Funkcje systemów operacyjnych czasu rzeczywistego

Witold Paluszyński
Katedra Cybernetyki i Robotyki
Wydział Elektroniki, Politechnika Wrocławska
http://www.kcir.pwr.edu.pl/~witold/

Systemy operacyjne czasu rzeczywistego

System operacyjny czasu rzeczywistego (RTOS) służy do implementacji systemów czasu rzeczywistego. Taki system musi nie tylko dostarczać mechanizmów i usług dla wykonywania, planowania, i zarządzania zasobami aplikacji, ale powinien również sam sobą zarządzać w sposób oszczędny, przewidywalny, i niezawodny.

System operacyjny czasu rzeczywistego powinien być modularny i rozszerzalny. W przypadku systemów wbudowanych jądro systemu musi być małe, ze względu na lokalizację w ROM, i często ograniczoną wielkość pamięci RAM. W przypadku systemów, dla których krytycznym aspektem jest bezpieczeństwo, niekiedy wymagana jest certyfikacja całego systemu (łącznie z aplikacją), jak również samego systemu operacyjnego.

Podstawowymi zaletami jest prostota i oszczędność. Dlatego RTOS może mieć mikrojądro dostarczające tylko podstawowych usług planowania, synchronizacji, i obsługi przerwań. Gdy niektóre aplikacje wymagają większego spektrum usług systemowych (systemu plików, systemu wejścia/wyjścia, dostępu do sieci, itp.), RTOS zwykle dostarcza tych usług w postaci modułów dołączonych do jądra. Praca tych modułów również musi podlegać monitorowaniu.

Ze względu na konieczność zaprojektowania ścisłego budżetu czasu i zasobów całego systemu, aplikacja jest często budowana jako całość, i system ma charakter zamknięty. Możliwa jest również architektura otwarta, gdzie silne (*hard*) aplikacje czasu rzeczywistego współistnieją ze słabymi (*soft*) aplikacjami czasu rzeczywistego, jak również z aplikacjami nie wymagającymi czasu rzeczywistego.

Jądra, mikrojądra, i nanojądra

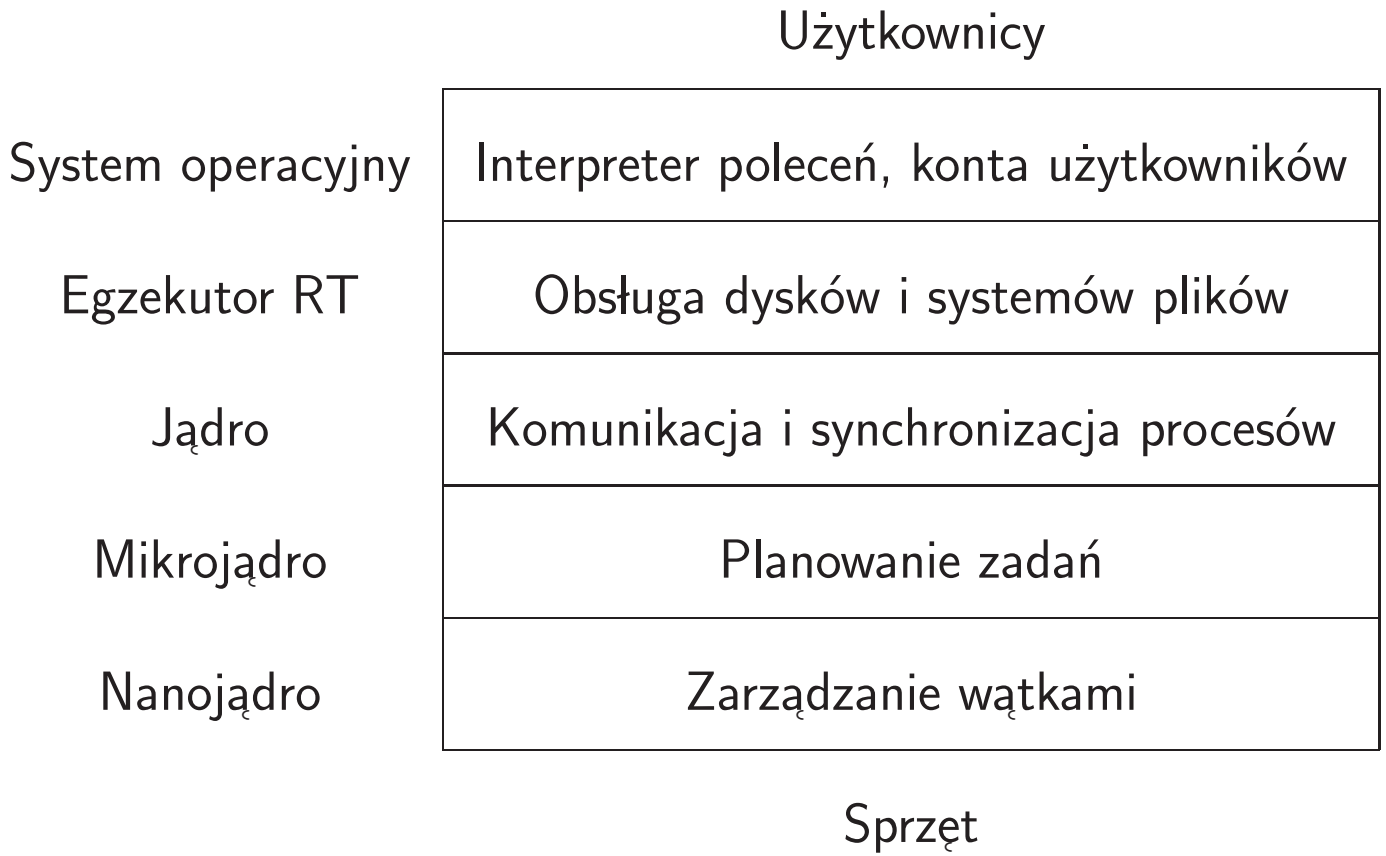
Jądro (*kernel*) systemu RTOS zawiera moduły realizujące co najmniej następujące trzy funkcje: **planowanie** (*scheduling*), **dyspozycja** (*dispatching*), i usługi komunikacji i synchronizacji.

