helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

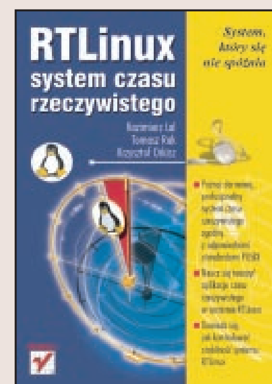
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

RTLinux – system czasu rzeczywistego



Autorzy: Kazimierz Lal, Tomasz Rak, Krzysztof Orkisz
ISBN: 83-7197-898-7
Format: B5, stron: 116

Systemy czasu rzeczywistego stanowią specyficzną, ale bardzo ważną dziedzinę współczesnej informatyki. Złożoność samego przetwarzania w czasie rzeczywistym oraz fakt, że oprogramowanie czasu rzeczywistego jest implementowane najczęściej w niewidocznych, najniższych warstwach systemów informatycznych, sprawia, że te elementy są ukryte przed przeciętnymi użytkownikami, a wiedza na ich temat jest niewielka i trudno dostępna.

Typowymi i najliczniejszymi przedstawicielami systemów czasu rzeczywistego są systemy sterowania procesów przemysłowych, często należące do klasy systemów określanych jako wbudowane. Ich zadaniem jest sterowanie liniami technologicznymi, obrabiarkami, robotami, układami napędowymi itp. W życiu codziennym systemy te stosowane są stosowane w nowoczesnych aparatach fotograficznych do określania parametrów ekspozycji, w samochodach do sterowania bezpośrednim wtryskiem paliwa oraz w osobistych komputerach kieszonkowych.

Książka opisuje RTLinux, darmowy system operacyjny czasu rzeczywistego oparty na Linuksie. Nowoczesna architektura i szeroki wybór platform, na których pracuje, czyni go interesującym dla wszystkich osób potrzebujących takiego systemu. W książce znajdziesz opisane:

- Architekturę Linuxa i modyfikacje wprowadzone w systemie RTLinux
- Instalację i konfigurację RTLinuxa
- Tworzenie aplikacji czasu rzeczywistego
- Prosty system pomiarowy czasu rzeczywistego

Uzupełnieniem książki są dodatki opisujące dokładnie pełną listę funkcji implementowanych przez RTLinux, a także przykłady ich praktycznego wykorzystania.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wstęp ... **5**

Rozdział 1. Linux a czas rzeczywisty ... **11**

Jądro systemu ... 11
Szeregowanie procesów ... 12
Rozdzielczość szeregowania ... 12
Wywołania systemowe ... 13
Przerwania sprzętowe ... 13
Pamięć wirtualna ... 13
Optymalizacja wykorzystania zasobów systemowych ... 14

Rozdział 2. RTLinux ... **15**

Podstawowe założenia ... 15
Architektura systemu ... 15
Wirtualny system przerwań ... 17
Zadania czasu rzeczywistego ... 18
Szeregowanie zadań ... 19
Odmierzanie czasu ... 21
Komunikacja międzyprocesowa ... 23
- Kolejki czasu rzeczywistego ... 23
- Pamięć dzielona ... 24
- Synchronizacja i wzajemne wykluczanie ... 24

Rozdział 3. Instalacja i konfiguracja systemu RTLinux ... **27**

Pakiet ... 27
- Historia ... 27
- Autorzy ... 28
- Licencja ... 29
- Zawartość ... 30
- Mini-RTL ... 31

Przebieg instalacji ... 32
- Wybór dystrybucji ... 32
- Wymagania sprzętowe ... 33
- Wymagania dla środowiska programistycznego ... 34
- Źródła pakietów ... 36
- Przebieg instalacji ... 37

Kompilacja jądra ... 38
Konfiguracja i kompilacja RTLinuksa ... 40

Moduły czasu rzeczywistego ... 43
Uruchamianie modułów systemowych ... 43
Uruchamianie modułów czasu rzeczywistego użytkownika ... 47

Rozdział 4. Aplikacje czasu rzeczywistego ... 49
Standard POSIX ... 49
Struktura systemu RT-Linux ... 54
Źródła pomocy przy programowaniu zadań czasu rzeczywistego ... 56
Kompilacja modułów ... 57
Przykładowe problemy spotykane w aplikacjach RTLinuksa ... 57
Aplikacja „Hello World!” ... 57
Zadania czasu rzeczywistego ... 59
Kolejki czasu rzeczywistego ... 60
Obsługa przerwań systemowych ... 63
Obsługa sygnałów czasu rzeczywistego w procesach Linuksa ... 64
Zegar i funkcje konwersji czasu ... 66
Pamięć dzielona ... 68
Mechanizmy synchronizacji międzyzadaniowej ... 69

Rozdział 5. Prosty system pomiarowy czasu rzeczywistego ... 73
Multimetr METEX 3650CR — dane techniczne ... 73
Komunikacja z multimetrem ... 74
Sterownik portu szeregowego ... 75
Moduł obsługi multimetru ... 78
Uwagi końcowe ... 85

Rozdział 6. Podsumowanie ... 87

Dodatek A Pełna lista funkcji implementowanych przez system RTLinux ... 89
Funkcje charakterystyczne dla systemu ... 89
Podzbiór funkcji interfejsu POSIX, implementowany przez wersję 3.1 ... 91
Opcjonalne funkcje POSIX zależne od konfiguracji ... 93
Zmienne warunkowe POSIX ... 93
Semafory POSIX ... 93
Funkcje o ograniczonym zakresie używania ... 94

