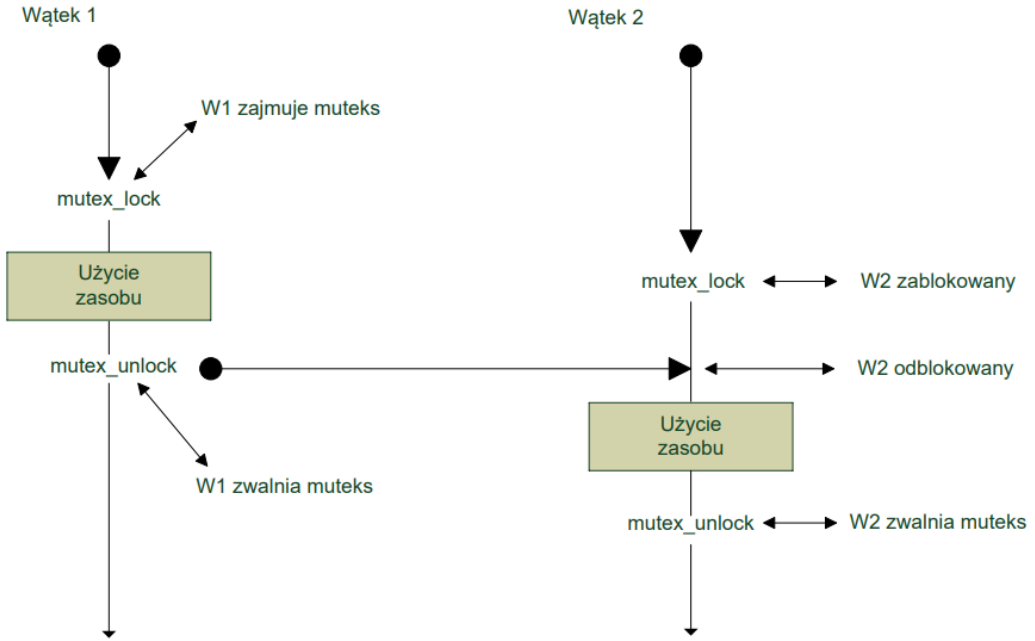
- Mutex'y,
- Semafory.

Mutex’y

Mutex czyli semafor zapewniający wzajemne wykluczanie, mutex posiada dwa stany wolny lub zajęty.

Wątek W1 zajmuje mutex, a zarazem otrzymuje dostęp do zasobu, w tym czasie wątek W2 stara się o ten sam zasób. Z racji tego że mutex został już zajęty przez wątek W1, wątek W2 zostaje zablokowany do czasu zwolnienia mutex’u. Wątek W1 przeprowadza operacje na zasobach po czym zwalnia mutex, w tym czasie wątek W2 zajmuję mutex i rozpoczyna przetwarzanie zasobów, po zakończeniu przetwarzania mutex zostaje zwolniony.

Tylko wątek który zajął mutex może go zwolnić!



Mutex'y

int	**rt_mutex_create** (RT_MUTEX *mutex, const char *name) Create a mutex. More...
int	**rt_mutex_delete** (RT_MUTEX *mutex) Delete a mutex. More...
int	**rt_mutex_acquire** (RT_MUTEX *mutex, RTIME timeout) Acquire a mutex. More...
int	**rt_mutex_acquire_until** (RT_MUTEX *mutex, RTIME timeout) Acquire a mutex (with absolute timeout date). More...
int	**rt_mutex_release** (RT_MUTEX *mutex) Unlock mutex. More...
int	**rt_mutex_inquire** (RT_MUTEX *mutex, **RT_MUTEX_INFO** *info) Inquire about a mutex. More...

Mutex'y

```
#include <native/mutex.h>
RT_MUTEX mutex_desc;

int main(int argc, char *argv[])
{
        int err;

        /*Utwórz mutex. Można również utowrzyć powiązanie z wcześniej istniejącym obiektem z
        wykorzystaniem funkcji rt_mutex_bind(). */
        err = rt_mutex_create(&mutex_desc,"MyMutex");

        /* zablokowanie dostępu do sekcji krytycznej: */
        rt_mutex_acquire(&mutex_desc,TM_INFINITE);

        /* ... Sekcja krytyczna i odblokowanie mutex'u... */
        rt_mutex_release(&mutex_desc);
        /* ... */
}

void cleanup(void)
{
        rt_mutex_delete(&mutex_desc);
}
```