Planista (*scheduler*) realizuje algorytm decydujący które zadanie ma być uruchomione w następnej kolejności w systemie wielozadaniowym. **Dyspozytor** (*dispatcher*), zwany również **ekspedytorem** administruje strukturami niezbędnymi do uruchamiania zadań. Mechanizmy komunikacji i synchronizacji obejmują: semafory, monitory, kolejki komunikatów, i inne.

Mikrojądrem nazywa się jądro o funkcjonalności ograniczonej do modułu planowania i dyspozytora. Inaczej mówiąc, mikrojądro jest w stanie realizować wielozadaniowość na poziomie procesów.

Nanojądro obsługuje jedynie zarządzanie wątkami.

Egzekutor RT i system operacyjny



Typowe komercyjne systemy RTOS są **egzekutorami RT** (*realtime executive*).

Pseudojądra

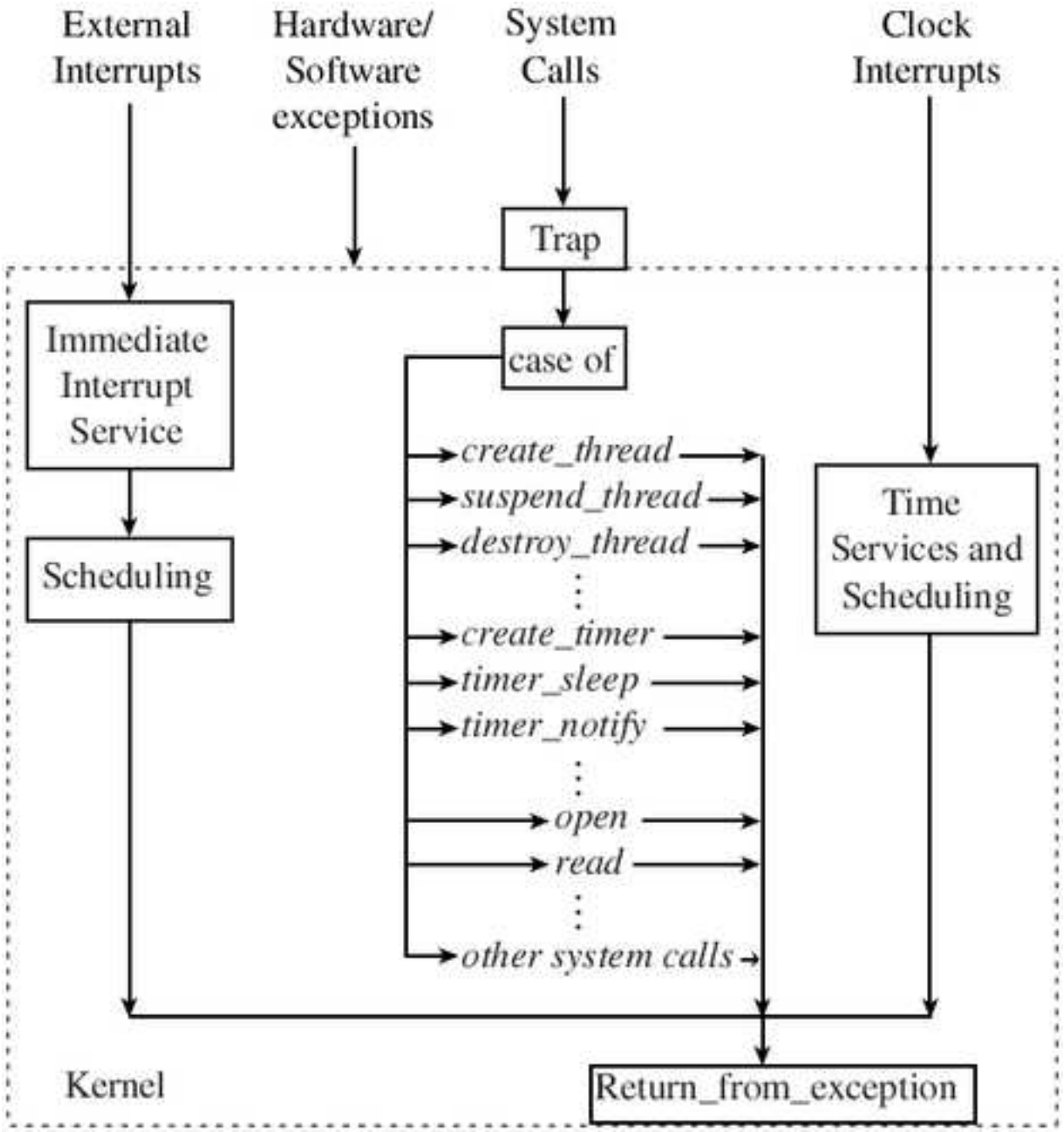
Wielozadaniowość czasu rzeczywistego może być osiągnięta bez przerwań, a nawet bez wsparcia systemu operacyjnego. Podejście takie ma wiele zalet, np. otrzymane systemy można łatwiej analizować. Architekturę systemu niekorzystającego z usług systemu operacyjnego, realizującego samodzielnie funkcje wielozadaniowości, nazywa się architekturą **pseudojądra**.