Dodatek B Przykładowe programy z wykorzystaniem API systemu RTLinux ... 95
Przechwytywanie przerwań ... 95
Sygnały czasu rzeczywistego w procesach Linuksa ... 97

Dodatek C Architektura i386, a wielozadaniowe systemy operacyjne ... 99
Zarządzanie pamięcią w trybie chronionym ... 99
Segmentacja ... 99
Stronicowanie (pamięć wirtualna) ... 101
Ochrona ... 102
Przerwania i obsługa wyjątków ... 103
Zarządzanie zadaniami ... 105

Bibliografia ... 109

Skorowidz ... 111

Rozdział 2.
RTLinux

Podstawowe założenia

Dosyć oczywistym rozwiązaniem, eliminującym wcześniej wymienione wady, wydaje się być modyfikacja jądra Linuksa. Taką drogę właśnie wybrali twórcy systemu KURT (*http://www.ittc.ku.edu/kurt/*). Konieczne zmiany to między innymi: wprowadzenie wywłaszczającego szeregowania procesów o stałym priorytecie, zwiększenie rozdzielczości szeregowania i wprowadzenie zegara wyższej rozdzielczości. Osiągnięty kompromis pozwolił na stworzenie systemu o łagodnych (ang. *firm*) ograniczeniach czasowych.

Całkowicie inną drogę do osiągnięcia cech systemu operacyjnego (ale o twardych ograniczeniach czasowych — ang. *hard real-time operating system*) wybrali twórcy RTLinuksa [2, 3]. Inspiracją w tym wypadku była architektura eksperymentalnego systemu MERT, zbudowanego przez badaczy z Bell Labs w latach siedemdziesiątych. W zamierzeniach system miał mieć możliwość uruchamiania zarówno aplikacji czasu rzeczywistego, jak i zwykłych programów. Intencją projektantów systemu MERT było stworzyć nie jeden system operacyjny, który wspiera obydwa typy przetwarzania, lecz sprawić, aby system czasu rzeczywistego i system ogólnego przeznaczenia współistniały razem. Twierdzili oni, że ... *dostępność wyrafinowanego systemu ogólnego przeznaczenia na tym samym komputerze, co system czasu rzeczywistego, dostarcza potężnego narzędzia, które może być wykorzystane w projektach interfejsu człowiek-maszyna dla aplikacji czasu rzeczywistego...* [1].

Architektura systemu

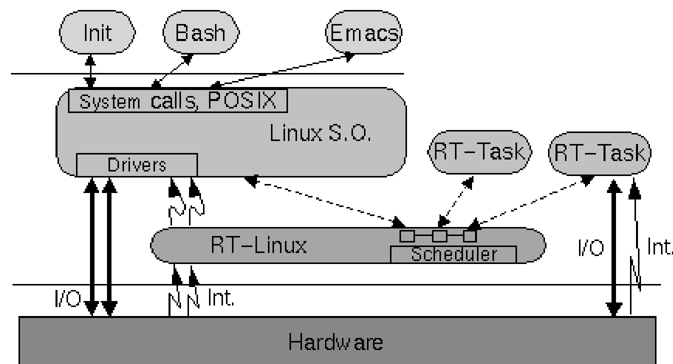
Bazując na ideach systemu MERT — RTLinux oddziela mechanizmy systemu operacyjnego czasu rzeczywistego od systemu operacyjnego ogólnego zastosowania. RTLinux działa traktując zwykłe jądro Linuksa jako zadanie pod kontrolą niewielkiego i prostego systemu operacyjnego czasu rzeczywistego. W istocie, Linux jest zadaniem tła (ang. *idle task*) dla RTLinuksa, wykonywanym jedynie wtedy, gdy żadne z zadań czasu rzeczywistego nie ubiega się o procesor. Z założenia zadanie Linuksa nigdy nie może zablo-

kować przerwań i zapobiec wywłaszczeniu siebie. Technicznym kluczem do osiągnięcia tego jest dodanie programowej warstwy emulującej sprzętowy mechanizm kontroli przerwań. Linux nigdy nie może zablokować przerwań sprzętowych. Kiedy podejmuje taką próbę, część czasu rzeczywistego przechwytuje ten fakt, zaznacza odpowiednio i oddaje sterowanie z powrotem do jądra Linuksa. Niezależnie od trybu, Linuksowi nie pozwala się na zwiększenie opóźnienia odpowiedzi na przerwanie czasu rzeczywistego. Kiedy zostaje zgłoszone przerwanie, jądro RTLinuksa przechwytuje je i decyduje, co z nim zrobić. Jeśli aktualnie istnieje procedura obsługi przerwania, pochodząca z zadania czasu rzeczywistego — zostaje wywołana. Kiedy przerwanie jest obsługiwane przez Linuksa (lub współdzielone z nim), jest oznaczane jako oczekujące. Po wykryciu próby włączenia przerwań przez jądro Linuksa wszystkie oczekujące przerwania są emulowane i wywoływane są odpowiednie procedury obsługi.

Bez znaczenia jest to, w jakim trybie działa Linux. W trybie użytkownika, w trybie systemowym czy nawet w sekcji krytycznej jądra, RTLinux jest w stanie zareagować na przychodzące przerwania. Techniczne aspekty dyskutowanych rozwiązań poddane są szczegółowej analizie w podrozdziale „Wirtualny system przerwań".

