Rozszerzenia czasu rzeczywistego w systemach Linux

Witold Paluszyński
Instytut Informatyki, Automatyki i Robotyki
Politechnika Wrocławska
http://sequoia.iiar.pwr.wroc.pl/~witold/

2011–2013

Ten utwór jest dostępny na licencji Creative Commons Uznanie autorstwa-Na tych samych warunkach 3.0 Unported

Utwór udostępniany na licencji Creative Commons: uznanie autorstwa, na tych samych warunkach. Udziela się zezwolenia do kopiowania, rozpowszechniania i/lub modyfikacji treści utworu zgodnie z zasadami w/w licencji opublikowanej przez Creative Commons. Licencja wymaga podania oryginalnego autora utworu, a dystrybucja materiałów pochodnych może odbywać się tylko na tych samych warunkach (nie można zastrzec, w jakikolwiek sposób ograniczyć, ani rozszerzyć praw do nich).

System Linux

Linux jest popularnym systemem operacyjnym, zbudowanym na wzór systemu Unix. To znaczy, wzorując się na istniejącym systemie operacyjnym Unix, napisany został od podstaw system bardzo wiernie go imitujący. Linux jest systemem działającym na wielu platformach sprzętowych, w tym na komputerach typu PC, ale również na wielu typach mikrokontrolerów. Jest systemem szeroko wykorzystywanym, i silnie rozwijającym się w wielu dziedzinach.

Jedną z przyczyn tej popularności jest fakt, że Linux jest niekomercyjnym systemem typu FOSS (*Free and Open Source Software*). To znaczy, jego kod źródłowy jest dostępny do dowolnego wykorzystania. W szczególności, można tworzyć rozszerzenia systemu dla dowolnych potrzeb, i wykorzystywać go komercyjnie.

Drugim ważnym źródłem popularności Linuksa jest bogaty zestaw dobrej jakości oprogramowania, które w większości również jest dostępne na zasadzie FOSS.

Modyfikacje Linuksa

Powyższe cechy powodują, że wielu użytkowników, w tym firmy komercyjne, także całkiem duże, próbują dostosować i wykorzystać Linuksa do innych celów, niż jego zasadnicze przeznaczenie. Obejmuje to budowę kolejnych dystrybucji, niekiedy specjalizowanych do określonych zastosowań, przenoszenie go na coraz to nowe platformy sprzętowe, itd. Jak również dostosowanie do potrzeb systemów wbudowanych i systemów czasu rzeczywistego.

Jednak Linux sam w sobie, jako system operacyjny ogólnego przeznaczenia (GPOS - *General Purpose Operating System*), nie jest w stanie spełniać twardych wymagań czasu rzeczywistego. Wynika to z silnie wbudowanych weń zasad demokratycznego zarządzania zasobami i procesami, zarządzania pamięcią przez naprzemienne jej alokowanie i zwalnianie, oraz ochronę, a także z niewywłaszczalności pewnych krytycznych obszarów jądra nawet przez nadchodzące przerwania.

Mechanizmy standard POSIX 1003.1b wprowadzone już od wersji 2.0 jądra, dostarczają dodatkowych algorytmów szeregowania, oraz pewnych mechanizmów czasu rzeczywistego, jednak zmiany te pozwalają tworzyć na bazie Linuksa co najwyżej miękkie systemy czasu rzeczywistego o charakterze miękkim.