- pętla odpytywania (*polling*)
- synchronizowana pętla odpytywania
- cykliczny egzekutor
- maszyna stanów
- koprocesy

Funkcje (mikro)jądra

System operacyjny czasu rzeczywistego typowo składa się z mikrojądra zapewniającego podstawowe usługi systemowe. Mikrojądro wywłaszcza wykonujące się zadanie, aby wykonywać samo siebie, w następujących przypadkach:

- obsługa wywołania systemowego,
- obsługa timerów i planowanie zadań,
- obsługa przerwań zewnętrznych,
- obsługa wyjątków sprzętowych i programowych.


External Interrupts
Hardware/ Software exceptions
System Calls
Clock Interrupts
Trap
Immediate Interrupt Service
case of
Scheduling
create_thread
suspend_thread
destroy_thread
create_timer
timer_sleep
timer_notify
open
read
other system calls
Time Services and Scheduling
Kernel
Return_from_exception

Obsługa wywołań systemowych

Obsługa wywołania systemowego składa się z zapamiętaniu kontekstu wykonującego się zadania, przełączenia się w tryb jądra, i wykonaniu funkcji z argumentami znajdującymi się na szczycie stosu zadania. Po zakończeniu wykonywania funkcji jądro wykonuje powrót z wyjątku, co powoduje powrót do trybu użytkownika.

Systemy wbudowane często nie posiadają ochrony pamięci, która narzuca istotne wymagania pamięciowe (rzędu kilku kilobajtów na proces), jak również wydłużony czas przełączania kontekstu. Brak ochrony pamięci jest często akceptowalny w środowisku aplikacji wbudowanych.

Obsługa funkcji czasu

Centralną częścią mikrojądra jest planista. Wykonywany jest zawsze gdy zmienia się stan któregoś z zadań, jak również okresowo, w wyniku przerwania zegarowego. Okres przerwania zegarowego nazywany jest **tikiem** (*tick*). Tik stosowany przez większość systemów operacyjnych wynosi 10 milisekund.

Operacje wykonywane w momencie przerwania zegarowego:

1. Przetwarzanie zdarzeń czasowych — sprawdzanie kolejki timerów i zakolejkowanie zdarzeń dla tych, które zostały przeterminowane.

2. Aktualizacja budżetu czasowego zadania aktywnego, o ile jest ono planowane algorytmem RR (albo innym wywłaszczającym).

3. Aktualizacja kolejki zadań gotowych — pewne zadania mogły stać się gotowe w wyniku wyzwolenia, a zadanie wykonywane mogło zostać wywłaszczone. Do wykonywania wybierane jest pierwsze zadanie z kolejki o najwyższym priorytecie.