RTLinux separuje mechanizmy jądra czasu rzeczywistego i mechanizmy jądra zwykłego systemu (rysunek 2.1). Tak więc każdy z osobna może być optymalizowany niezależnie. Jest tak zaprojektowany, że wyeliminowane są sytuacje, w których musi czekać na zwolnienie jakichkolwiek zasobów przez Linuksa. RTLinux nie alokuje pamięci, nie dzieli sekcji krytycznych ani nie synchronizuje żadnych struktur danych, z wyjątkiem sytuacji niezbędnych do współdziałania obydwu systemów.

Rysunek 2.1.
Architektura systemu RTLinux [21]



Mechanizmy komunikacyjne używane do wymiany danych pomiędzy zwykłymi procesami a zadaniami czasu rzeczywistego są nieblokujące po stronie RTLinuksa. Nigdy nie występuje przypadek, że zadanie czasu rzeczywistego czeka na zakolejkowanie lub pobranie danych z kolejki.

Jedną z kluczowych zasad projektowych RTLinuksa jest, aby pozostawić go jak najmniejszym i jak najprostszym. Im mniej spraw do „załatwienia" po stronie RTLinuksa i im więcej po stronie Linuksa, tym lepiej. Tak więc startem systemu, inicjalizacją urządzeń, ładowaniem modułów, systemem plików i dynamicznym przydzielaniem zasobów zajmuje się zwykły system. Zadaniem RTLinuksa jest dostarczenie bezpośredniego dostępu do sprzętu dla wątków czasu rzeczywistego, szeregowanie, dostarczanie mechanizmów odmierzania czasu i technik komunikacji międzyprocesowej.

Wirtualny system przerwań

Jądro Linuksa jest duże i monolityczne. Wielu ludzi zaangażowanych w jego rozwój często używa wyłączania przerwań w celu ochrony sekcji krytycznych. Rodzi to wspomniane już wcześniej reperkusje. Niezbędna korekta takiego stanu rzeczy pociągnęłaby szereg zmian w jądrze, jednak bez gwarancji wystarczająco dobrych wyników dla przetwarzania czasu rzeczywistego.

W systemie RTLinux problem ten został rozwiązany przez Victora Yodaikena [1, 2] poprzez zastosowanie programowej warstwy emulacji pomiędzy jądrem Linuksa a sprzętowym układem kontroli przerwań. Wszystkie wystąpienia makr *cli*, *sti*, *iret* w kodzie źródłowym jądra zostały zastąpione makrami: *S_CLI*, *S_STI*, *S_IRET*. W ten sposób emulator jest w stanie wykryć każdą próbę wyłączenia i włączenia przerwań przez jądro Linuksa.

> *cli* — wyzerowanie znacznika IF zezwolenia na przerwania w rejestrze stanu procesora.
>
> *sti* — ustawienie znacznika IF zezwolenia na przerwania.
>
> *iret* — instrukcja powrotu z procedury obsługi przerwania.

Budowę dwóch pierwszych makr przedstawia listing 2.1 (zapis w konwencji assemblera AT&T).

Listing 2.1. *Budowa makr S_CLI oraz S_STI*

```
S_CLI:
  movl $0, SFIF

S_STI:
  sti
  pushfl
  pushl $KERNEL_CS
  pushl $1f
  S_IRET
1:
```

Makro *S_CLI* powoduje, że zamiast rzeczywistego wyłączenia przerwań przez procesor zerowana jest odpowiednia zmienna w emulatorze. Jeśli nastąpiło przerwanie i zmienna jest ustawiona, emulator wywołuje bezpośrednio procedurę obsługi przerwania ustaloną przez jądro Linuksa. W przeciwnym razie, gdy przerwania są wyłączone, fakt zgłoszenia przerwania jest zapamiętywany bitowo w zmiennej, przechowującej informacje o wszystkich oczekujących przerwaniach. Kiedy Linux na powrót próbuje włączyć przerwania, wszystkie oczekujące przerwania są emulowane. Makro *S_STI* rzeczywiście włącza przerwania i przygotowuje stos procesora jak w przypadku wywołania przerwania: odkłada na niego flagi procesora, rejestr segmentowy jądra i adres powrotu, w tym wypadku adres ten oznaczony jest etykietą *1:*. Makro *S_IRET* wykonuje całą pracę emulatora przerwań (listing 2.2).

Listing 2.2. *Budowa makra S_IRET*

```
S_IRET:
  push %ds
  pushl %eax
  pushl %edx
  movl $KERNEL_DS,%edx
  mov %dx, %ds
  cli
  movl SFREQ, %edx
  andl SFMASK, %edx
  bsfl %edx,%eax
  jz 1f
  S_CLI
  sti
  jmp SFIDT(,%eax,4)
1:
  movl $1, SFIF
  popl %edx
  popl %eax
  pop %ds
  iret
```

Makro najpierw zachowuje używane rejestry i ustawia rejestr segmentowy danych na segment jądra, w celu dostępu do zmiennych globalnych. Następnie za pomocą „maski bitowej" wskazującej wszystkie niezamaskowane przerwania odrzuca te przerwania, które nie mają być wywoływane. Rejestr *edx,* zawierający wynik maskowania, jest przeszukiwany w celu detekcji oczekujących przerwań. Ustawiony bit w omawianym rejestrze powoduje wywołanie odpowiedniej procedury przerwania w jądrze Linuksa. Jeśli nie ma żadnych oczekujących przerwań, wykonywany jest bezpośredni powrót z przerwania. Instrukcja *iret* w procedurze obsługi przerwania, zamieniona przez RTLinuksa na *S_IRET*, spowoduje odnalezienie następnego oczekującego przerwania, aż do wyczerpania całej listy. Wszystkie potencjalne przerwania, które mogą nadejść w czasie między poszukiwaniem oczekującego przerwania a jego wywołaniem, są opóźnione w wywołaniu co najwyżej do następnej instrukcji *S_STI* lub *S_IRET*.