Rozszerzenia czasu rzeczywistego Linuksa

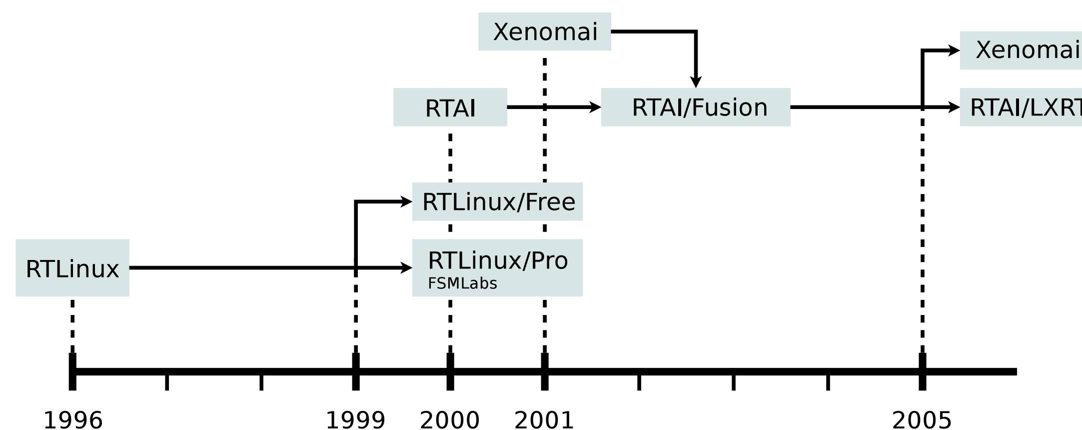
Podstawową zasadą systemów GPOS jest, że najważniejsze jest jądro systemu operacyjnego, które decyduje o wszystkim. Gdy jądro ma jakieś prace do wykonania, to wykonuje je przed wszystkimi zadaniami. W systemie czasu rzeczywistego, jądro nie jest najważniejsze. Jeśli jakieś zadanie musi mieć gwarantowany czas wykonania, to może ono być również wykonywane kosztem jądra. Mówimy, że jądro jest **wywłaszczane** (usuwane z procesora, aby go zwolnić dla innego zadania).

Poza ogólnym przygotowaniem Linuksa do tworzenia zadań czasu rzeczywistego, jak implementacja funkcji POSIX, timerów, oraz wprowadzenia priorytetu czasu rzeczywistego, stworzone zostały systemy stanowiące rozszerzenia Linuksa, pozwalające budować prawdziwe aplikacje czasu rzeczywistego. Rozszerzenia te nazywane są „Real-Time Linux" (RT Linux), i jest ich cały szereg, niektóre komercyjne, a inne o charakterze FOSS. Trzy bardziej znaczące:

- RTLinux, opracowany przez New Mexico Tech i obecnie utrzymywany przez Wind River Systems (komercyjny),
- RTAI (*Real-Time Application Interface*), opracowany przez Politechnikę Mediolańską,
- Xenomai, wywodzący się z RTAI Linux, wprowadza uzupełnienia umożliwiające czystą przenośność (tzw. *skórki*).

Historia

Pierwszym linuksowym systemem czasu rzeczywistego o charakterze twardym był RTLinux (1996). Wkrótce autorzy rozdzielili projekt na ścieżkę otwartą (GPL) i komercyjną. Obecnie rozwijana jest tylko wersja komercyjna.
W roku 2000 powstał projekt RTAI korzystający z rozwiązań RTLinuksa. Problemy wynikające z licencji RTLinuksa zmusiły twórców RTAI do zmiany oprogramowania odpowiedzialnego za emulację warstwy sprzętowej (HAL).
W roku 2003 powstał Adeos, inna warstwa abstrakcji, która zapewniła niezbędną bazę wirtualizacji sprzętu dla RTAI.
W roku 2001 pojawił się Xenomai ze swoim mikrojądrem czasu rzeczywistego — Nucleus. Oba projekty miały wspólny cel — wyjście z przestrzeni jądra i sprostanie twardym wymaganiom w przestrzeni użytkownika. Z ich fuzji powstał RTAI/Fusion.
W 2005 roku projekty się rozdzieliły i postanowiły podążać własnymi drogami. Prace stworzone w ramach RTAI/Fusion są kontynuowane pod szyldem Xenomai, natomiast RTAI przemianował się na RTAI/LXRT.



RT_Preempt patch

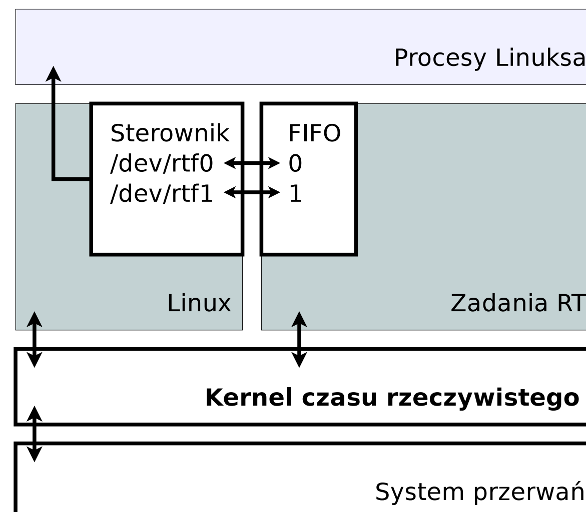
Linux może spełniać wymagania czasu rzeczywistego na dwa sposoby. Pierwszym jest doprowadzenie do pełnej wywłaszczalności jądra poprzez zmiany w nim samym, natomiast drugi dotyczy modelu dodatkowego mikrojądra czasu rzeczywistego i wykonywanie jądra Linuksa jako jednego z zadań. Prace prowadzone nad łatą RT_Preempt podążają pierwszą drogą. Oprócz wywłaszczalnego jądra, modyfikacji w obsłudze przerwań wprowadza ona dodatkowe timery wysokiej rozdzielczości.