Jądro może również aktualizować liczniki czasu CPU zadań — zarówno bezpośrednio zużyty, jak również zużyty przez jądro na usługi dla zadania.

Jak widać, planista wykonuje swoje krytyczne zadania z okresem równym tikowi, co jest na ogół wystarczające dla zadań pracujących w podziale czasu, ale może być niewystarczające dla zadań czasu rzeczywistego.

Zmniejszenie wielkości tiku mogłoby rozwiązać ten problem, ale oczywiście kosztem znacznego wzrostu narzutów planowania. Dlatego większość systemów stosuje planowanie czasowe w połączeniu ze zdarzeniowym. Jądro wywołuje planistę dodatkowo zawsze wtedy gdy budzone, wyzwalane, lub odblokowywane jest jakieś zadanie, albo tworzone jest nowe.

Obsługa przerwań zewnętrznych

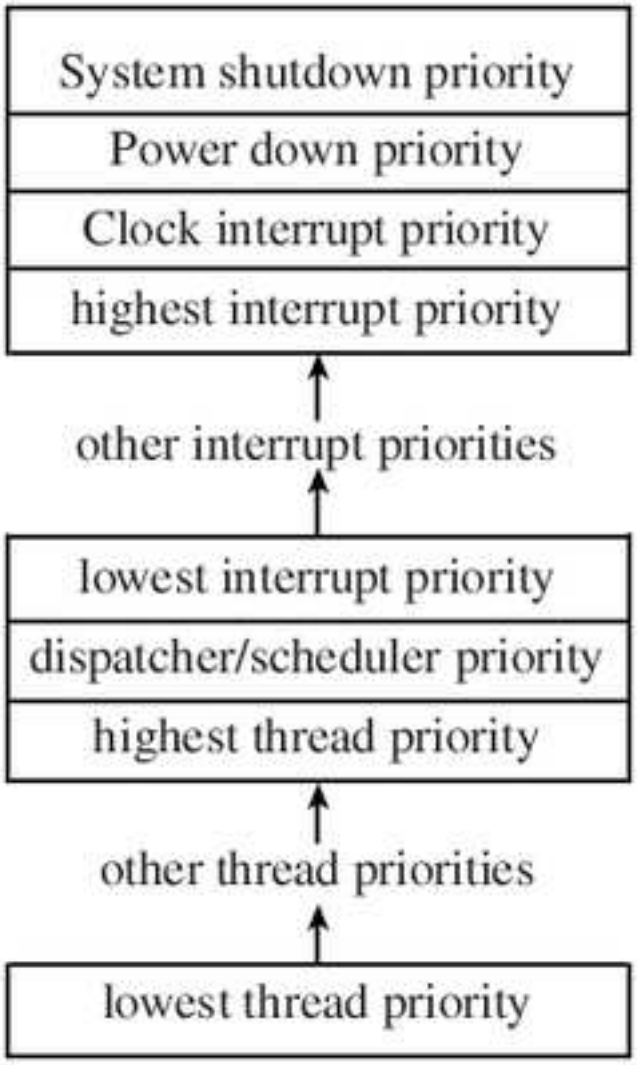
Przerwania sprzętowe stanowią mechanizm powiadamiania aplikacji o zdarzeniach zewnętrznych i obsługi sporadycznych operacji I/O. Nakład czasu niezbędny do obsługi przerwania zależy od jego źródła.

W szczególności, obsługa przerwań DMA wymaga znacznego czasu. Na przykład, dla obsługi nadchodzącego pakietu sieciowego, interfejs sieciowy generuje przerwanie. Dla jego obsługi jądro musi wywołać funkcję obsługi odpowiedniego protokołu. Ta z kolei identyfikuje zadanie, które ma otrzymać pakiet, kopiuje go do bufora w przestrzeni adresowej zadania, wykonuje dodatkowe czynności wymagane przez protokół (np. generuje komunikat potwierdzenia), itp. Wymagany na to czas może być długi i trudny do określenia.

Procedury obsługi przerwań generowanych przez system dyskowy i sieciowy typowo trwają setki mikrosekund do dziesiątek milisekund. Dlatego większość systemów dzieli ich obsługę na dwie części: natychmiastową i planowaną. Jest to tzw. **podzielona obsługa przerwań** (*split interrupt handling*).

Natychmiastowa obsługa przerwań

Pierwsza faza obsługi przerwania, zwana **natychmiastową obsługą przerwania**, wykonywana jest z zastosowaniem właściwego priorytetu. Priorytety przerwań są ogólnie wyższe niż priorytety zadań, jak również wyższe niż priorytet planisty.