Zadania czasu rzeczywistego

Zadania czasu rzeczywistego są zdefiniowanymi przez użytkownika programami, wykonywanymi pod kontrolą jądra czasu rzeczywistego. Zadania czasu rzeczywistego wprowadzone przez RTLinuksa znacznie różnią się od zwykłych procesów. Odmienność ta objawia się przede wszystkim prostotą i szybkością działania. Zadania RTLinuksa wykonują się we wspólnej przestrzeni adresowej (w przestrzeni jądra) z maksymalnym poziomem uprzywilejowania i bezpośrednim dostępem do urządzeń. Szeregowanie zadań na tym samym poziomie ochrony przy użyciu programowego sposobu przełączania kontekstu zadania daje wiele korzyści. Przełączanie kontekstu zadania jest proste i sprowadza się do zapamiętania i odtworzenia rejestrów procesora. Odpada narzut czasowy związany ze zmianą trybu uprzywilejowania procesora, który podczas przełączania z trybu systemowego do trybu użytkownika zajmuje kilkadziesiąt taktów procesora, podczas gdy inne

instrukcje — poniżej dziesięciu taktów. Niebagatelny jest również czas zyskiwany przy zaniechaniu zmiany rejestru bazowego jednostki zarządzającej pamięcią i związanego z tym czyszczenia rejestrów asocjacyjnych procesora, czyli unieważnienia rejestrów związanych z TLB (ang. *translation lookaside buffer*). Rezygnacja z mechanizmu stronicowania i ochrony pamięci godzi niestety w integralność systemu. Każdy błąd w programie może mieć skutek w postaci zawieszenia całego systemu. Brak ochrony pamięci uniemożliwia wykorzystanie prostych mechanizmów śledzenia programów i znajdowania błędów. Z drugiej strony jednak wspólna przestrzeń adresowa pozwala zadaniom czasu rzeczywistego komunikować się i dzielić dane bezpośrednio (poprzez zmienne globalne) bez konieczności użycia złożonych technik komunikacji międzyprocesowej.

W odróżnieniu od zwykłych procesów Linuksa, procesy czasu rzeczywistego są tzw. „procesami lekkimi" (ang. *light-weight processes*) lub „wątkami" (ang. *threads*) ze względu na szybkość i łatwość przełączania. Nie są jednak wątkami w rozumieniu takim, jak w systemach operacyjnych np. Solaris czy Windows NT. Najbardziej adekwatną nazwą dla tej kategorii procesów jest „zadanie". Taka też nazwa będzie używana dla określenia procesów RTLinuksa. Dodatkowo jednak użycie pojęcia „wątek" w kontekście systemu RTLinux w dalszej części pracy również oznaczać będzie zadanie czasu rzeczywistego.

Mechanizmem, na którym zasadza się cała idea procesów wykonywanych w przestrzeni jądra, jest *mechanizm modułów ładowalnych*, wykorzystywany w prawie każdej nowoczesnej instalacji Linuksa. Technika ta, opcjonalna dla jądra Linuksa, jest krytyczna dla działania RTLinuksa, gdyż wsparcia dla modułów ładowalnych wymagają zarówno poszczególne moduły funkcjonalne tego systemu, jak i programy napisane przez użytkownika. Moduły stanowią „kawałki" jądra w postaci plików obiektowych (skompilowane, ale nieskonsolidowane), które mogą być zarówno dynamicznie linkowane i dołączane do rezydującej w pamięci części jądra podczas działania systemu (bez konieczności czasochłonnego restartu), jak i odłączane od jądra (również bez restartu).

Szeregowanie zadań

Planista (ang. *scheduler*) zajmujący się szeregowaniem zadań w systemie RTLinux stanowi odrębny moduł. Jedynym jego zadaniem jest planowanie przydziału procesora zadaniom według określonego algorytmu. Standardowo RTLinux używa algorytmu szeregowania bazującego na stałych priorytetach zadań, w którym do wykonania wybierane jest zawsze gotowe zadanie o najwyższym priorytecie — *Priority-Based Rate Monotonic Scheduling Algorithm* (RMS). Jeśli istnieje kilka zadań o tym samym priorytecie, do wykonania wybierane jest to zadanie, które w kolejce zadań zostało odnalezione jako pierwsze. Zadaniom nie przydziela się „szczeliny czasowej", tak jak to jest w systemach z podziałem czasu. Zakłada się, że zadanie samo odda procesor lub zostanie wywłaszczone przez zadanie o wyższym priorytecie. W RTLinuksie nic nie stoi na przeszkodzie, aby zaimplementować inny, własny algorytm planowania zadań czasu rzeczywistego. Zmiana algorytmu szeregowania sprowadza się do wprowadzenia zmian w funkcji podejmującej decyzję: `rtl_schedule()` i ewentualnie w strukturze zadania, jeśli występuje taka potrzeba.