Rozwiązanie to nie wymaga dodatkowego API, programy są zwykłymi procesami Linuksa. Jako takie mogą być debugowane oraz analizowane jak inne programy, zapewniona jest również ochrona pamięci.

Reszta tego opracowania koncentruje się na drugim podejściu, tzn. mikrojądra czasu rzeczywistego. Pierwszym systemem tego typu był RTLinux.

RTLinux

RTLinux składa się z dwóch elementów: mikrojądra czasu rzeczywistego, oraz samego Linuksa. Systemy te nie istnieją jednak na równi. Bezpośredni dostęp do rzeczywistych przerwań (sprzętowych, zegara) ma tylko mikrojądro, natomiast po stronie Linuksa przerwania są emulowane. Najważniejszym elementem mikrojądra jest planista, który jest odpowiedzialny za szeregowanie zadań czasu rzeczywistego. W tym układzie Linux również jest zadaniem czasu rzeczywistego działającym pod kontrolą mikrojądra. Dodatkowo posiada najniższy priorytet co sprawia, że jest tzw. procesem bezczynności (*idle task*), czyli wykonuje się tylko wtedy, gdy inne zadania akurat nie korzystają z czasu procesora. Oznacza to, że żadne zadanie po stronie Linuksa nie może wpłynąć na działanie całego systemu, a dzięki emulacji przerwań jest pewność, że inne zadania czasu rzeczywistego otrzymają je pierwsze.



RTLinux posiada modułową architekturę, każdy z elementów systemu, np. moduł kolejek FIFO, może być załadowany, bądź odłączony w ramach potrzeb. Algorytmy szeregowania również są modułami, dzięki czemu w względnie łatwy sposób można dodać kolejne.

Programy realizujące zadania czasu rzeczywistego mogą być uruchamiane tylko z przestrzeni adresowej jądra, dlatego muszą być kompilowane i ładowane jako jego moduły. Narzuca to pewne ograniczenia co do API i bibliotek, zasady są takie jak przy pisaniu modułów jądra. Niestety przez to nie można używać standardowych debuggerów, a błąd w programie może oznaczać zawieszenie się, bądź nawet uszkodzenie całego systemu.

Jednak RTLinux dostarcza szereg mechanizmów komunikacji międzyprocesowej, w tym komunikacji pomiędzy zadaniami RT, a zwykłymi procesami Linuksa. Ograniczenie wynikające z ładowania programów jako modułów narzuca logiczny podział, w którym zadania RT zajmują się tylko ściśle określonymi działaniami (np. akwizycja danych) po czym wyniki wysyłają do programu wizualizacyjnego. Taka komunikacja może się odbywać za pomocą kolejki FIFO, która po stronie Linuksa widziana jest jako plik urządzenia i tak też może być odczytywana. Mechanizmy komunikacji po stronie Linuksa są nieblokujące, więc praktycznie program odczytujący dane może być nawet skryptem powłoki, bądź Pythona i nie wpłynie to na jakość pobieranych wyników.

System przerwań

System operacyjny jest napędzany przez system przerwań, które można uważać za puls komputera.

- Wszystkie programy działające w systemie operacyjnym są planowane przez planistę, który jest uruchamiany przez przerwanie zegarowe (tik).
- Wykonujący się program może zablokować się w oczekiwaniu na jakieś zasoby, lub dobrowolnie zwolnić procesor. W takim przypadku planista jest informowany poprzez przerwanie programowe (wywołanie systemowe).
- Sprzęt może generować przerwania w celu przerwania normalnej pracy systemu operacyjnego dla szybkiej obsługi zdarzenia sprzętowego.

Obsługa przerwań

RT Linux używa systemu przerwań dla zapewnienia mikrojądru priorytetu wyższego od jądra Linuksa.