W chwili przerwania, procesor wpisuje rejestry PC i stanu na stos przerwań, i wywołuje procedurę obsługi przerwania jądra. Jądro najpierw maskuje przerwania w celu bezpiecznego wykonania operacji na strukturach danych związanych z aktualnym przerwaniem. Następnie zapisuje stan procesora na stosie przerwań, odblokowuje przerwania, i dopiero wtedy wykonuje właściwy kod obsługi przerwania.

Więcej niż jedno urządzenie może być podpięte pod jedną linię przerwania. W takim przypadku wywoływane są po kolei procedury obsługi ich wszystkich. Procedura urządzenia, które nie wygenerowało przerwania wraca natychmiast.

Jeśli w czasie obsługi przerwania pojawia się inne przerwanie o wyższym priorytecie, to jest obsługiwane dokładnie w taki sam sposób.

Opóźnienie obsługi przerwania

Całkowity czas, jaki upływa od chwili wygenerowania przerwania, do momentu rozpoczęcia wykonywania jego obsługi, jest nazywany **opóźnieniem obsługi przerwania** (*interrupt latency*). Stanowi on miarę reaktywności systemu na zdarzenia zewnętrzne. Wielkość tego czasu jest sumą następujących elementów:

1. czas na dokończenie aktualnej instrukcji, obsługę stanu procesora, i zainicjowanie obsługi przerwania,
2. czas na zamaskowanie przerwań,
3. czas na natychmiastową obsługę przerwań o wyższym priorytecie, jeśli takie wystąpiły,
4. czas na wyskładowanie kontekstu wykonywanego zadania, i ustalenia które urządzenie zgłosiło przerwanie,
5. czas na wywołanie procedury obsługi.

Największą (i nieprzewidywalną) rolę odgrywa tu element 5.

Jedynym sposobem na zwiększenie reaktywności systemu jest zminimalizowanie całej procedury natychmiastowej obsługi przerwania.

Planowana obsługa przerwań

Oprócz przypadków bardzo trywialnych funkcji, natychmiastowa obsługa przerwania nie kończy jego właściwej obsługi. Właściwa faza obsługi przerwania — **planowana obsługa przerwania** — powinna być wywłaszczalna, i wykonywana z priorytetem odpowiednim dla danego urządzenia. Na przykład, może być on równy priorytetowi zadania, które zainicjowało operacje na urządzeniu. Dlatego na diagramie zadań jądra, po wykonaniu natychmiastowej obsługi przerwania wywoływany jest planista, który umieszcza zadanie planowanej obsługi przerwania w kolejce zadań gotowych.

Zadanie planowanej obsługi przerwania stanowi zadanie aperiodyczne lub sporadyczne.

Usługi i mechanizmy dla planowania

System operacyjny dostarcza zestaw standardowych algorytmów planowania, takich jak FIFO albo RR. Innym typowym mechanizmem jest planowanie priorytetowe z wywłaszczaniem. Często aplikacja użytkownika ma powody ingerować w procedurę planowania zadań, i systemy operacyjne dostarczają mechanizmów, które to umożliwiają.

Przykładem może być mechanizm blokowania wywłaszczania, dostarczany przez wiele systemów czasu rzeczywistego.

Usługi komunikacji i synchronizacji

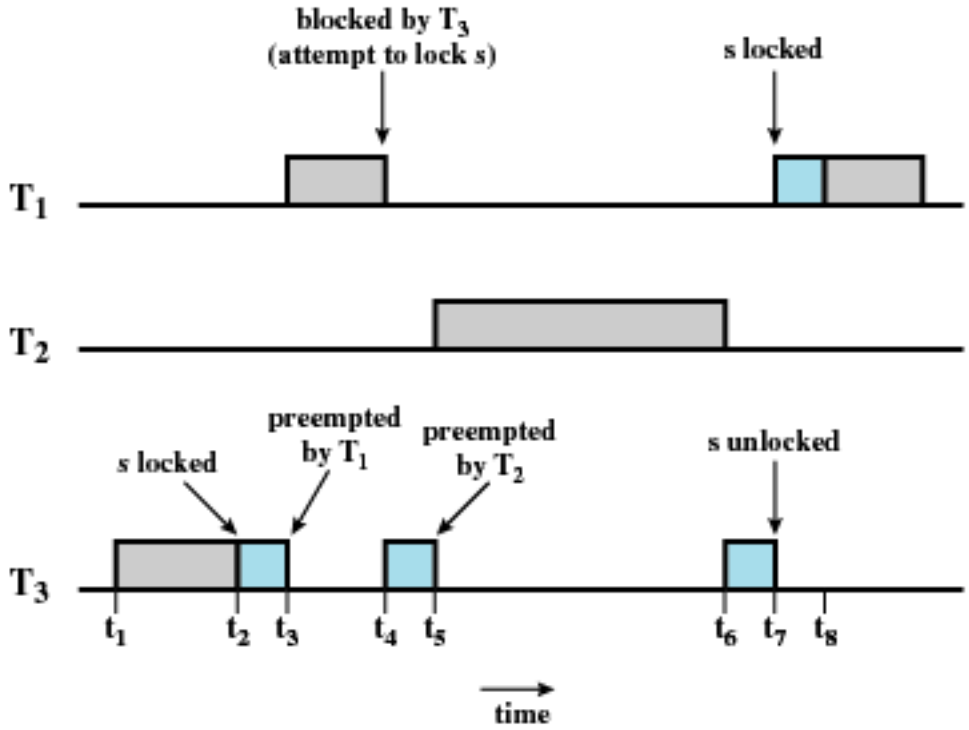
System operacyjny dostarcza usług komunikacji i synchronizacji dla zadań, takich jak: kolejki komunikatów, pamięć współdzielona, muteksy, blokady zapisu i odczytu, i inne.