W przeciwieństwie do innych znanych systemów operacyjnych, RTLinux nie buduje oddzielnej kolejki procesów gotowych, które czekają na przydział procesora. Wszystkie zadania tworzą jedną kolejkę. Stan konkretnego zadania zapisywany jest w jego strukturze. Od momentu, gdy RTLinux przystosowany został do działania z wykorzystaniem symetrycznej wieloprocesorowości, dla każdego procesora tworzona jest odrębna struktura `rtl_sched_cpu_struct`, zawierająca między innymi kolejkę procesów do wykonania na danym procesorze.

Ważniejsze pola struktury `rtl_sched_cpu_struct` to:

- `struct rtl_thread_struct *rtl_current` — wskaźnik do aktualnie wykonywanego zadania;
- `struct rtl_thread_struct rtl_linux_task` — struktura reprezentująca zadanie Linuksa;
- `struct rtl_thread_struct *rtl_task_fpu_owner` — wskaźnik do struktury zadania obecnie zajmującego koprocesor;
- `struct rtl_thread_struct *rtl_tasks` — kolejka zadań czasu rzeczywistego (jednokierunkowa lista struktur zadań czasu rzeczywistego, zadanie Linuksa również istnieje w tej kolejce);
- `struct rtl_thread_struct *rtl_new_tasks` — lista nowo utworzonych zadań, oczekujących na dodanie do kolejki `rtl_tasks;`
- `clockid_t clock` — zegar używany przez jednostkę szeregującą do odmierzania czasu;
- `int sched_flags` — wewnętrzne flagi planisty, związane między innymi z odmierzaniem czasu.

Struktura zadania `rtl_thread_struct` reprezentuje wszelkie dane potrzebne do zarządzania, szeregowania i przełączania kontekstu. Zawiera ona między innymi następujące pola:

- `struct rtl_sched_param sched_param` — określa atrybuty, według których zadanie jest szeregowane (np. priorytet);
- `RTL_FPU_CONTEXT fpu_regs` — przechowuje kontekst (rejestry) koprocesora;
- `int uses_fp` — pole to wskazuje, czy dane zadanie używa koprocesora, czy też nie; jeśli zadanie nie korzysta z obliczeń zmiennoprzecinkowych, nie jest konieczne zapamiętywanie i odtwarzanie kontekstu koprocesora;
- `int CPU` — zawiera identyfikator procesora, do którego kolejki jest przydzielone zadanie;
- `hrtime_t resume_time` — pole zawiera godzinę „budzenia” dla zadań okresowych; dla zadań nieokresowych ma wartość *HRTIME_INFINITY*;
- `hrtime_t period` — niezerowa wartość w tym polu określa odstęp czasu pomiędzy kolejnymi wznowieniami zadania (zadanie periodyczne);

- `int errno_val` — służy do przechowywania numerów części błędów zaistniałych podczas działania zadania;
- `int magic` — to pole służy jedynie do prostego sprawdzenia poprawności struktury zadania; jego wartość różna od stałej *RTL_THREAD_MAGIC* oznacza, że struktura zadania nie jest prawidłowa;
- `rtl_sigset_t pending` — zmienna ta przechowuje sygnały sterujące stanem zadania: *RTL_SIGNAL_SUSPEND*, *RTL_SIGNAL_WAKEUP*, *RTL_SIGNAL_TIMER*, *RTL_SIGNAL_CANCEL*, *RTL_SIGNAL_READY*;
- `rtl_sigset_t blocked` — przechowuje maskę aktywnych sygnałów.

Schemat algorytmu szeregowania przy użyciu pseudokodu znajduje się na listingu 2.3.

Listing 2.3. *Algorytm szeregowania implementowany przez RTLinuksa*

```
rtl_schedule() {
  for (każdy proces realtime)
    wybierz gotowy proces o najwyższym priorytecie;
  if (wybrano proces) {
    for (każdy wstrzymany proces realtime o wyższym priorytecie niż priorytet
          ➥wybranego procesu)
      wybierz proces, który powinien być wznowiony najwcześniej;
    if (wybrano wstrzymany proces)
      ustaw zegar na czas "budzenia" procesu;
  }
  wznów wybrany proces;
}
```

Wartości priorytetów zadań czasu rzeczywistego można ustalać na etapie ich tworzenia lub później, w trakcie ich działania. Wyższa wartość liczbowa oznacza wyższy priorytet. Prawidłowy zakres wartości priorytetów jest ustalany przez dwie funkcje:

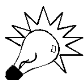
- `sched_get_priority_max(int policy)` — zwraca maksymalną wartość priorytetu; obecnie zwracana wartość jest stała i wynosi 1000000;
- `sched_get_priority_min(int policy)` — zwraca minimalną wartość priorytetu; standardowo funkcja ta zwraca wartość 0 (zero).

Priorytet szczególnego zadania, jakim jest Linux, nie mieści się w podanym zakresie. Jego priorytet, jak już zostało zaznaczone wcześniej, jest najniższy i wynosi –1.

Odmierzanie czasu

Precyzyjne odmierzanie czasu jest bardzo ważne dla prawidłowego działania planisty i zadań czasu rzeczywistego. Pewne zadania potrzebują aktywowania w określonym momencie czasu, oczekiwania na zdarzenie przez określony czas, bądź to uruchamiania co pewien okres. Niedokładność w odmierzaniu czasu powoduje odchylenia od zaplanowanych terminów i jest zjawiskiem bardzo niekorzystnym.