- Kiedy pojawia się przerwanie, jest ono przekazywane do mikrojądra czasu rzeczywistego, a nie do jądra Linuksa. Jednak przerwania są przechowywane, i następnie przekazywane jądru Linuksa, gdy mikrojądro czasu rzeczywistego skończyło swoją obsługę.
- Ponieważ mikrojądro jest pierwsze, może uruchamiać swoje zadania zgodnie z otrzymanymi przerwaniami.
- Dopiero gdy mikrojądro czasu rzeczywistego nie ma nic do roboty, zapamiętane przerwania są przekazywane do jądra Linuksa.
- Jako drugie, jądro Linuksa może planować własne procesy zgodnie z otrzymanym przerwaniem.

Obsługa przerwań (cd.)

Kiedy w czasie działania normalnego programu pojawia się nowe przerwanie:

- najpierw jest ono obsługiwane przez procedurę obsługi przerwań mikrojądra czasu rzeczywistego,
- typowo procedura obsługi przerwań mikrojądra wywołuje zadanie czasu rzeczywistego realizujące rzeczywistą obsługę przerwania (mikrojądro nazywa się mikrojądrem, bo usunięto z niego wiele procedur, przeniesionych do zwykłych programów),
- natychmiast po obsłudze przerwania (a właściwie jej zainicjowaniu w postaci zadania), uruchamiany jest planista zadań czasu rzeczywistego,
- planista czasu rzeczywistego zauważa, że istnieje zadanie czasu rzeczywistego gotowe do uruchomienia, więc wywłaszcza jądro Linuksa, i uruchamia gotowe zadanie czasu rzeczywistego.

Aby mikrojądro czasu rzeczywistego i zwykłe jądro Linuksa mogły współistnieć na jednym komputerze, musi istnieć szczególny mechanizm przekazywania przerwań pomiędzy tymi jądrami. Każda wersja RT Linux realizuje to na swój sposób. Np. Xenomai używa potoku przerwań tzw. Adeos.

Adeos

Adeos jest warstwą wirtualizacji zasobów w postaci łaty na jądro Linuksa. Stanowi ona platformę, na której można budować systemy czasu rzeczywistego. Umożliwia wykonywanie normalnego środowiska GNU/Linux i systemu RTOS, jednocześnie na tym samym sprzęcie.

Adeos wprowadza pojęcie domen, które istnieją jednocześnie na tej samej maszynie. Domeny nie muszą widzieć siebie nawzajem, ale każda z nich widzi Adeos. Domeny zwykle zawierają kompletne systemy operacyjne.

Podstawową strukturą Adeosa jest kolejka zdarzeń (*event pipeline*).

`http://www.xenomai.org/documentation/branches/v2.4.x/pdf/Life-with-Adeos-rev-B.pdf`

RTAI

RTAI oraz Xenomai były przez długi czas rozwijane jako jeden projekt. Dlatego są one dość podobne pod względem budowy i właściwości. Więcej na temat Xenomai będzie dalej, natomiast tu będą przedstawione główne różnice tych rozwiązań.

Wspólnym celem RTAI/Xenomai było stworzenie możliwości pisania programów RT w przestrzeni użytkownika. W zamian za niewielkie straty wydajności związane z przełączaniem kontekstu można uzyskać korzyści tj. ochrona pamięci, dostęp do bibliotek użytkownika czy debugowanie. Taki program po zaprojektowaniu i przetestowaniu może zostać bezproblemowo przeniesiony do przestrzeni jądra, gdyż API jest niezależne od kontekstu. Moduł o nazwie LXRT zapewnia właśnie taką funkcjonalność. Pozwala zwykłemu użytkownikowi (nie root) na wywoływanie funkcji systemowych RTAI.

RTAI udostępnia własny interfejs programistyczny zawierający podstawowe mechanizmy do komunikacji i synchronizacji:

- Kolejki FIFO
- Pamięć wspólna
- Komunikaty (Mailboxes)
- Semafory
- Zdalne wywołanie procedur (RPC)

Oprócz tego zapewnia zgodność z następującymi standardami POSIX:

1003.1c Wątki (Pthreads), muteksy i zmienne warunkowe
1003.1b Kolejki (Pqueues)

RTAI jest wykorzystywane w środowisku kontroli i akwizycji danych o nazwie RTAI-Lab. Składa się ono dodatkowo z oprogramowania Scicos/Scilab. Razem pozwala to na graficzne tworzenie aplikacji czasu rzeczywistego tworzących cały system akwizycji pomiarów i kontrolowania urządzeń zewnętrznych.