Dodatkowym mechanizmem wspomagającym sprawną pracę mechanizmów synchronizacji, są algorytmy zapobiegania i unikania zakleszczeń.

W systemach czasu rzeczywistego dodatkową okolicznością są priorytety zadań. Czasami w sytuacjach związanych z komunikacją i synchronizacją niezbędne jest zastosowanie algorytmów zmieniających priorytety pewnych zadań w celu zapewnienia sprawnej realizacji komunikacji, albo synchronizacji.

Odwrócenie (inwersja) priorytetów

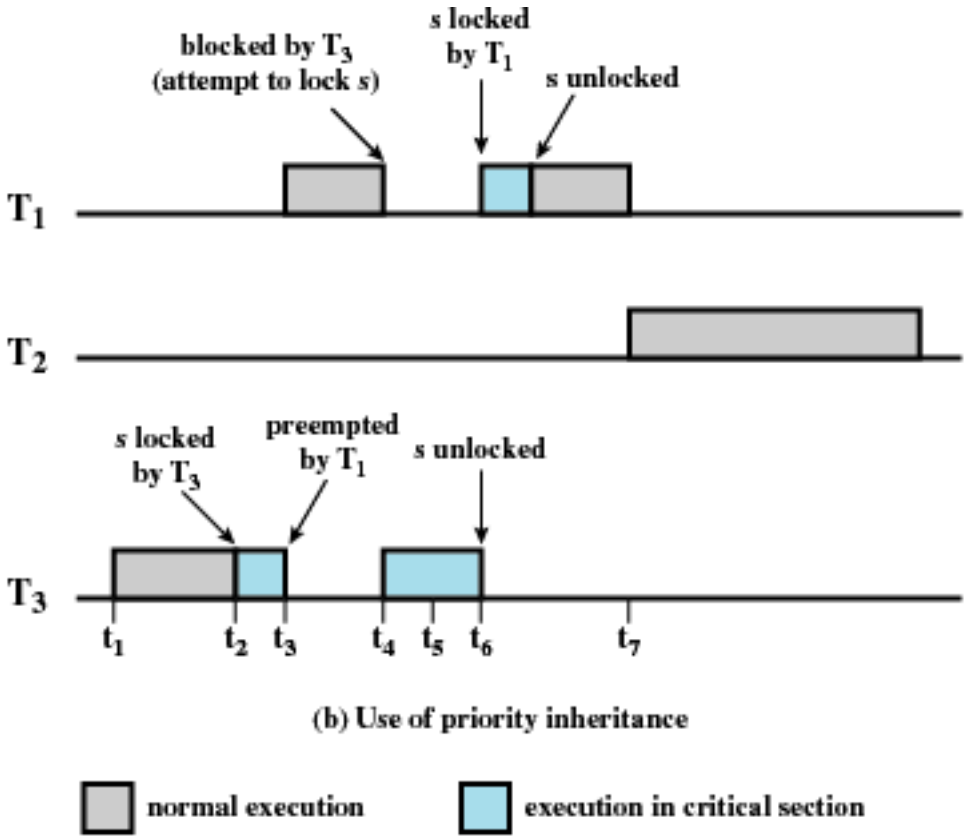
Jak widać na rysunku, zadanie o niskim priorytecie T_3 poprawnie uzyskuje dostęp do jakiegoś zasobu s (semafora), po czym zostaje wywłaszczone przez zadanie T_1 o wysokim priorytecie. Jednak gdy T_1 zgłasza żądanie dostępu do s, wykonywanie wraca do T_3. Ta sytuacja, wynikająca z interakcji pomiedzy T_1 a T_3 może jednak nadal być poprawna — programista wiedział, że zadania T_1 i T_3 współdzielą zasób.