Powodem, dla którego w większości uniwersalnych systemów operacyjnych istnieją zegary o niskiej rozdzielczości, jest użycie okresowych przerwań od sprzętowego układu zegarowego. Niska rozdzielczość jest kompromisem pomiędzy dokładnością zegara a czasem spędzanym przez system na obsłudze przerwań zegarowych. W RTLinuksie zamiast generowania przerwań zegarowych (periodycznych) generowane są przerwania po upływie zadanego czasu (*time-out*). Programowalne kontrolery czasu w komputerach klasy PC używając takiego trybu pracy pozwalają na uzyskanie rozdzielczości czasu na poziomie 1 mikrosekundy. Dodatkowo taki sposób działania zapewnia znaczne ograniczenie kosztów stałych obsługi przerwań zegarowych. Korzyści z tego trybu pracy ilustruje przykład 2.1.

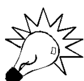


W komputerach PC jest to programowalny układ czasowy Intel 8254.

Przykład 2.1. Wykorzystanie przerwania po upływie określonego czasu

Jeśli jedno zadanie musi być wykonywane co 331 jednostek czasu, a inne co 1 027 jednostek (brak jest dobrego wspólnego podzielnika), to w typowej obsłudze zegara trzeba zliczać poszczególne takty i inicjować zadanie po określonej liczbie taktów. W trybie programowania „na żądanie" (ang. *one-shot mode*) licznik czasowy jest najpierw programowany na wygenerowanie przerwania po 331 jednostkach czasu, a następnie przeprogramowany na następne przerwanie po 691 jednostkach (zakładając, że przeprogramowanie zegara trwa 5 jednostek czasu).

Niestety, jak już wspomniano, czasochłonne programowanie zegara w jednoprocesorowych komputerach klasy PC ogranicza wartość minimalnego kwantu czasu do pojedynczych mikrosekund. W systemach wieloprocesorowych sytuacja jest znacznie korzystniejsza z powodu obecności układu regulatora czasowego wysokiej częstotliwości.



Warto wspomnieć, że bardzo precyzyjnej techniki mierzenia upływu czasu dostarcza rodzina procesorów Pentium. Procesory takie i zgodne z nimi posiadają wbudowany licznik wysokiej rozdzielczości (ang. *Time Stamp Counter*), zerowany przy starcie procesora i zwiększany po każdym cyklu zegara taktującego. Ten licznik może być odczytywany przy użyciu specjalnej instrukcji RDTSC (*Read Time Stamp Counter*), która zwraca 64-bitową liczbę taktów procesora. Przepełnienie tego licznika jest praktycznie niemożliwe. Zakładając użycie procesora pracującego z częstotliwością 4 GHz, przepełnienie licznika nastąpi dopiero po ponad 146 latach nieprzerwanej pracy komputera.

W obecnej chwili planista systemu RTLinux udostępnia dwa tryby działania zegara: okresowy (*RTL_CLOCK_MODE_PERIODIC*) i „na żądanie" (*RTL_CLOCK_MODE_ONESHOT*). Okresowe przerwania o częstotliwości 100 Hz są emulowane dla jądra Linuksa. Realizowane jest to stosunkowo prosto: aby zainicjować przerwanie, ustawiany jest odpowiedni bit w zmiennej przechowującej informacje o przerwaniach oczekujących na obsługę. Przy następnym powrocie z dowolnego przerwania (makro *S_IRET*) lub przy „odblokowywaniu" przerwań (makro *S_STI*) procedura obsługi zegara w jądrze Linuksa zostanie wywołana.

Komunikacja międzyprocesowa

Główną zasadą RTLinuksa jest maksymalne uproszczenie i zminimalizowanie rozmiaru tej części aplikacji, która działa w czasie rzeczywistym. Oznacza to, że pierwotna aplikacja powinna być podzielona w taki sposób, że wszelkie operacje wymagające spełnienia ograniczeń czasowych i bezpośredniego dostępu do sprzętu wykonuje zadanie czasu rzeczywistego, zaś całą resztę działań (np. zapis danych na dysk, wizualizacja danych), które nie są z natury ograniczone czasowo, bierze na siebie zwykły proces wykonywany w przestrzeni użytkownika. Konsekwencją takiej architektury jest konieczność zastosowania dodatkowych kanałów komunikacyjnych.

Niestety, do komunikacji pomiędzy zadaniem czasu rzeczywistego i zwykłym procesem nie można wykorzystać (całkiem bogatego zbioru) mechanizmów komunikacji międzyprocesowej, jakie oferuje standardowy Linux. Jego jądro może zostać wywłaszczone w dowolnym momencie. Tak więc żadne wywołanie systemowe Linuksa zmieniające systemowe struktury danych nie może być bezpiecznie użyte przez zadanie czasu rzeczywistego.

RTLinux sam dostarcza więc kilku mechanizmów komunikacji. Najważniejszymi są kolejki czasu rzeczywistego (ang. *real-time FIFOs, RT-FIFOs*) i pamięć dzielona. Do synchronizacji procesów czasu rzeczywistego oraz wzajemnego wykluczania system ten oferuje semafory i muteksy (omówione w podrozdziale „Mechanizmy synchronizacji międzyzadaniowej” w rozdziale 4.).