Xenomai

Projekt Xenomai został uruchomiony w sierpniu 2001 roku. Xenomai opiera się na abstrakcyjnym rdzeniu RTOS, dostarczającym zestaw ogólnych usług RTOS:

- timery
- bufory IPC
- zmienne warunkowe
- flag zdarzeń
- usługi sterty pamięci
- obsługa przerwań
- muteksy
- potoki
- kolejki komunikatów
- semafory
- obsługa wątków
- obsługa timerów

Różne interfejsy programistyczne (skórki)

Porównując podejście Xenomai do RTAI można powiedzieć, że o ile RTAI koncentruje się na zapewnieniu najmniejszych technicznie możliwych opóźnień, Xenomai uwzględnia również przenośność kodu.

Lista funkcji zapewniających dostęp do usług Xenomai stanowi tzw. *native API*, to znaczy interfejs bezpośredni. Jednak istnieją inne specyfikacje interfejsów funkcji czasu rzeczywistego, jak np. interfejs POSIX, realizowany przez wiele systemów. System Xenomai zapewnia alternatywny interfejs POSIX do tych samych usług. Oznacza to, że usługi RTOS Xenomai można również wywoływać przez funkcje POSIX. Na przykład, wywołanie funkcji utworzenia muteksu `pthread_mutex_create` w systemie Xenomai spowoduje utworzenie muteksu czasu rzeczywistego Xenomai.

System Xenomai rozciągnął tę koncepcję dalej, tworząc szereg alternatywnych interfejsów programistycznych do swoich usług RTOS. Te interfejsy nazywane są w terminologii Xenomai skórkami (*skins*). Pozwalają one przenosić programy napisane w innych systemach RTOS i kompilować je w systemie Xenomai bez modyfikacji. (Jest to tylko możliwe o tyle o ile dany program korzysta w sposób standardowy z danego mechanizmu RTOS, a nie używa jakichś specyficznych mechanizmów danego systemu operacyjnego.)

Przykładowe skórki realizowane w Xenomai:

- *native* — bezpośredni interfejs do funkcji Xenomai też jest traktowany jako skórka
- POSIX
- PSO+
- VxWorks
- VRTX
- uITRON
- RTAI: tylko w wątkach jądra

Zadania w systemach grupy RT Linux

Program w systemie RT Linux tworzy wątki czasu rzeczywistego. Zwykłe programy linuksowe (nie-realtime) używają zwykłych wątków Pthread.

Zadanie czasu rzeczywistego może być wykonywane w przestrzeni jądra wykorzystując moduł mikrojądra. Zadanie wykonywane w module jądra ma niższe narzuty, ale jakikolwiek błąd w takim zadaniu może zakłócić pracę mikrojądra, albo „wywalić'' cały system.

Zadanie czasu rzeczywistego może być również wykonywane w przestrzeni użytkownika za pomocą zwykłego programu w C. Daje to większe bezpieczeństwo, ponieważ błąd w programie użytkownika może jedynie zawiesić ten program.

Ponadto, programy uruchamiane w przestrzeni użytkownika mogą korzystać z interfejsu funkcji czasu rzeczywistego, jak również ze zwykłych funkcji Linuksa. (W trybie jądra można korzystać tylko z interfejsu funkcji czasu rzeczywistego i funkcji mikrojądra.)

Jednak podczas korzystania przez zadania przestrzeni użytkownika z API Linuksa, nie mogą być one planowane przez planistę czasu rzeczywistego, tylko muszą być planowane przez zwykłego Linuksa. Nie będą więc one zadaniami *HARD real-time*, lecz *SOFT real-time*. Po zakończeniu wywołania funkcji z API Linuksa, zadanie powraca do planisty czasu rzeczywistego.

Zadania w systemie Xenomai

Planista czasu rzeczywistego systemu Xenomai realizuje algorytm planowania oparty na priorytetach z wywłaszczaniem. W ten sposób realizowany jest model wieloprocesowości. Mechanizmy wieloprocesowości są głównymi usługami systemu Xenomai.

Zadania czasu rzeczywistego Xenomai są podobne do watków Linuksa. Tworzone są funkcją `rt_task_create`. W chwili utworzenia zadanie jest w zasadzie tylko deskryptorem zadania, i musi być uruchomione funkcją `rt_task_start`. Można jednocześnie stworzyć i od razu uruchomić zadanie funkcją `rt_task_spawn`.