Mutex'y

```
#include <native/mutex.h>
RT_MUTEX mutex_desc;
long count;

int increment_count()
{
        rt_mutex_acquire(&mutex_desc,TM_INFINITE);
        count = count+1;
        rt_mutex_release(&mutex_desc);
}

void cleanup(void)
{
        long c;
        rt_mutex_acquire(&mutex_desc,TM_INFINITE);
        c = count;
        rt_mutex_release(&mutex_desc);
}
```

Semafory

Semafor jest mechanizmem mechanizmem synchronizacyjnym podobnym do mutex'a. Podobnie jak mutex pełni on funkcje ochrony dostępu do wspólnego zasobu.
Różnica pomiędzy semaforem a mutex'em polega na tym, że semafor jest wyposażony w licznik, który służy do zliczania ilości wolnego zasobu.
Do operacji na tym liczniku służą funkcje:

- **rt_sem_v** zwiększa dany licznik,
- **rt_sem_p** zmniejsza go, aż do osiągnięcia wartości 0, wtedy wątek zasypia i oczekuje na podniesienie wartości.

Semafory mogą być wykorzystywane tam, gdzie zasób dzielony jest na ograniczoną ilość użytkowników. Semafor działa jak furtka kontrolująca ilość wątków wykonujących jakiś fragment kodu. Za pomocą semaforów aplikacja może kontrolować na przykład maksymalną ilość otwartych plików, czy utworzonych okien. Semafory są w działaniu bardzo podobne do mutex'ów.

Semafory

int	**rt_sem_create** (RT_SEM *sem, const char *name, unsigned long icount, int mode) Create a counting semaphore. More...
int	**rt_sem_delete** (RT_SEM *sem) Delete a semaphore. More...
int	**rt_sem_p** (RT_SEM *sem, RTIME timeout) Pend on a semaphore. More...
int	**rt_sem_p_until** (RT_SEM *sem, RTIME timeout) Pend on a semaphore (with absolute timeout date). More...
int	**rt_sem_v** (RT_SEM *sem) Signal a semaphore. More...
int	**rt_sem_broadcast** (RT_SEM *sem) Broadcast a semaphore. More...
int	**rt_sem_inquire** (RT_SEM *sem, RT_SEM_INFO *info) Inquire about a semaphore. More...

Semafory

```
#include <native/sem.h>
#define SEM_INIT 1
#define SEM_MODE S_FIFO

RT_SEM sem_desc;
void foo(void)
{
        int err;
        err=rt_sem_create(&sem_desc,"MySemaphore",SEM_INIT,SEM_MODE);
        for (;;) {
                rt_sem_p(&sem_desc,TM_INFINITE);

                rt_sem_v(&sem_desc);
        }
}

void cleanup(void)
{
        rt_sem_delete(&sem_desc);
}
```

Komunikacja pomiędzy procesami

Wielozadaniowy systemu czasu rzeczywistego poza mechanizmami synchronizacji powinien również zapewniać mechanizmy komunikacji pomiędzy tymi zadaniami.

Potrzeba wymiany informacji zachodzi gdy np. jedno zadanie odpowiedzialne jest za zebranie danych pomiarowych z przetwornika ADC, drugie za wykonanie obliczeń, trzecie za wyświetlenie wyników. Mechanizmy wymiany danych są więc silnie związane z synchronizacją – jeden wątek zapisuje a drugi odczytuje dane w określonym czasie.

Komunikacja przez pamięć wspólną

Jednym z podstawowych mechanizmów komunikacji pomiędzy zadaniami jest wykorzystanie pamięci wspólnej.

Pamięć wspólna jest to obszar pamięci, do której dostęp może mieć wiele wątków pisząc i czytając bezpośrednio w niej.

W przypadku komunikacji takiej niezbędne są mechanizmy ochrony sekcji krytycznej.

Komunikacja przez pamięć wspólną jest najszybszym rodzajem komunikacji międzyprocesowej ponieważ dane nie są nigdzie przesyłane.

Komunikacja przez pamięć wspólną wymaga osobnych mechanizmów synchronizacji.

