3 Synchronizacja zadań. Muteksy i semafory.

Wielozadaniowość systemu oprócz oczywistych korzyści stwarza również pewne problemy. W środowisku, w którym wiele wątków działa równolegle i niezależnie od siebie, potrzebne są mechanizmy, które pozwolą w pewien sposób zdeterminować ich kolejność, oraz dostęp do tzw. sekcji krytycznych. Sekcje te są to fragmenty kodu, które nie mogą zostać wykonane „w tym samym czasie" przez kilka wątków.

Systemy RT z natury wielozadaniowe posiadają szereg mechanizmów do wzajemnej synchronizacji. Podstawowym jest **muteks**. W Xenomai do utworzenia muteksa służy funkcja:

```
int rt_mutex_create
(RT_MUTEX *mutex, const char *name)
```

Tak jak w przypadku zadań (tasks) nazwy muteksów nie mogą się powtarzać. Obsługa muteksa sprowadza się do procedury zajęcia (`rt_mutex_acquire`), oraz zwolnienia (`rt_mutex_release`). Muteks musi mieć właściciela, zostaje nim wątek, w którego funkcji wykonawczej wykonywane są operacje na muteksie. Szczegółowe informacje, oraz spis wszystkich funkcji obsługi muteksa znajduje się w dokumentacji w `Mutex Services`.

Nieco bardziej złożonym od muteksa mechanizmem synchronizacyjnym jest **semafor**. Pełni taką samą funkcję, jego zadaniem jest ochrona dostępu do wspólnego zasobu. Różnica polega na tym, że semafor jest wyposażony w licznik, który służy do zliczania ilości wolnego zasobu. Funkcja `rt_sem_v` zwiększa dany licznik, natomiast `rt_sem_p` zmniejsza go, aż do osiągnięcia wartości 0, wtedy wątek zasypia i oczekuje na podniesienie wartości. Informacji na temat opisanych funkcji i ich parametrów należy szukać w dokumentacji w dziale `Counting semaphore services`

(a) Sytuacja przedstawia teleturniej, w którym udział bierze 3 zawodników. Każdy z nich znajduje się w osobnym pomieszczeniu i ma przed sobą ekran, na którym komputer wyświetla pytanie. Po dobrej odpowiedzi uczestnik jest punktowany, a komputer wyświetla kolejne pytanie ze wspólnej puli. Każdy z zawodników ilustruje wątek, który chce jak najszybciej odpowiedzieć i otrzymać kolejne pytanie. Wiąże się to ze zmniejszeniem licznika pytań przez komputer. Ta pula pytań ilustruje wspólny zasób.

Zapoznaj się z zawartością pliku `src/ex3/a/ex3a_main.c`. Przedstawia on opisany powyżej pro-

blem. Skompiluj go poleceniem `make` i uruchom. Wynik powinien być zbliżony do tego:

```
A got point! (4 questions left)
A got point! (1 questions left)
A got point! (0 questions left)
Game over. A got 3 points.
B got point! (3 questions left)
Game over. B got 1 points.
C got point! (2 questions left)
Game over. C got 1 points.
```

Jak widać program nie działa poprawnie. Dodaj do programu odpowiednią bibliotekę (`native/mutex.h`) oraz sam muteks. Na końcu programu nie zapomnij go usunąć za pomocą odpowiedniej funkcji. Zastanów się, który fragment kodu wymaga ochrony i stosując funkcje muteksa ochroń ten fragment. Skompiluj i uruchom. Jeśli sekcja krytyczna jest poprawnie chroniona rezultat powinien być następujący: (oczywiście kolejność może być inna, ponieważ nie wiemy który wątek będzie akurat szybszy)

```
A got point! (4 questions left)
B got point! (3 questions left)
C got point! (2 questions left)
A got point! (1 questions left)
B got point! (0 questions left)
Game over. C got 1 points.
Game over. A got 2 points.
Game over. B got 2 points.
```

Poniżej znajdują się dwa różne sposoby wyjścia z pętli. Zastanów się i odpowiedz czy oba są poprawne. Odpowiedź uzasadnij.

```
while(!end) {
  rt_mutex_acquire(&mutex,
                   TM_INFINITE);
  if (...)
    end = true;
  rt_mutex_release(&mutex);
}
```

```
while(1) {
  rt_mutex_acquire(&mutex,
                   TM_INFINITE);
  if (...)
    break;
  rt_mutex_release(&mutex);
}
```

(b) Innym przykładem synchronizacji dostępu do wspólnego zasobu jest problem Czytelników-Pisarzy. Przedstawia on sytuację, w której do wspólnej czytelni może wejść tylko jeden pisarz żeby napisać N stron książki. Gdy to zrobi opuszcza czytelnię, gdzie z kolei udają się czytelnicy. Po przeczytaniu N stron książki wychodzą. Do czytelni znowu wchodzi pisarz napisać N nowych stron. Nasz problem zawęzimy tylko do jednego pisarza i jednego czytelnika.

Istnieje wspólny zasób (książka) o stałej liczbie stron. Pisarz gdy tylko może zapisuje kolejne strony. Czytelnik gdy tylko może (jest zapisana chociaż jedna strona) czyta kolejne strony i oznacza jako przeczytane. Wtedy pisarz znowu może pisać nadpisując przeczytane strony. Czyli pisarz jest blokowany kiedy książka jest pełna, a czytelnik jest blokowany kiedy książka jest pusta. Tak więc zadanie polega na przekazaniu czytelnikowi wszystkiego co ma do przekazania pisarz tylko za pomocą książki o mniejszej liczbie stron niż potrzeba.
Skompiluj program `src/ex3/b/ex3b_main.c`, zawiera on niepełną implementację przedstawionego problemu. Po uruchomieniu pojawi się błąd naruszenia ochrony pamięci (segfault) oraz zostaną wypisane niepoprawne dane. Uzupełnij program niezbędnymi obiektami semaforów (`native/sem.h`), odpowiednio je zainicjalizuj oraz poprawnie ochroń sekcję krytyczną.

Do rozwiązania zadania użyj następujących obiektów:

```
RT_SEM    empty,
          full;
```

Semafor `empty` zlicza liczbę przeczytanych stron, dlatego należy go zainicjalizować wartością `BUFF_LEN`. `full` zlicza strony czekające na przeczytanie. Zarówno po napisaniu jak i po przeczytaniu strony można zrobić krótką przerwę (`rt_task_sleep`), żeby dopisać bądź przeczytać kolejne. `idea` może być dowolnym ciągiem znaków. Liczba stron również może być dowolna, ale nie powinna być większa od długości przekazywanej wiadomości. Wynik programu przy trzystronicowej książce powinien wyglądać następująco:

```
write: G(0) [G]
                        read: G(0) [G        ]
write: o(1) [Go]
write: o(2) [Goo]
write: d(0) [doo]
                        read: o(1) [Go       ]
```

```
write: _(1) [d_o]
                        read: o(2) [Goo      ]
write: i(2) [d_i]
                        read: d(0) [Good     ]
write: d(0) [d_i]
                        read: _(1) [Good_    ]
write: e(1) [dei]
                        read: i(2) [Good_i   ]
write: a(2) [dea]
                        read: d(0) [Good_id  ]
                        read: e(1) [Good_ide ]
                        read: a(2) [Good_idea]
```