Typowymi operacjami wykonywanymi przez zadania, mającymi związek z planowaniem zadań są: zawieszanie i wznawianie, oraz usypianie.

Tworzenie zadania

Podczas tworzenia zadania czasu rzeczywistego w Xenomai struktura `RT_TASK` jest używana jako deskryptor tego zadania. Służy ona do przechowywania wszelkich informacji na temat zadania, takich jak:

- funkcja do wykonania przez zadanie czasu rzeczywistego,
- przekazywany do niej argument,
- rozmiar stosu przeznaczonego na jego zmienne,
- jego priorytet,
- czy będzie używał arytmetyki zmiennoprzecinkowej,
- handler sygnału, która zostanie wywołany, gdy zadanie stanie się aktywne.

Zadanie jest tworzone poprzez wywołanie:

```
int rt_task_create(RT_TASK *task, const char *name,
                   int stack_size, int priority, int mode)
```

`task` jest wskaźnikiem do struktury `RT_TASK`, która musi być wcześniej utworzona i wypełniona.
`name` jest napisem stanowiącym symboliczną nazwę zadania. Tej symbolicznej nazwy można użyć aby pobrać strukturę zadaniową przez wywołanie funkcji `rt_task_bind()`.

`stack_size` jest rozmiarem stosu, który może być wykorzystany przez nowe zadanie.
`priority` jest priorytetem zadania od 1 do 99.
`mode` to wektor flag, które wpływają na zadanie:

- `T_FPU` — umożliwia zadaniu używanie FPU (jednostka zmiennoprzecinkowa), jeśli jest dostępna (flaga wymuszana dla zadań przestrzeni użytkownika)
- `T_SUSP` powoduje uruchomienie zadania w trybie zawieszonym; w takim przypadku, wątek będzie musiał być jawnie wznowiony za pomocą `rt_task_resume()`,
- `T_CPU(CPUID)` określa, że nowe zadanie ma działać na procesorze nr CPUID (od 0 do `RTHAL_NR_CPUS-1` włącznie).
- `T_JOINABLE` (tylko w przestrzeni użytkownika) pozwala innemu zadaniu na oczekiwanie na zakończenie nowo tworzonego zadania, oznacza to, że `rt_task_join()` będzie wywołane po zakończeniu tego zadania dla wyczyszczenia wszelkich zaalokowanych zasobów przestrzeni użytkownika.

Uruchamianie zadania

Zadanie jest uruchomiane przez wywołanie:

```
int rt_task_start(RT_TASK *task, void (* task_func) (void *), void *arg)
```

task jest wskaźnikiem do struktury typu `RT_TASK`, która musi być już zainicjowana przez wywołanie `rt_task_create()`
`task_func` jest funkcją do wykonania przez to zadanie czasy rzeczywistego
arg jest argumentem typu void pointer przekazywanym do funkcji zadania

Jeśli flaga `T_SUSP` została podana jako flaga trybu `rt_task_create()`, to zadanie zostanie uruchomione w trybie zawieszonym. W takim przypadku należy jawnie wznowić to zadanie funkcją `rt_task_resume()`. W przeciwnym wypadku, zadanie czasu rzeczywistego staje się gotowe do wykonania natychmiast.

Przykładowy program Xenomai

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/timer.h>

#include  <rtdk.h>
RT_TASK demo_task;

void demo(void *arg)
{
  RT_TASK *curtask;
  RT_TASK_INFO curtaskinfo;

  // hello world
  rt_printf("Hello World!\n");

  // inquire current task
  curtask=rt_task_self();
  rt_task_inquire(curtask,&curtaskinfo);
```

```
  // print task name
  rt_printf("Task name : %s \n", curtaskinfo.name);
}

int main(int argc, char* argv[])
{
  char str[10] ;

  // Perform auto-init of rt_print buffers if the task doesn't do so
  rt_print_auto_init(1);

  // Lock memory : avoid memory swapping for this program
  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_printf("start task\n");

  /*
   * Arguments: &task,
   *            name,
   *            stack size (0=default),
   *            priority,
   *            mode (FPU, start suspended, ...)
   */
  sprintf(str,"hello");
  rt_task_create(&demo_task, str, 0, 50, 0);
  /*
```