Komunikacja przez pamięć wspólną

```
#include <native/heap.h>
RT_HEAP heap_desc;
void *shared_mem;

int main(int argc, char *argv[])
{
          int err;
          err = rt_heap_bind(&heap_desc,"SomeShmName",TM_NONBLOCK);
          if (err) fail();
          rt_heap_alloc(&heap_desc,0,TM_NONBLOCK,&shared_mem);
}
void cleanup(void)
{
          rt_heap_unbind(&heap_desc);
}
```

Komunikacja przez pamięć wspólną

int	**rt_heap_create** (RT_HEAP *heap, const char *name, size_t heapsize, int mode) Create a memory heap or a shared memory segment. More...
int	**rt_heap_delete** (RT_HEAP *heap) Delete a real-time heap. More...
int	**rt_heap_alloc** (RT_HEAP *heap, size_t size, RTIME timeout, void **blockp) Allocate a block or return the single segment base. More...
int	**rt_heap_free** (RT_HEAP *heap, void *block) Free a block. More...
int	**rt_heap_inquire** (RT_HEAP *heap, **RT_HEAP_INFO** *info) Inquire about a heap. More...
int	**rt_heap_bind** (RT_HEAP *heap, const char *name, RTIME timeout) Bind to a mappable heap. More...
int	**rt_heap_unbind** (RT_HEAP *heap) Unbind from a mappable heap. More...

Komunikacja przez kolejkę komunikatów

```
#include <sys/mman.h>
#include <stdio.h>
#include <string.h>
#include <native/task.h>
#include <native/queue.h>
#define TASK_PRIO 99 /* Highest RT priority */
#define TASK_MODE 0 /* No flags */
#define TASK_STKSZ 0 /* Stack size (use default one) */
RT_QUEUE q_desc;
RT_TASK task_desc;
void consumer(void *cookie)
{
        ssize_t len;
        void *msg;
        int err;
        err = rt_queue_bind(&q_desc,"SomeQueueName",TM_INFINITE);
        if (err) fail();
        while ((len = rt_queue_receive(&q_desc,&msg,TM_INFINITE)) > 0)
        {
                printf("received message> len=%d bytes, ptr=%p, s=%s\n",len,msg,(const char *)msg);
                rt_queue_free(&q_desc,msg);
        }
        rt_queue_unbind(&q_desc);
        if (len != -EIDRM)  fail();
}
```

Komunikacja przez kolejkę komunikatów

```
int main(int argc, char *argv[])
{
        static char *messages[] = { "hello", "world", NULL };
        int n, len;
        void *msg;
        mlockall(MCL_CURRENT|MCL_FUTURE);
        err = rt_task_create(&task_desc,"MyTaskName", TASK_STKSZ, TASK_PRIO, TASK_MODE);
        if (!err)
        rt_task_start(&task_desc,&task_body,NULL);
        for (n = 0; messages[n] != NULL; n++)
        {
                len = strlen(messages[n]) + 1;
                msg = rt_queue_alloc(&q_desc,len);
                if (!msg) fail();
                strcpy(msg,messages[n]);
                rt_queue_send(&q_desc,msg,len,Q_NORMAL);
        }
        rt_task_delete(&task_desc);
}
```

Komunikacja przez kolejkę komunikatów

int	**rt_queue_create** (RT_QUEUE *q, const char *name, size_t poolsize, size_t qlimit, int mode) Create a message queue. More...
int	**rt_queue_delete** (RT_QUEUE *q) Delete a message queue. More...
void *	**rt_queue_alloc** (RT_QUEUE *q, size_t size) Allocate a message queue buffer. More...
int	**rt_queue_free** (RT_QUEUE *q, void *buf) Free a message queue buffer. More...
int	**rt_queue_send** (RT_QUEUE *q, void *mbuf, size_t size, int mode) Send a message to a queue. More...
int	**rt_queue_write** (RT_QUEUE *q, const void *buf, size_t size, int mode) Write a message to a queue. More...
ssize_t	**rt_queue_receive** (RT_QUEUE *q, void **bufp, RTIME timeout) Receive a message from a queue. More...
ssize_t	**rt_queue_receive_until** (RT_QUEUE *q, void **bufp, RTIME timeout) Receive a message from a queue (with absolute timeout date). More...
ssize_t	**rt_queue_read** (RT_QUEUE *q, void *buf, size_t size, RTIME timeout) Read a message from a queue. More...
ssize_t	**rt_queue_read_until** (RT_QUEUE *q, void *buf, size_t size, RTIME timeout) Read a message from a queue (with absolute timeout date). More...
int	**rt_queue_flush** (RT_QUEUE *q) Flush a message queue. More...
int	**rt_queue_inquire** (RT_QUEUE *q, RT_QUEUE_INFO *info) Inquire about a message queue. More...