Problem powstaje dopiero, gdy zadanie T_1 zostanie zablokowane na czas dowolnie długi, gdyż T_3 może być wywłaszczone przez inne zadania, takie jak T_2, o priorytecie niższym niż T_1, powodując **odwrócenie (inwersję) priorytetów**.

Odwrócenie priorytetów — dziedziczenie priorytetów

Metoda **dziedziczenia priorytetów** polega na tym, że w momencie gdy zadanie o wyższym priorytecie T_1 zostaje zablokowane w oczekiwaniu na zasób s zajęty przez zadanie T_3 o niższym priorytecie, T_3 chwilowo dziedziczy wysoki priorytet T_1. T_3 nie podlega wtedy wywłaszczeniu przez zadania typu T_2.



Odwrócenie priorytetów — pułap priorytetów

Inna metoda rozwiązania problemu inwersji priorytetów, zwana **pułapem priorytetów** (*priority ceiling*) polega na przypisaniu priorytetów zasobom, przy czym najniższy priorytet zasobu jest wyższy od najwyższego priorytetu wszystkich zadań. W chwili zajęcia zasobu zadanie uzyskuje chwilowo priorytet równy temu zasobowi, i może wykonywać się do chwili zwolnienia zasobu, kiedy jego priorytet jest obniżany do pierwotnego poziomu.

Odwrócenie priorytetów — misja Pathfindera

Misja sondy marsjańskiej Pathfinder wystrzelonej w 1996 roku jest znana z kilku powodów, z których jednym jest wyjątkowo krótki czas przygotowania i skromny budżet — 150 M$ — porównywalny z budżetem niektórych produkcji filmowych. Innym symbolem tej misji był łazik marsjański Sojourner, który wylądował na Marsie 4 lipca 1997, i wykonał wysokiej rozdzielczości zdjęcia powierzchni Marsa.

Jednak misja została zagrożona przez błąd w systemie sterującym lądownika, który został wcześniej zaobserwowany w testach poprzedzających start, lecz zlekceważony, ponieważ nie miał związku z oprogramowaniem lądowania.

Po wylądowaniu Pathfinder rozpoczął gromadzenie danych meteorologicznych (między innymi). W trakcie tych operacji zaczęło dochodzić to pełnego restartu jego systemu, co powodowało utratę danych i opóźnienia.

W dniach 5-14 lipca 1997 system wykonał cztery restarty, i powstało zagrożenie dla kontynuacji właściwej pracy.

Pathfinder posiadał magistralę do przesyłania danych. Niezbyt często wykonywane zadanie gromadzenia danych meteorologicznych miało niski priorytet, lecz zajmowało magistralę do przesłania tych danych, poprawnie blokując mutex. Najwyższy priorytet miało zadanie obsługi magistrali, lecz okresowo musiało krótko czekać na zwolnienie magistrali przez zadanie meteorologiczne. W tym krótkim oknie czasowym mogło wystąpić przerwanie wywołujące zadanie komunikacyjne średniego priorytetu, które nie zajmowało magistrali.

Przedłużający się czas nieaktywności zadania wysokiego priorytetu spowodowało przeterminowanie timera watchdoga, który zainicjował pełny restart systemu. Zadania zaplanowane na dany dzień zostały wtedy przesunięte na następny dzień.

Jest to zatem klasyczny scenariusz inwersji priorytetów.

Komputer na pokładzie lądownika był oparty na procesorze IBM Risc 6000 Single Chip (wersja uodporniona na promieniowanie kosmiczne), z 128MB RAM, 6MB EEPROM, i systemie operacyjnym VxWorks. System ten posiada mechanizm dziedziczenia priorytetów, lecz domyślna konfiguracja ma ten mechanizm wyłączony.

Problem został zdiagnozowany przez intensywne testy na dokładnej kopii systemu łazika, które pozwoliły zduplikować zjawisko. 21 lipca 1997 udało się wgrać poprawkę włączającą dziedziczenie priorytetów, i misja mogła być kontynuowana.

Zarządzanie pamięcią

Systemy czasu rzeczywistego często ignorują kwestie zarządzania pamięcią, ponieważ związane z nim algorytmy są często absorbujące i trudno przewidywalne. W najprostszym przypadku można założyć, ze system będzie budowany w taki sposób, że w całości będzie rezydował w pamięci RAM. Zadanie nie zostanie utworzone, jeśli system nie zdoła zaalokować pamięci wystarczającej dla szczytowych wymagań tego zadania.