```
   * Arguments: &task,
   *            task function,
   *            function argument
   */
  rt_task_start(&demo_task, &demo, 0);
}
```

Blokowanie stron pamięci

W programie powyżej, pamięć jest zablokowana przez wywołanie

```
mlockall ( MCL_CURRENT | MCL_FUTURE);
```

Powoduje to wyłączenie mechanizmu pamięci wirtualnej dla tego zadania,
i przydzielone mu strony pamięci fizycznej nie będą podlegały wymianie na dysk.
Jest to praktycznie konieczne dla wszystkich zadań czasu rzeczywistego, ponieważ
operacje pamięci wirtualnej trwają zbyt długo, aby mogły być zachowane
jakiekolwiek wymagania czasu rzeczywistego.

Xenomai generuje ostrzeżenie, kiedy ten krok jest pominięty w programie.

Zadania jednorazowe i okresowe

Systemy czasu rzeczywistego często wymagają okresowego wykonywania pewnych zadań. Ich wielokrotne każdorazowe indywidualne uruchamianie wprowadzałoby narzuty na tworzenie tych zadań, oraz konieczność utworzenia dodatkowego zadania do ich uruchamiania.

Zamiast tego, stosuje się zadania okresowe, które istnieją cały czas, i są okresowo wznawiane przez system.

Tworzenie zadań okresowych

Zadania tworzone funkcją `rt_task_start` są domyślnie zadaniami jednorazowymi (*one-shot tasks*). Można przekształcić je na okresowe za pomocą funkcji `rt_task_set_periodic()`. Zadaje ona czas wystartowania zadania (który można podać za pomocą makra `TM_NOW`), oraz jego okres.

```
const long long SLEEP_TIME = 1000000000; /* = 1 s */
/* ... */
rt_task_create(&task1, "task1", 0, 50, 0);
rt_task_set_periodic(&task1, TM_NOW, SLEEP_TIME);
rt_task_start(&task1, &func1, NULL);
```

Przekształcenie zadania na okresowe musi nastąpić albo przed jego uruchomieniem, albo musi to zrobić samo zadanie (może wtedy podać wskaźnik do opisu zadania jako `NULL`).

Gdy okresowe zadanie kończy swoją pracę w danym uruchomieniu, musi jawnie zwolnić procesor wywołując funkcję `rt_task_wait_period`

Przykład zadania okresowego

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/timer.h>

RT_TASK demo_task;

/* NOTE: error handling omitted. */

void demo(void *arg)
{
  RTIME now, previous;

  /*
   * Arguments: &task (NULL=self),
   *            start time,
   *            period (here: 1 s)
   */
  rt_task_set_periodic(NULL, TM_NOW, 1000000000);
  previous = rt_timer_read();
```

```
  while (1) {
    rt_task_wait_period(NULL);
    now = rt_timer_read();

    /*
     * NOTE: printf may have unexpected impact on the timing of
     *       your program. It is used here in the critical loop
     *       only for demonstration purposes.
     */
    printf("Time since last turn: %ld.%06ld ms\n",
           (long)(now - previous) / 1000000,
           (long)(now - previous) % 1000000);
    previous = now;
  }
}

void catch_signal(int sig)
{
}

int main(int argc, char* argv[])
{
  signal(SIGTERM, catch_signal);
  signal(SIGINT, catch_signal);
```

```
  /* Avoids memory swapping for this program */
  mlockall(MCL_CURRENT|MCL_FUTURE);

  /*
   * Arguments: &task,
   *            name,
   *            stack size (0=default),
   *            priority,
   *            mode (FPU, start suspended, ...)
   */
  rt_task_create(&demo_task, "trivial", 0, 99, 0);

  /*
   * Arguments: &task,
   *            task function,
   *            function argument
   */
  rt_task_start(&demo_task, &demo, NULL);

  pause();

  rt_task_delete(&demo_task);
}
```

Biblioteka drukowania czasu rzeczywistego rtdk

Zauważ, że w powyższym programie użyto funkcji `rt_printf()` zamiast `printf()`. Nie użyto `printf()` ponieważ używa ona wywołania funkcji Linuksa i dlatego znacznie spowalnia zadanie czasu rzeczywistego. Zamiast tego można używać funkcji czasu rzeczywistego `rt_printf()` biblioteki rtdk. Jest to osadzona w przestrzeni użytkownika Xenomai biblioteka usług printf. W rzeczywistości nie zależy ona nawet od żadnych usług Xenomai, tylko wykorzystuje czyste API POSIX. API biblioteki `librtprint` wygląda podobnie jak strony podręcznika printf(3):

```
rt_vfprintf
rt_vprintf
rt_fprintf
rt_printf
```