Kolejki czasu rzeczywistego

Kolejki czasu rzeczywistego (*RT-FIFO*) są buforami alokowanymi w przestrzeni jądra, które realizują algorytm „pierwszy przyszedł, pierwszy obsłużony” (*First In First Out*). Dla odróżnienia od zwykłych kolejek, będących standardowym mechanizmem IPC (ang. *inter-process communication*) Linuksa, kolejki czasu rzeczywistego będą nazywane RT-FIFO. Mogą być odczytywane i zapisywane zarówno przez procesy linuksowe, jak i zadania RTLinuksa. Kolejki RT-FIFO są jednokierunkowe: aplikacja nie może wykorzystywać tej samej kolejki do czytania i zapisywania jednocześnie. Do uzyskania dwukierunkowego połączenia należy użyć dwóch kolejek.

Kolejki RT-FIFO są urządzeniami znakowymi o numerze głównym (ang. *major number*) 150. Są tworzone podczas instalacji RTLinuksa i istnieją w katalogu */dev/* jako urządzenia o nazwach */dev/rtf0*, */dev/rtf1*, i tak dalej, aż do */dev/rtfN*, gdzie *N* (standardowo 64) oznacza maksymalną liczbę kolejek czasu rzeczywistego ustaloną podczas kompilacji systemu. Wymienione urządzenia są obsługiwane przez osobne moduły RTLinuksa.

Po stronie czasu rzeczywistego na interfejs kolejek RT-FIFO składają się operacje: tworzenie, niszczenie, czytanie i zapis kolejki. Odczyt i zapis są operacjami niepodzielnymi i nieblokującymi. Dla zwykłych procesów Linuksa kolejki RT są widoczne jako zwykłe pliki o wspomnianych wyżej nazwach, dlatego do operowania na nich (wykonywania operacji we/wy na plikach) może zostać użyte standardowe API (ang. *Application Programing Interface*). W najprostszym przypadku dostęp do kolejek RT-FIFO można zrealizować przy pomocy skryptu powłoki lub polecenia jednowierszowego.

Pamięć dzielona

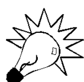
Pamięć dzielona może służyć do wymiany większych ilości danych pomiędzy zadaniem czasu rzeczywistego a zwykłym procesem. Jest to najszybszy sposób komunikacji pomiędzy zadaniami, ze względu na prostotę i brak zaangażowania systemu w realizację tej techniki, nie uwzględniając oczywiście mechanizmu odwzorowywania pamięci i przechwytywania wyjątków.

Używanie pamięci wspólnej pozwala wielu zadaniom korzystać z tych samych danych. Niestety, zapis do pamięci dzielonej wymaga postępowania według określonego protokołu w celu zachowania integralności danych. W przeciwieństwie do kolejek, przy użyciu pamięci wspólnej możliwa jest wymiana danych w postaci struktur. Jedynym limitem rozmiaru pamięci dzielonej jest rozmiar fizycznej pamięci dostępnej w systemie komputerowym.

Synchronizacja i wzajemne wykluczanie

W każdym środowisku wielozadaniowym procesy mogą na siebie oddziaływać. W przypadku korzystania ze wspólnej przestrzeni adresowej współbieżny dostęp do danych dzielonych może powodować ich niespójność. W celu zapewnienia wyłącznego dostępu do zasobu konieczne jest użycie pewnych mechanizmów wzajemnego wykluczania i synchronizacji. Dla systemów czasu rzeczywistego, oprócz zachowania integralności zasobu dzielonego i uniknięcia zakleszczeń, ważnym problemem jest minimalizacja inwersji priorytetów.

Inwersja priorytetów oznacza fakt zajmowania zasobu dzielonego przez zadanie o niskim priorytecie, podczas gdy w systemie istnieje zadanie o wyższym priorytecie, oczekujące na dostęp do tego samego zasobu.



Zakleszczenie (ang. *deadlock*) jest sytuacją, w której zbiór procesów czeka w nieskończoność na zdarzenie, które może być spowodowane tylko przez jeden z oczekujących procesów.

Przykład 2.2. Inwersja priorytetów

Rozważmy dla przykładu dwa procesy okresowe, które chcą mieć wyłączny dostęp do pewnego logicznego zasobu. W systemie ze statycznym priorytetem zadań może dojść do sytuacji, gdzie niskopriorytetowy proces zarygluje (ang. *lock*) zasób i zostanie wywłaszczony przez proces o wyższym priorytecie.

Kiedy drugi proces będzie próbował uzyskać dostęp do zasobu, zostanie zablokowany. Jeśli w tym momencie pojawi się trzeci proces, o średnim priorytecie, wywłaszczy proces niskopriorytetowy i będzie się wykonywać kosztem procesu o najwyższym priorytecie. Ponieważ zadanie o średnim priorytecie wykonuje się, gdy istnieje zadanie o wyższym priorytecie, *de facto* priorytety zadań są odwrócone (ang. *priority inversion*).

Dla uniknięcia takiego niekorzystnego zjawiska w literaturze [4, 22] poświęconej tej tematyce zostało zaproponowanych kilka modeli dostępu do zasobów dzielonych. Dla pokazania istoty problemu rozważmy dwa z nich, przy założeniu, że dla procesów (zadań) i zasobów prawdziwe są następujące stwierdzenia:

- do procesów przypisane są statyczne priorytety,
- zasoby są dostępne w sposób wzajemnie wykluczający się,
- używany jest *scheduler* planujący zadania na podstawie ich priorytetu (gdzie procesor dostaje gotowe zadanie o najwyższym priorytecie),
- zasoby, do których proces ma dostęp, są znane z góry, przed uruchomieniem procesu.