W szczególności, systemy operacyjne czasu rzeczywistego często nie zapewniają obsługi pamięci wirtualnej: odwzorowania wirtualnej przestrzeni adresowej dla procesów, stronicowania na żądanie, ani wymiatania. W zależności od konkretnych usług zarządzania pamięcią można podzielić systemy operacyjne na klasy.

Alokacja i dealokacja pamięci

Podstawową usługą dotyczącą pamięci dostarczaną przez systemy operacyjne, w tym również systemy RTOS, jest alokacja i dealokacja pamięci. Program żądający w trakcie pracy dynamicznego przydziału dodatkowych bloków pamięci dostaje je od systemu, a po wykorzystaniu zwalnia je, pozwalając systemowi wykorzystać je do innych celów.

Najprostszymi metodami alokacji pamięci są metody alokacji ciągłej, przydzielające jednolite bloki pamięci. Ich wadą jest fragmentacja pamięci. Po przydzieleniu pewnej liczby bloków o zmiennej wielkości, uzyskanie kolejnego większego bloku może nie być możliwe, w sytuacji kiedy mniejsze bloki są nadal wolne. Problem fragmentacji rozwiązuje się przez defragmentację, która jednak w systemach czasu rzeczywistego jest trudna, ponieważ zarówno konieczność jej wykonania jak i potrzebny nakład czasu są trudne do przewidzenia.

Alokacja stronicowana

Fragmentacja nie występuje w systemach alokacji stronicowanej, które dzielą pamięć fizyczną na niewielkie bloki zwane ramkami, z jednoczesnym podziałem przestrzeni adresowej procesu na identycznego rozmiaru bloki zwane stronami. Każdy proces posiada tablicę stron, i każde odwołanie do pamięci podlega translacji adresu z wykorzystaniem tej tablicy.

Jednak nieciągłość pamięci fizycznej i translacja adresów komplikuje operacje I/O wykonywane przez DMA. Operacje DMA są znacznie prostsze i wymagają mniejszych narzutów, jeśli wykorzystują ciągły obszar pamięci fizycznej.
W przypadku nieciągłych obszarów pamięci, transfer DMA musi być podzielony na szereg transferów, z których każdy musi być indywidualnie zainicjowany przez procesor.

Najczęściej stosowanym systemem alokacji stronicowanej jest system pamięci wirtualnej. Zasadniczą wadą tego mechanizmu z punktu widzenia systemów czasu rzeczywistego jest nieprzewidywalność czasu dostępu do pamięci w przypadku błędu strony.

Blokowanie pamięci

Są jednak powody, dla których system operacyjny RTOS może stosować pamięć wirtualną. Na przykład, dla wsparcia narzędzi do tworzenia oprogramowania, jak edytory, debuggery, profilery, które typowo mają duże wymagania pamięciowe, choć nie wymagają pracy w czasie rzeczywistym. Dla zapewnienia współpracy aplikacji czasu rzeczywistego z takimi aplikacjami czasu podzielonego, system RTOS musi dostarczać mechanizmów pozwalających na kontrolowanie stronicowania.

Specyfikacja POSIX dostarcza mechanizmu **blokowania** pamięci. Zadanie może zarówno zablokować cały swój kod w pamięci fizycznej, albo pewien zakres adresowy (co wymaga dobrej znajomości rozlokowania aplikacji w pamięci). W ten sposób zadanie czasu rzeczywistego może pracować w systemie pamięci wirtualnej, ale jego pamięć nie będzie podlegać usuwaniu z pamięci RAM.

Zabezpieczenie pamięci

Inną podstawową usługą zapewnianą przez systemy operacyjne ogólnego przeznaczenia jest zabezpieczenie obszarów pamięci. Wiele systemów RTOS nie zapewnia mechanizmów zabezpieczenia pamięci dla zadań jądra ani aplikacja użytkownika. Jak zwykle, argumentem za stosowaniem pojedynczej przestrzeni adresowej jest prostota (z punktu widzenia systemu) i mniejsze narzuty tego rozwiązania.

Niestety, te zalety okupione są większą komplikacją projektu aplikacji użytkownika, zwłaszcza, gdy trzeba wprowadzać modyfikacje. Zatem to rozwiązanie jest właściwe jedynie dla bardzo niewielkich systemów wbudowanych. Natomiast wiele systemów RTOS zapewnia ochronę pamięci procesów.

Pewną alternatywą jest możliwość konfiguracji mechanizmów pamięci wirtualnej oferowana przez niektóre systemy RTOS.

Bibliografia

Jane W.S. Liu: Real-Time Systems, Prentice-Hall 2000