Podstawową ideą librtdk jest utrzymanie drukowania tak tanio, jak to możliwe — nie są używane żadne blokady ani żadne funkcje systemowe. Każdy wątek, który używa `rt_printf()` i podobnych ma swój lokalny bufor kołowy. Centralny wątek wyjściowy (zwykły, nie czasu rzeczywistego; jest to jeden wątek na proces) nadzoruje przekazywanie treści wszystkich buforów kołowych wątków do strumieni wyjściowych. Kolejność komunikatów zostanie zachowana (zgrubnie) przez globalny licznik sekwencji. Wątek wyjściowy wykorzystuje okresowe odpytywanie (domyślnie: 10 Hz), aby uniknąć potrzeby stosowania specjalnych mechanizmów sygnalizacji.

Aby uruchomić wątek wyjściowy (non-RT) zadanie musi wywołać na początku następującą funkcję Xenomai:

```
rt_print_auto_init(1);
```

Ta funkcja inicjalizuje bufory `rt_print` i uruchamia wątek wyjściowy.

Real-Time Driver Model

Dla konkretnych układów peryferyjnych konieczne jest napisanie dedykowanych sterowniki czasu rzeczywistego. Obecnie są dostępne drivery dla RS232, CAN, FireWire i kart pomiarowych National Semiconductors.

http://www.xenomai.org/documentation/branches/v2.4.x/pdf/RTDM-and-Applications.pdf

Kompilacja programów Xenomai

Aby skompilować dowolny program Xenomai, należy najpierw uzyskać CFLAGS i LDFLAGS przez:

```
$ xeno-config --xeno-cflags
$ xeno-config --xeno-ldflags
```

Aby ustawić te flagi, można użyć:

```
$ export CFLAGS=‘xeno-config --xeno-cflags‘
$ export LDFLAGS=‘xeno-config --xeno-ldflags‘
```

Skompilowanie programu ex01.c z „natywną” biblioteką implementującą Xenomai API i biblioteką druku rtdk:

```
$ gcc $CFLAGS $LDFLAGS -lnative -lrtdk ex01.c -o ex01

$ export LD_LIBRARY_PATH=/usr/xenomai/lib
```

Uruchomienie programu:

```
$./Ex01
```

Ćwiczenia

Ćwiczenie 1a Skompiluj i uruchom program ex01.c, aby uzyskać pewne podstawowe doświadczenie w pracy z zadaniami Xenomai. Najpierw spróbuj kompilacji ręcznej za pomocą polecenia podanego powyżej. Następnie spróbuj zbudować program binarny w sposób zautomatyzowany za pomocą dostarczonego Makefile (za pomocą polecenia „`make ./ex01`”).

Ćwiczenie 1b Wprowadź błąd, wykomentowując przypisanie identyfikatora bieżącego zadania w następujący sposób:

```
// Curtask = rt_task_self();
```

Zapisz wynikający z niego program jako ex01b.c. Skompiluj i uruchom ten program. Obserwuj wynik.

Przykłady programów

user_alarm.c — tworzenie alarmów
http://www.xenomai.org/documentation/xenomai-2.6/html/api/user_alarm_8c-example.html

event_flags.c — tworzenie zdarzeń i synchronizacja
http://www.xenomai.org/documentation/xenomai-2.6/html/api/event_flags_8c-example.html

msg_queue.c — kolejki komunikatów, pozwalają przesyłać dowolnej długości komunikaty od nadawcy do odbiorcy:
http://www.xenomai.org/documentation/xenomai-2.6/html/api/msg_queue_8c-example.html

potoki komunikatów (*message pipe*) — pozwalają przesyłać dowolnej długości komunikaty między zadaniami RT a zwykłymi procesami Linuksa:
http://www.cs.ru.nl/lab/xenomai/exercises/ex05/Exercise-5.html

szeregowanie zadań — priorytetowe:
http://www.cs.ru.nl/lab/xenomai/exercises/ex06/Exercise-6.html

szeregowanie zadań — rotacyjne:
http://www.cs.ru.nl/lab/xenomai/exercises/ex07/Exercise-7.html