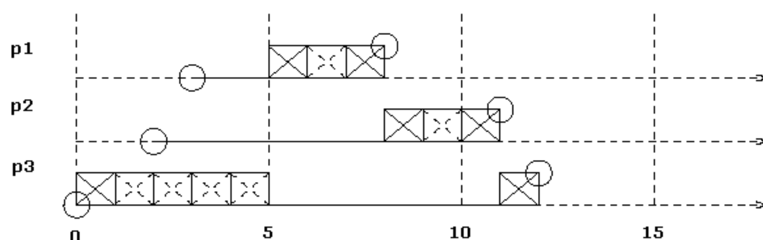
Najprostszą techniką dostępu do *danych dzielonych* wydaje się być protokół bazujący na dziedziczeniu priorytetów (ang. *priority-inheritance protocol*), w którym proces używający zasobu potrzebnego procesowi o wysokim priorytecie dziedziczy jego priorytet dopóki, dopóty nie przestanie korzystać z zasobu będącego przedmiotem sporu. Po zwolnieniu zasobu jego priorytet uzyskuje pierwotną wartość, zatem możliwość wywłaszczenia procesu niskopriorytetowego w jego sekcji krytycznej przez proces o średnim priorytecie zostaje wyeliminowana. Niestety, takie rozwiązanie niesie ze sobą poważne zagrożenia. Jeśli w rozważanym poprzednio przykładzie proces, który odziedziczył wysoki priorytet w sposób zagnieżdżony, zażąda dostępu do drugiego zasobu aktualnie używanego przez zablokowany proces o wysokim priorytecie, nastąpi zakleszczenie (ang. *impas*).

Ponieważ dziedziczenie priorytetów nie eliminuje niedopuszczalnego w systemach czasu rzeczywistego zakleszczenia procesów, twórcy systemu RTLinux nie stosują tego protokołu w obiektach synchronizacji, służących do ochrony zasobów dzielonych. Z całkiem bogatej rodziny protokołów dostępu, bazujących na stałym priorytecie, RTLinux implementuje protokół CSP (ang. *Ceiling Semaphore Protocol*). Działanie protokołu opiera się na pojęciu *pułapu priorytetu*, przypisanego do zasobu. Pułap zasobu ma wartość równą najwyższemu priorytetowi spośród procesów, które mogą go zająć. Idea działania protokołu CSP polega na ustawianiu priorytetu procesu, który posiadł zasób, na wartość pułapu tego zasobu. Proces działa z priorytetem równym pułapowi zasobu aż do jego zwolnienia.

Przykład 2.3. Działanie protokołu CSP

Rozważmy dla przykładu procesy *p1*, *p2* i *p3* o relacji priorytetów *p1>p2>p3*, które próbują uzyskać wyłączny dostęp do zasobu *R*, jak na rysunku 2.2.

Rysunek 2.2. *Działanie protokołu Ceiling Semaphore Protocol [22]*



Rysunek 2.2 prezentuje następującą sekwencję zdarzeń, przy założeniu, że pułap zasobu *R* jest równy 1:

t = 0: pojawia się *p3* i wykonuje się;

t = 1: *p3* żąda i zajmuje zasób *R*, *p3* dziedziczy pułap priorytetu równy 1;

t = 2: pojawia się *p2*, lecz *p3* wykonuje się dalej, jako proces o wyższym priorytecie;

t = 3: pojawia się *p1*, lecz dalej wykonuje się *p3*, ponieważ *p1* nie ma wyższego priorytetu;

t = 4: *p3* kończy sekcję krytyczną zwalnia *R* i wraca do poprzedniego priorytetu;

t = 5: wykonuje się *p1*;

t = 6: *p1* alokuje *R* ;

t = 7: *p1* kończy sekcją krytyczną, wykonuje się i kończy;

t = 8: wykonuje się *p2*;

t = 9: *p2* zajmuje *R*;

t =10: *p2* kończy sekcję krytyczną, wykonuje się i kończy;

t =11: *p3* wykonuje się i kończy.

Protokół CSP posiada kilka zalet:

- Po pierwsze, protokół zabezpiecza przed wystąpieniem zakleszczenia.
- Po drugie, unika częstego przełączania kontekstu — w porównaniu z innymi protokołami ze swojej klasy.
- Trzecią zaletą jest zredukowana złożoność w trakcie wykonywania, protokół jest przejrzysty i przewidywalny.

Wadą, jakiej można się dopatrzyć, jest zwiększony czas odpowiedzi, można to zauważyć w przykładzie 2.3. Jeśli proces *p3* zajmie zasób *R*, jego priorytet zostanie podwyższony do pułapu *R*, w tym wypadku do 1. Wykonywanie procesu *p2* zostanie zablokowane, nawet jeśli nie będzie oczekiwał na zwolnienie zasobu.

Obecnie RTLinux dostarcza podstawowych mechanizmów synchronizacji i wzajemnego wykluczania w postaci semaforów i muteksów zgodnych z POSIX. Oprócz podstawowych operacji ryglowania (ang. *lock*) i zwalniania zasobu (ang. *unlock*) za pomocą semaforów możliwe jest również ryglowanie warunkowe. W takim wypadku przy wykonywaniu operacji „czekaj” na semaforze podawany jest limit czasu, przez jaki zadanie może czekać na podniesienie semafora. Muteksy (ang. *mutex — mutual exclusion*) mogą opcjonalnie zostać wyposażone w atrybuty pozwalające na realizację omawianego protokołu CSP przy dostępie do zasobów. Możliwość korzystania z tego protokołu jest ustalana przy konfiguracji i instalacji RTLinuksa.